**Lesson 3: Designing User-Defined Functions**

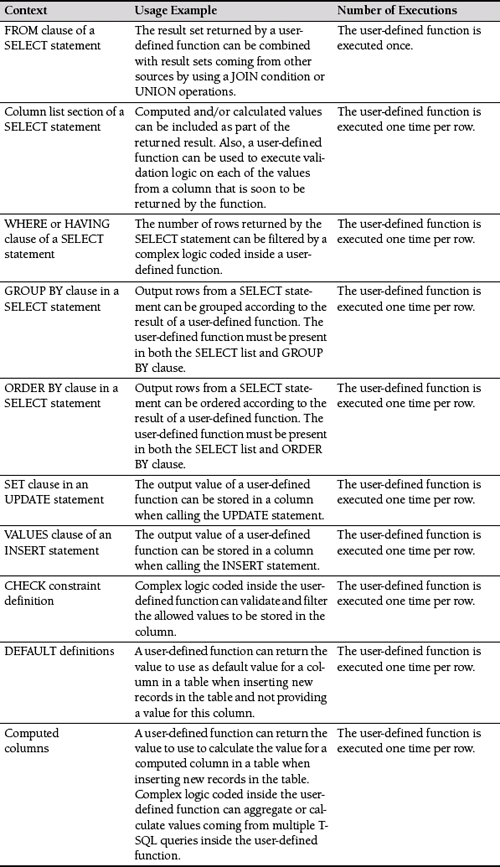
Estimated lesson time: 45 minutes

As you saw in the previous lesson, stored procedures provide the capability to include programming logic to generate a dynamic response—either a scalar value or a result set. In this lesson, you will learn how UDFs offer the same functionality with a twist: the possibility for this functionality to be called from different contexts than stored procedures.

Similar to views and stored procedures, UDFs do not store data. A UDF stores T-SQL queries that are executed when the function is called. As with stored procedures, the T-SQL code inside a function is stored in pre-compiled format, so a query plan has already been calculated, which translates into better execution performance.

However, unlike stored procedures, functions enable you to reuse their result in much more flexible ways than stored procedures do. A stored procedure is called by using the EXECUTE statement; it’s impossible to call a stored procedure from a SELECT statement. In contrast, UDFs can be called from multiple contexts.

Table 6-2 summarizes some of the different contexts from which UDFs can be executed.



**Table 6-2** Executing User-Defined Functions in SQL Server 2005

When designing UDFs, there are several design decisions you need to consider:

* What type of UDF do you need?
* What type of data will the UDF return?
* What kind of input parameters do you need to define for the UDF?
* What error handling routines do you need to include?
* What security context should the UDF execute under?

**Exam Tip**

Unlike stored procedures, UDFs cannot be used to perform actions that modify the database state. The T-SQL code inside a UDF can modify its internal values (internal state) only. Hence, statements such as INSERT, UPDATE, and DELETE are not allowed inside a UDF unless they are used to modify local TABLE-type variables.

**What Type of UDF Do You Need?**

SQL Server 2005 supports UDFs written in T-SQL code or with the .NET programming languages (called CLR user-defined functions). Both programming environments support two types of UDFs: scalar UDFs and table-valued UDFs.

**Scalar User-Defined Functions**

Scalar UDFs enable you to return a single scalar value. This scalar value can be either a constant value or the result of a complex arithmetic calculation inside the UDF. Scalar UDFs can be written as *inline* scalar UDFs when just a single T-SQL statement is used or as *multistatement* scalar UDFs when multiple T-SQL statements are used inside a *BEGIN-END* block.

Some of the most common uses of this type of UDF are in the column list section of a SELECT statement and in the WHERE clause of a SELECT statement. If used in the column list, the UDF can execute validation logic on a column’s values, calculations, or computations on a column’s values or return calculated values not using a column value as input. If used in the WHERE clause, the UDF can execute validation and filtering logic.

The following code example shows how to define a scalar UDF that calculates the total amount of tax that a certain territory must pay:

CREATE FUNCTION Calculate\_Tax\_For\_Territory

(@TerritoryID INT, @TaxPercent FLOAT)

RETURNS MONEY

AS

BEGIN

DECLARE @Tax MONEY

SELECT @Tax = SUM(Sales.SalesOrderDetail.LineTotal)

FROM Sales.SalesOrderDetail INNER JOIN Sales.SalesOrderHeader ON

Sales.SalesOrderDetail.SalesOrderID = Sales.SalesOrderHeader.SalesOrderID

WHERE (Sales.SalesOrderHeader.TerritoryID = @TerritoryID)

RETURN @Tax \* @TaxPercent

END

You can call this scalar UDF in the SELECT’s column list, as the following example shows; this example calculates the tax for all territories in *AdventureWorks*:

DECLARE @TaxPercentage FLOAT

SET @TaxPercentage = 0.13

SELECT Sales.SalesTerritory.Name,

dbo.Calculate\_Tax\_For\_Territory(Sales.SalesTerritory.TerritoryID, @TaxPercentage) AS Tax

FROM Sales.SalesOrderDetail INNER JOIN Sales.SalesOrderHeader ON

Sales.SalesOrderDetail.SalesOrderID = Sales.SalesOrderHeader.SalesOrderID

INNER JOIN Sales.SalesTerritory ON

Sales.SalesOrderHeader.TerritoryID = Sales.SalesTerritory.TerritoryID

GROUP BY Sales.SalesTerritory.Name, Sales.SalesTerritory.TerritoryID

ORDER BY Tax DESC

Notice how the dbo.Calculate\_Tax\_For\_Territory UDF is called from the SELECT for each of the Territory IDs.

Alternatively, you can call the UDF in the WHERE clause, as does the following code example, which returns the cost to date, the sales to date, and the sales from last year for all territories that pay more than $2 million in taxes:

DECLARE @TaxPercentage FLOAT

SET @TaxPercentage = 0.13

SELECT Name, CostYTD, SalesYTD, SalesLastYear

FROM Sales.SalesTerritory

WHERE

(dbo.Calculate\_Tax\_For\_Territory(TerritoryID, @TaxPercentage) > 2000000)

Notice how the dbo.Calculate\_Tax\_For\_Territory UDF is called from the WHERE for each of the Territory IDs to filter which rows are returned by the SELECT query.

**Table-Valued User-Defined Functions**

You use a table-valued UDF whenever a view or a table is expected. Because table-valued functions can be parameterized, they offer a very powerful replacement for views or stored procedures.

Table-valued UDFs can be written as an inline table-valued UDF when just a single T-SQL statement is needed or as a multistatement table-valued UDF when multiple T-SQL statements are used inside a *BEGIN-END* block. Multistatement table-valued UDFs are called multistatement because they do not directly return the result of executing a single SELECT statement (as inline table-valued UDFs do) but instead return a TABLE-type variable that needs to be defined and filled with data. Database designers must carefully define the format of the TABLE-type variable to be returned by the function in the RETURNS clause when creating the function. Inside the function, data must be inserted into this TABLE-type variable only.

Multistatement table-valued UDFs are used to encapsulate complex queries that can be pre-filtered by input parameters so that they can be called in the FROM clause of the SELECT statement. The following code example defines a multistatement table-valued UDF that calculates the total sales made by all sales employees in *AdventureWorks* and returns the TOP N best sellers:

CREATE FUNCTION BestSellingEmployees(@TerritoryID INT = 0, @Top INT)

RETURNS @Results TABLE

(

[Name] nvarchar(160) NOT NULL,

TotalSales money NOT NULL,

Territory nvarchar(50) NOT NULL

)

AS

BEGIN

IF (@TerritoryID = 0)

BEGIN

INSERT @Results

SELECT TOP(@Top)

Person.Contact.FirstName + ' ' + Person.Contact.LastName +

'(' + Person.Contact.EmailAddress + ')',

SUM(Sales.SalesOrderHeader.SubTotal) AS SalesTotal,

Sales.SalesTerritory.Name AS Territory

FROM Person.Contact

INNER JOIN HumanResources.Employee ON

Person.Contact.ContactID = HumanResources.Employee.ContactID

INNER JOIN Sales.SalesPerson ON

HumanResources.Employee.EmployeeID = Sales.SalesPerson.SalesPersonID

INNER JOIN Sales.SalesOrderHeader ON

Sales.SalesPerson.SalesPersonID = Sales.SalesOrderHeader.SalesPersonID

AND

Sales.SalesPerson.SalesPersonID = Sales.SalesOrderHeader.SalesPersonID

INNER JOIN Sales.SalesTerritory ON

Sales.SalesOrderHeader.TerritoryID = Sales.SalesTerritory.TerritoryID

GROUP BY

Person.Contact.FirstName,

Person.Contact.LastName,

Person.Contact.EmailAddress,

Sales.SalesTerritory.Name

ORDER BY SalesTotal DESC

END

ELSE

BEGIN

INSERT @Results

SELECT TOP(@Top)

Person.Contact.FirstName +''+ Person.Contact.LastName +

'('+ Person.Contact.EmailAddress +')',

SUM(Sales.SalesOrderHeader.SubTotal) AS SalesTotal,

Sales.SalesTerritory.Name AS Territory

FROM Person.Contact

INNER JOIN HumanResources.Employee ON

Person.Contact.ContactID = HumanResources.Employee.ContactID

INNER JOIN Sales.SalesPerson ON

HumanResources.Employee.EmployeeID = Sales.SalesPerson.SalesPersonID

INNER JOIN Sales.SalesOrderHeader ON

Sales.SalesPerson.SalesPersonID = Sales.SalesOrderHeader.SalesPersonID

AND

Sales.SalesPerson.SalesPersonID = Sales.SalesOrderHeader.SalesPersonID

INNER JOIN Sales.SalesTerritory ON

Sales.SalesOrderHeader.TerritoryID = Sales.SalesTerritory.TerritoryID

WHERE Sales.SalesTerritory.TerritoryID = @TerritoryID

GROUP BY

Person.Contact.FirstName, Person.

Contact.LastName,

Person.Contact.EmailAddress,

Sales.SalesTerritory.Name

ORDER BY SalesTotal DESC

END

RETURN

END

Notice that the @TerritoryID parameter declares a default value. This multistatement table-valued UDF can be executed like this:

SELECT \*

FROM dbo.BestSellingEmployees(5, 3)

In this example, the function returns the top three best-selling employees for the Southeast territory. If the caller is interested in overall results that evaluate all employees in the company, the function can be executed like this:

SELECT \*

FROM dbo.BestSellingEmployees(DEFAULT, 3)

By specifying the *DEFAULT* keyword, the @TerritoryID parameter will use its default value of 0.

**CLR User-Defined Functions**

CLR UDFs are written in any .NET programming language (usually Visual Basic .NET or C# .NET). As with CLR stored procedures, the main advantage of this type of function is that all the .NET base class libraries are available for use. For example, CLR functions can be used to access external resources such as files, network resources, Web services, and other databases. As with T-SQL UDFs, CLR UDFs can be created as CLR scalar functions or CLR table-valued functions.

This type of function is usually used to encapsulate reusable procedural code that must run inside the database server. It is not recommended for set-based operations or to manipulate heavy loads of data.

CLR UDFs are outside the scope of this chapter; for more information about them, see the “CLR User-Defined Functions” topic in SQL Server 2005 Books Online at .

**What Type of Data Will the UDF Return?**

UDFs must always return data, either as a scalar value or as a result set.

**Scalar UDFs**

When designing a scalar UDF, database designers should carefully choose the data type returned by the UDF:

* To minimize the need for casting to a different type to client callers.
* So that the data type is big enough to handle all possible results.

A scalar UDF can return any SQL Server 2005 native scalar types (except the *timestamp* data type) or any user-defined type created by using a .NET programming language.

**Table-Valued UDFs**

When designing a table-valued UDF, database designers must choose between creating it as an inline function or as a multistatement function.

For inline functions, you specify the function’s return data type by using a *TABLE* return value, and, as explained previously, only one T-SQL statement is accepted. The following code example shows the *BestSellingEmployee* UDF written as an inline UDF:

CREATE FUNCTION BestSellingEmployees2(@TerritoryID INT, @Top INT)

RETURNS TABLE

AS

RETURN SELECT TOP(@Top)

Person.Contact.FirstName + ' ' + Person.Contact.LastName + ' (' +

Person.Contact.EmailAddress + ')' AS [Name],

SUM(Sales.SalesOrderHeader.SubTotal) AS SalesTotal, Sales.

SalesTerritory.Name AS Territory

FROM Person.Contact INNER JOIN HumanResources.Employee ON

Person.Contact.ContactID = HumanResources.Employee.ContactID

INNER JOIN Sales.SalesPerson ON

HumanResources.Employee.EmployeeID = Sales.SalesPerson.SalesPersonID

INNER JOIN Sales.SalesOrderHeader ON

Sales.SalesPerson.SalesPersonID = Sales.SalesOrderHeader.SalesPersonID AND

Sales.SalesPerson.SalesPersonID = Sales.SalesOrderHeader.SalesPersonID

INNER JOIN Sales.SalesTerritory ON

Sales.SalesOrderHeader.TerritoryID = Sales.SalesTerritory.TerritoryID

WHERE Sales.SalesTerritory.TerritoryID = @TerritoryID

GROUP BY Person.Contact.FirstName, Person.Contact.LastName, Person.Contact.EmailAddress,

Sales.SalesTerritory.Name

ORDER BY SalesTotal DESC

For multistatement functions, you specify the function’s return data type by using a TABLE variable defined as a return value. The structure of this TABLE variable must be defined inside the function declaration. Database designers should carefully define the returned table to:

* Include all expected columns in the result.
* Use a data type big enough to handle all possible results for each of the columns in the result.
* Correctly specify column settings such as DEFAULT values, column constraints (for instance, accept null values, primary keys, unique constraints), computed expressions for computed columns, and so on.

**IMPORTANT Defining a returned table**

Make sure you take into consideration the same factors when defining a returned table from a table-valued UDF as when defining a database table.

**Defining Input Parameters for the UDF**

Unlike stored procedures, UDFs can declare only input parameters. This means that the only way to return more than one scalar value is to use a table-valued function. As noted earlier, parameters can be declared as any SQL Server 2005 native scalar types (except the *timestamp* data type) or as a user-defined type created with any .NET programming language.

In Lesson 2, “Designing Stored Procedures,” several recommendations were presented for defining parameters for stored procedures, including choosing the right number of parameters, names, data types, default values, and direction. The same recommendations apply for UDFs, with some differences in the last two recommendations:

* **Default values** UDF parameters can declare default values (as with stored procedure parameters). The difference is that when calling the function to execute, the *DEFAULT* keyword must be specified to retrieve the default value for the parameter. This behavior is different from using parameters with default values in stored procedures, in which omitting the parameter also implies the default value.
* **Parameter direction** As specified previously, UDFs can have only input parameters.

**Designing Error Handling Routines**

Lesson 2 talked about error handling routines and stored procedures. When designing UDFs, error handling routines are even more important. In contrast to an error in a stored procedure, which does not stop execution, an error in a UDF causes the function to stop executing, which in turn causes the statement that called the function to be canceled.

**Exam Tip**

The new T-SQL TRY . . . CATCH statements in SQL Server 2005 do not apply to UDFs because they are not allowed in that context.

When using UDFs, if there is a computational error or a different type of error that needs to be handled, the calling context needs to handle the error and decide what the proper error handling action is.

**Executing Under the Right Security Context**

Lesson 2 explained security execution context for stored procedures, and the fact that UDFs cannot modify the internal database state does not mean that security context is not important. For example, UDFs can call other functions or stored procedures that do modify the database state. The same considerations for stored procedure security context discussed in Lesson 2 also apply to UDFs.

**Practice: Designing User-Defined Functions**

In this practice, you will design a scalar UDF, using the scenario and tables from this chapter’s previous practices as a basis.

**IMPORTANT Practices build upon each other**

To work successfully with this practice, you need to have finished the practices from Lessons 1 and 2.

**Exercise: Design a Scalar User-Defined Function**

In this exercise, you must create a scalar UDF to validate whether a product should be discontinued. It is created as a UDF and not as a stored procedure so that it can be executed from different constructs and contexts.

The Product table contains a DiscontinuedDate column. This column indicates when the product will no longer be produced. Management wants to determine which products need to be discontinued. You are given the task to implement a UDF to validate whether a product should be discontinued. The required logic is that if the product price is 0, then the product should not be discontinued. Calculate the cost per day (dividing the product cost by the number of days to manufacture); calculate the number of units that need to be sold to reach the cost level (dividing the product sale price by the product cost). All products with a cost per day higher than 500 and with a handicap of having to sell at least 1.5 to make a profit should be marked as discontinued. Design your own scalar UDF to meet these requirements before reading the suggested answer.

**Suggested Answer**

The following T-SQL scalar UDF satisfies the business scenario requirements:

USE [AdventureWorks]

GO

CREATE FUNCTION SHOULD\_SET\_TO\_DISCONTINUE

(

@Cost MONEY,

@Price MONEY,

@DaysToManufacture INT

)

RETURNS BIT

AS

BEGIN

IF (@Price = 0) RETURN 0

DECLARE @CostPerDay MONEY, @UnitsToProfit MONEY

IF (@DaysToManufacture = 0)

SET @CostPerDay = @Cost

ELSE

SET @CostPerDay = (@Cost / @DaysToManufacture)

SET @UnitsToProfit = (@Price / @Cost)

IF (@CostPerDay > 500 AND @UnitsToProfit > 1.5)

RETURN 1

ELSE

RETURN 0

RETURN 0

END

**Quick Check**

1. What type of UDF is required to encapsulate logic to execute as a CHECK constraint definition?
2. List two facts that make UDFs much more agile than stored procedures.
3. What is the main difference between an inline UDF and a multistatement UDF?

**Quick Check Answers**

1. In this context, SQL Server 2005 supports scalar UDFs only. Usually, the returned value from the function is used to validate a CHECK condition to evaluate whether it’s an allowed value.
2. There are many possible reasons UDFs can be more agile than stored procedures; for example, UDFs can be used in different contexts and integrated with the SELECT, INSERT, UPDATE, and DELETE syntax. In addition, table-valued UDFs permit the creation of parameterized result sets.
3. An inline UDF contains a single T-SQL block that must return a value to the caller. In a multistatement UDF, there could be several T-SQL blocks working together to generate a single scalar answer or result set.