CREATE FUNCTION (Transact-SQL)

Creates a user-defined function in SQL Server 2012. A user-defined function is a Transact-SQL or common language runtime (CLR) routine that accepts parameters, performs an action, such as a complex calculation, and returns the result of that action as a value. The return value can either be a scalar (single) value or a table. Use this statement to create a reusable routine that can be used in these ways:

- In Transact-SQL statements such as SELECT
- In applications calling the function
- In the definition of another user-defined function
- To parameterize a view or improve the functionality of an indexed view
- To define a column in a table
- To define a CHECK constraint on a column
- To replace a stored procedure

Transact-SQL Syntax Conventions

Syntax

```
--Transact-SQL Scalar Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN scalar_expression
    END
[ ; ]
--Transact-SQL Inline Table-Valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS TABLE
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    RETURN [ ( ] select_stmt [ ) ]
[ ; ]
--Transact-SQL Multistatement Table-valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [READONLY] }
    [ ,...n ]
  ]
)
RETURNS @return_variable TABLE <table_type_definition>
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN
    END
[ ; ]
--Transact-SQL Function Clauses
<function_option>::=
{
    [ ENCRYPTION ]
  | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}

<table_type_definition>:: =
( { <column_definition> <column_constraint>
  | <computed_column_definition> }
    [ <table_constraint> ] [ ,...n ]
)

<column_definition>::=
{
    { column_name data_type }
    [ [ DEFAULT constant_expression ]
      [ COLLATE collation_name ] | [ ROWGUIDCOL ]
    ]
    | [ IDENTITY [ (seed , increment ) ] ]
    [ <column_constraint> [ ...n ] ]
}

<column_constraint>::=
{
    [ NULL | NOT NULL ]
    { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      [ WITH FILLFACTOR = fillfactor
        | WITH ( < index_option > [ , ...n ] )
      [ ON { filegroup | "default" } ]
  | [ CHECK ( logical_expression ) ] [ ,...n ]
}

<computed_column_definition>::=
column_name AS computed_column_expression

<table_constraint>::=
{
    { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      ( column_name [ ASC | DESC ] [ ,...n ] )
        [ WITH FILLFACTOR = fillfactor
        | WITH ( <index_option> [ , ...n ] )
  | [ CHECK ( logical_expression ) ] [ ,...n ]
}

<index_option>::=
{
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS ={ ON | OFF }
}
--CLR Scalar Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS { return_data_type }
    [ WITH <clr_function_option> [ ,...n ] ]
    [ AS ] EXTERNAL NAME <method_specifier>
[ ; ]
--CLR Table-Valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS TABLE <clr_table_type_definition>
    [ WITH <clr_function_option> [ ,...n ] ]
    [ ORDER ( <order_clause> ) ]
    [ AS ] EXTERNAL NAME <method_specifier>
[ ; ]
--CLR Function Clauses
<order_clause> ::=
{
   <column_name_in_clr_table_type_definition>
   [ ASC | DESC ]
} [ ,...n]

<method_specifier>::=
    assembly_name.class_name.method_name

<clr_function_option>::=
}
    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}

<clr_table_type_definition>::=
( { column_name data_type } [ ,...n ] )
```

Arguments

schema_name

Is the name of the schema to which the user-defined function belongs.

function_name

Is the name of the user-defined function. Function names must comply with the rules for identifiers and must be unique within the database and to its schema.

Note

Parentheses are required after the function name even if a parameter is not specified.

@parameter_name

Is a parameter in the user-defined function. One or more parameters can be declared.

A function can have a maximum of 2,100 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the parameter is defined.

Specify a parameter name by using an at sign (@) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the function; the same parameter names can be used in other functions. Parameters can take the place only of constants; they cannot be used instead of table names, column names, or the names of other database objects.

Note

ANSI_WARNINGS is not honored when you pass parameters in a stored procedure, user-defined function, or when you declare and set variables in a batch statement. For example, if a variable is defined as char(3), and then set to a value larger than three characters, the data is truncated to the defined size and the INSERT or UPDATE statement succeeds.

[type_schema_name.] parameter_data_type

Is the parameter data type, and optionally the schema to which it belongs. For Transact-SQL functions, all data types, including CLR user-defined types and user-defined table types, are allowed except the timestamp data type. For CLR functions, all data types, including CLR user-defined types, are allowed except text, ntext, image, user-defined table types and timestamp data types. The nonscalar types, cursor and table, cannot be specified as a parameter data type in either Transact-SQL or CLR functions.

If type_schema_name is not specified, the Database Engine looks for the scalar_parameter_data_type in the following order:

- The schema that contains the names of SQL Server system data types.
- The default schema of the current user in the current database.
- The dbo schema in the current database.

[=default]

Is a default value for the parameter. If a default value is defined, the function can be executed without specifying a value for that parameter.

Note

Default parameter values can be specified for CLR functions except for the varchar(max) and varbinary(max) data types.

When a parameter of the function has a default value, the keyword DEFAULT must be specified when the function is called to retrieve the default value. This behavior is different from using parameters with default values in stored procedures in which omitting the parameter also implies the default value. However, the DEFAULT keyword is not required when invoking a scalar function by using the EXECUTE statement.

READONLY

Indicates that the parameter cannot be updated or modified within the definition of the function. If the parameter type is a user-defined table type, READONLY should be specified.

return_data_type

Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the timestamp data type. For CLR functions, all data types, including CLR user-defined types, are allowed except the text, ntext, image, and timestamp data types. The nonscalar types, cursor and table, cannot be specified as a return data type in either Transact-SQL or CLR functions.

function_body

Specifies that a series of Transact-SQL statements, which together do not produce a side effect such as modifying a table, define the value of the function. function_body is used only in scalar functions and multistatement table-valued functions.

In scalar functions, function_body is a series of Transact-SQL statements that together evaluate to a scalar value.

In multistatement table-valued functions, function_body is a series of Transact-SQL statements that populate a TABLE return variable.

scalar_expression

Specifies the scalar value that the scalar function returns.

TABLE

Specifies that the return value of the table-valued function is a table. Only constants and @local_variables can be passed to table-valued functions.

In inline table-valued functions, the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables.

In multistatement table-valued functions, @return_variable is a TABLE variable, used to store and accumulate the rows that should be returned as the value of the function. @return_variable can be specified only for Transact-SQL functions and not for CLR functions.

Caution

Joining to a multistatement table valued function in a FROM clause is possible, but can give poor performance. SQL Server is unable to use all the optimized techniques against some statements that can be included in a multistatement function, resulting in a suboptimal query plan. To obtain the best possible performance, whenever possible use joins between base tables instead of functions.

select_stmt

Is the single SELECT statement that defines the return value of an inline table-valued function.

ORDER (<order_clause>)

Specifies the order in which results are being returned from the table-valued function. For more information, see the section, "Guidance on Using Sort Order," later in this topic.

EXTERNAL NAME <method_specifier> assembly_name.class_name.method_name

Specifies the method of an assembly to bind with the function. assembly_name must match an existing assembly in SQL Server in the current database with visibility on. class_name must be a valid SQL Server identifier and must exist as a class in the assembly. If the class has a namespace-qualified name that uses a period (.) to separate namespace parts, the class name must be delimited by using brackets ([ ]) or quotation marks (" "). method_name must be a valid SQL Server identifier and must exist as a static method in the specified class.

Note

By default, SQL Server cannot execute CLR code. You can create, modify, and drop database objects that reference common language runtime modules; however, you cannot execute these references in SQL Server until you enable the clr enabled option. To enable this option, use sp_configure.

Note

This option is not available in a contained database.

<table_type_definition> ({ <column_definition> <column_constraint> | <computed_column_definition> } [<table_constraint>] [,...n])

Defines the table data type for a Transact-SQL function. The table declaration includes column definitions and column or table constraints. The table is always put in the primary filegroup.

< clr_table_type_definition > ({ column_namedata_type } [,...n])

Defines the table data types for a CLR function. The table declaration includes only column names and data types. The table is always put in the primary filegroup.

<function_option>::= and <clr_function_option>::=

Specifies that the function will have one or more of the following options.

ENCRYPTION

Indicates that the Database Engine will convert the original text of the CREATE FUNCTION statement to an obfuscated format. The output of the obfuscation is not directly visible in any catalog views. Users that have no access to system tables or database files cannot retrieve the obfuscated text. However, the text will be available to privileged users that can either access system tables over the DAC port or directly access database files. Also, users that can attach a debugger to the server process can retrieve the original procedure from memory at runtime. For more information about accessing system metadata, see Metadata Visibility Configuration.

Using this option prevents the function from being published as part of SQL Server replication. This option cannot be specified for CLR functions.

SCHEMABINDING

Specifies that the function is bound to the database objects that it references. When SCHEMABINDING is specified, the base objects cannot be modified in a way that would affect the function definition. The function definition itself must first be modified or dropped to remove dependencies on the object that is to be modified.

The binding of the function to the objects it references is removed only when one of the following actions occurs:

- The function is dropped.

- The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.

A function can be schema bound only if the following conditions are true:

- The function is a Transact-SQL function.
- The user-defined functions and views referenced by the function are also schema-bound.
- The objects referenced by the function are referenced using a two-part name.
- The function and the objects it references belong to the same database.
- The user who executed the CREATE FUNCTION statement has REFERENCES permission on the database objects that the function references.

RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

Specifies the OnNULLCall attribute of a scalar-valued function. If not specified, CALLED ON NULL INPUT is implied by default. This means that the function body executes even if NULL is passed as an argument.

If RETURNS NULL ON NULL INPUT is specified in a CLR function, it indicates that SQL Server can return NULL when any of the arguments it receives is NULL, without actually invoking the body of the function. If the method of a CLR function specified in <method_specifier> already has a custom attribute that indicates RETURNS NULL ON NULL INPUT, but the CREATE FUNCTION statement indicates CALLED ON NULL INPUT, the CREATE FUNCTION statement takes precedence. The OnNULLCall attribute cannot be specified for CLR table-valued functions.

EXECUTE AS Clause

Specifies the security context under which the user-defined function is executed. Therefore, you can control which user account SQL Server uses to validate permissions on any database objects that are referenced by the function.

Note
EXECUTE AS cannot be specified for inline user-defined functions.

For more information, see EXECUTE AS Clause (Transact-SQL).

< column_definition >::=

Defines the table data type. The table declaration includes column definitions and constraints. For CLR functions, only column_name and data_type can be specified.

column_name

Is the name of a column in the table. Column names must comply with the rules for identifiers and must be unique in the table. column_name can consist of 1 through 128 characters.

data_type

Specifies the column data type. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except timestamp. For CLR functions, all data types, including CLR user-defined types, are allowed except text, ntext, image, char, varchar, varchar(max), and timestamp.The nonscalar type cursor cannot be specified as a column data type in either Transact-SQL or CLR functions.

DEFAULT constant_expression

Specifies the value provided for the column when a value is not explicitly supplied during an insert. constant_expression is a constant, NULL, or a system function value. DEFAULT definitions can be applied to any column except those that have the IDENTITY property. DEFAULT cannot be specified for CLR table-valued functions.

COLLATE collation_name

Specifies the collation for the column. If not specified, the column is assigned the default collation of the database. Collation name can be either a Windows collation name or a SQL collation name. For a list of and more information about collations, see Windows Collation Name (Transact-SQL) and SQL Server Collation Name (Transact-SQL).

The COLLATE clause can be used to change the collations only of columns of the char, varchar, nchar, and nvarchar data types.

COLLATE cannot be specified for CLR table-valued functions.

ROWGUIDCOL

Indicates that the new column is a row globally unique identifier column. Only one uniqueidentifier column per table can be designated as the ROWGUIDCOL column. The ROWGUIDCOL property can be assigned only to a uniqueidentifier column.

The ROWGUIDCOL property does not enforce uniqueness of the values stored in the column. It also does not automatically generate values for new rows inserted into the table. To generate unique values for each column, use the NEWID function on INSERT statements. A default value can be specified; however, NEWID cannot be specified as the default.

IDENTITY

Indicates that the new column is an identity column. When a new row is added to the table, SQL Server provides a unique, incremental value for the column. Identity columns are typically used together with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to tinyint, smallint, int, bigint, decimal(p,0), or numeric(p,0) columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. You must specify both the seed and increment or neither. If neither is specified, the default is (1,1).

IDENTITY cannot be specified for CLR table-valued functions.

seed

Is the integer value to be assigned to the first row in the table.

increment

Is the integer value to add to the seed value for successive rows in the table.

< column_constraint >::= and < table_constraint>::=

Defines the constraint for a specified column or table. For CLR functions, the only constraint type allowed is NULL. Named constraints are not allowed.

NULL | NOT NULL

Determines whether null values are allowed in the column. NULL is not strictly a constraint but can be specified just like NOT NULL. NOT NULL cannot be specified for CLR table-valued functions.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column through a unique index. In table-valued user-defined functions, the PRIMARY KEY constraint can be created on only one column per table. PRIMARY KEY cannot be specified for CLR table-valued functions.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints. UNIQUE cannot be specified for CLR table-valued functions.

CLUSTERED | NONCLUSTERED

Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints use CLUSTERED, and UNIQUE constraints use NONCLUSTERED.

CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY uses NONCLUSTERED.

CLUSTERED and NONCLUSTERED cannot be specified for CLR table-valued functions.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints cannot be specified for CLR table-valued functions.

logical_expression

Is a logical expression that returns TRUE or FALSE.

<computed_column_definition>::=

Specifies a computed column. For more information about computed columns, see CREATE TABLE (Transact-SQL).

column_name

Is the name of the computed column.

computed_column_expression

Is an expression that defines the value of a computed column.

<index_option>::=

Specifies the index options for the PRIMARY KEY or UNIQUE index. For more information about index options, see CREATE INDEX (Transact-SQL).

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

FILLFACTOR = fillfactor

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or change. fillfactor must be an integer value from 1 to 100. The default is 0.

IGNORE_DUP_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The default is OFF.

STATISTICS_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

Best Practices

If a user-defined function is not created with the SCHEMABINDING clause, changes that are made to underlying objects can affect the definition of the function and produce unexpected results when it is invoked. We recommend that you implement one of the following methods to ensure that the function does not become outdated because of changes to its underlying objects:

- Specify the WITH SCHEMABINDING clause when you are creating the function. This ensures that the objects referenced in the function definition cannot be modified unless the function is also modified.
- Execute the sp_refreshsqlmodule stored procedure after modifying any object that is specified in the definition of the function.

Data Types

If parameters are specified in a CLR function, they should be SQL Server types as defined previously for scalar_parameter_data_type. For information about comparing SQL Server system data types to CLR integration data types or .NET Framework common language runtime data types, see Mapping CLR Parameter Data.

For SQL Server to reference the correct method when it is overloaded in a class, the method indicated in <method_specifier> must have the following characteristics:

- Receive the same number of parameters as specified in [,...n].
- Receive all the parameters by value, not by reference.
- Use parameter types that are compatible with those specified in the SQL Server function.

If the return data type of the CLR function specifies a table type (RETURNS TABLE), the return data type of the method in <method_specifier> should be of type IEnumerator or IEnumerable, and it is assumed that the interface is implemented by the creator of the function. Unlike Transact-SQL functions, CLR functions cannot include PRIMARY KEY, UNIQUE, or CHECK constraints in <table_type_definition>. The data types of columns specified in <table_type_definition> must match the types of the corresponding columns of the result set returned by the method in <method_specifier> at execution time. This type-checking is not performed at the time the function is created.

For more information about how to program CLR functions, see CLR User-Defined Functions.

General Remarks

Scalar-valued functions can be invoked where scalar expressions are used. This includes computed columns and CHECK constraint definitions. Scalar-valued functions can also be executed by using the EXECUTE statement. Scalar-valued functions must be invoked by using at least the two-part name of the function. For more information about multipart names, see Transact-SQL Syntax Conventions (Transact-SQL). Table-valued functions can be invoked where table expressions are allowed in the FROM clause of SELECT, INSERT, UPDATE, or DELETE statements. For more information, see Execute User-defined Functions.

Interoperability

The following statements are valid in a function:

- Assignment statements.
- Control-of-Flow statements except TRY...CATCH statements.
- DECLARE statements defining local data variables and local cursors.
- SELECT statements that contain select lists with expressions that assign values to local variables.
- Cursor operations referencing local cursors that are declared, opened, closed, and deallocated in the function. Only FETCH statements that assign values to local variables using the INTO clause are allowed; FETCH statements that return data to the client are not allowed.
- INSERT, UPDATE, and DELETE statements modifying local table variables.

- EXECUTE statements calling extended stored procedures.
- For more information, see Create User-defined Functions (Database Engine).

Computed Column Interoperability

In SQL Server 2005 and later, functions have the following properties. The values of these properties determine whether functions can be used in computed columns that can be persisted or indexed.

Property	Description	Notes
IsDeterministic	Function is deterministic or nondeterministic.	Local data access is allowed in deterministic functions. For example, functions that always return the same result any time they are called by using a specific set of input values and with the same state of the database would be labeled deterministic.
IsPrecise	Function is precise or imprecise.	Imprecise functions contain operations such as floating point operations.
IsSystemVerified	The precision and determinism properties of the function can be verified by SQL Server.	
SystemDataAccess	Function accesses system data (system catalogs or virtual system tables) in the local instance of SQL Server.	
UserDataAccess	Function accesses user data in the local instance of SQL Server.	Includes user-defined tables and temp tables, but not table variables.

The precision and determinism properties of Transact-SQL functions are determined automatically by SQL Server. The data access and determinism properties of CLR functions can be specified by the user. For more information, see Overview of CLR Integration Custom Attributes.

To display the current values for these properties, use OBJECTPROPERTYEX.

A computed column that invokes a user-defined function can be used in an index when the user-defined function has the following property values:

- IsDeterministic = true
- IsSystemVerified = true (unless the computed column is persisted)
- UserDataAccess = false

- SystemDataAccess = false

For more information, see Indexes on Computed Columns.

Calling Extended Stored Procedures from Functions

The extended stored procedure, when it is called from inside a function, cannot return result sets to the client. Any ODS APIs that return result sets to the client will return FAIL. The extended stored procedure could connect back to an instance of SQL Server; however, it should not try to join the same transaction as the function that invoked the extended stored procedure.

Similar to invocations from a batch or stored procedure, the extended stored procedure will be executed in the context of the Windows security account under which SQL Server is running. The owner of the stored procedure should consider this when giving EXECUTE permission on it to users.

Limitations and Restrictions

User-defined functions cannot be used to perform actions that modify the database state.

User-defined functions cannot contain an OUTPUT INTO clause that has a table as its target.

The following Service Broker statements cannot be included in the definition of a Transact-SQL user-defined function:

- BEGIN DIALOG CONVERSATION
- END CONVERSATION
- GET CONVERSATION GROUP
- MOVE CONVERSATION
- RECEIVE
- SEND

User-defined functions can be nested; that is, one user-defined function can call another. The nesting level is incremented when the called function starts execution, and decremented when the called function finishes execution. User-defined functions can be nested up to 32 levels. Exceeding the maximum levels of nesting causes the whole calling function chain to fail. Any reference to managed code from a Transact-SQL user-defined function counts as one level against the 32-level nesting limit. Methods invoked from within managed code do not count against this limit.

Using Sort Order in CLR Table-valued Functions

When using the ORDER clause in CLR table-valued functions, follow these guidelines:

- You must ensure that results are always ordered in the specified order. If the results are not in the specified order, SQL Server will generate an error message when the query is executed.
- If an ORDER clause is specified, the output of the table-valued function must be sorted according to the collation of the column (explicit or implicit). For example, if the column collation is Chinese (either specified in the DDL for the table-valued function or obtained from the database collation), the returned results must be sorted according to Chinese sorting rules.
- The ORDER clause, if specified, is always verified by SQL Server while returning results, whether or not it is used by the query processor to perform further optimizations. Only use the ORDER clause if you know it is useful to the query processor.
- The SQL Server query processor takes advantage of the ORDER clause automatically in following cases:
    - Insert queries where the ORDER clause is compatible with an index.
    - ORDER BY clauses that are compatible with the ORDER clause.
    - Aggregates, where GROUP BY is compatible with ORDER clause.
    - DISTINCT aggregates where the distinct columns are compatible with the ORDER clause.

The ORDER clause does not guarantee ordered results when a SELECT query is executed, unless ORDER BY is also specified in the query. See sys.function_order_columns (Transact-SQL) for information on how to query for columns included in the sort-order for table-valued functions.

Metadata

The following table lists the system catalog views that you can use to return metadata about user-defined functions.

System View	Description

sys.sql_modules	Displays the definition of Transact-SQL user-defined functions. For example:
	
	Transact-SQL
	`USE AdventureWorks2012;`
	`GO`
	`SELECT definition, type`
	`FROM sys.sql_modules AS m`
	`JOIN sys.objects AS o ON m.object_id =`
	`o.object_id`
	`    AND type IN ('FN', 'IF', 'TF');`
	`GO`
	
	The definition of functions created by using the ENCRYPTION option cannot be viewed by using

	sys.sql_modules; however, other information about the encrypted functions is displayed.
sys.assembly_modules	Displays information about CLR user-defined functions.
sys.parameters	Displays information about the parameters defined in user-defined functions.
sys.sql_expression_dependencies	Displays the underlying objects referenced by a function.

Permissions

Requires CREATE FUNCTION permission in the database and ALTER permission on the schema in which the function is being created. If the function specifies a user-defined type, requires EXECUTE permission on the type.

Examples

A. Using a scalar-valued user-defined function that calculates the ISO week

The following example creates the user-defined function ISOweek. This function takes a date argument and calculates the ISO week number. For this function to calculate correctly, SET DATEFIRST 1 must be invoked before the function is called.

The example also shows using the EXECUTE AS clause to specify the security context in which a stored procedure can be executed. In the example, the option CALLER specifies that the procedure will be executed in the context of the user that calls it. The other options that you can specify are SELF, OWNER, and user_name.

Here is the function call. Notice that DATEFIRST is set to 1.

Transact-SQL

```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'dbo.ISOweek', N'FN') IS NOT NULL
    DROP FUNCTION dbo.ISOweek;
GO
CREATE FUNCTION dbo.ISOweek (@DATE datetime)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
     DECLARE @ISOweek int;
     SET @ISOweek= DATEPART(wk,@DATE)+1
          -DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104');
--Special cases: Jan 1-3 may belong to the previous year
     IF (@ISOweek=0)
          SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1
               AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS
CHAR(2)))+1;
```

```
--Special case: Dec 29-31 may belong to the next year
     IF ((DATEPART(mm,@DATE)=12) AND
          ((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))
          SET @ISOweek=1;
     RETURN(@ISOweek);
END;
GO
SET DATEFIRST 1;
SELECT dbo.ISOweek(CONVERT(DATETIME,'12/26/2004',101)) AS 'ISO Week';
```

Here is the result set.

ISO Week

52

B. Creating an inline table-valued function

The following example returns an inline table-valued function. It returns three columns ProductID, Name and the aggregate of year-to-date totals by store as YTD Total for each product sold to the store.

Transact-SQL

```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'Sales.ufn_SalesByStore', N'IF') IS NOT NULL
    DROP FUNCTION Sales.ufn_SalesByStore;
GO
CREATE FUNCTION Sales.ufn_SalesByStore (@storeid int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'Total'
    FROM Production.Product AS P
    JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
    JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID = 
SD.SalesOrderID
    JOIN Sales.Customer AS C ON SH.CustomerID = C.CustomerID
    WHERE C.StoreID = @storeid
    GROUP BY P.ProductID, P.Name
);
GO
```

To invoke the function, run this query.

Transact-SQL

```
SELECT * FROM Sales.ufn_SalesByStore (602);
```

C. Creating a multi-statement table-valued function

The following example creates the table-valued function fn_FindReports(InEmpID). When supplied with a valid employee ID, the function returns a table that corresponds to all the employees that report to the employee either directly or indirectly. The function uses a recursive common table expression (CTE) to produce the hierarchical list of employees. For more information about recursive CTEs, see WITH common_table_expression (Transact-SQL).

Transact-SQL

```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'dbo.ufn_FindReports', N'TF') IS NOT NULL
    DROP FUNCTION dbo.ufn_FindReports;
GO
CREATE FUNCTION dbo.ufn_FindReports (@InEmpID INTEGER)
RETURNS @retFindReports TABLE
(
    EmployeeID int primary key NOT NULL,
    FirstName nvarchar(255) NOT NULL,
    LastName nvarchar(255) NOT NULL,
    JobTitle nvarchar(50) NOT NULL,
    RecursionLevel int NOT NULL
)
--Returns a result set that lists all the employees who report to the
--specific employee directly or indirectly.*/
AS
BEGIN
WITH EMP_cte(EmployeeID, OrganizationNode, FirstName, LastName, JobTitle,
RecursionLevel) -- CTE name and columns
    AS (
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
p.LastName, e.JobTitle, 0 -- Get the initial list of Employees for
Manager n
        FROM HumanResources.Employee e
                       INNER JOIN Person.Person p
                       ON p.BusinessEntityID = e.BusinessEntityID
        WHERE e.BusinessEntityID = @InEmpID
        UNION ALL
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
p.LastName, e.JobTitle, RecursionLevel + 1 -- Join recursive member to
anchor
        FROM HumanResources.Employee e
            INNER JOIN EMP_cte
            ON e.OrganizationNode.GetAncestor(1) =
EMP_cte.OrganizationNode
                       INNER JOIN Person.Person p
                       ON p.BusinessEntityID = e.BusinessEntityID
        )
-- copy the required columns to the result of the function
   INSERT @retFindReports
   SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
   FROM EMP_cte
   RETURN
END;
GO
-- Example invocation
```

```
SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
FROM dbo.ufn_FindReports(1);

GO
```

D. Creating a CLR function

The example creates CLR function len_s. Before the function is created, the assembly SurrogateStringFunction.dll is registered in the local database.

```
DECLARE @SamplesPath nvarchar(1024);
-- You may have to modify the value of this variable if you have
-- installed the sample in a location other than the default location.
SELECT @SamplesPath = REPLACE(physical_name, 'Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\master.mdf', 'Microsoft SQL
Server\100\Samples\Engine\Programmability\CLR\')
    FROM master.sys.database_files
    WHERE name = 'master';

CREATE ASSEMBLY [SurrogateStringFunction]
FROM @SamplesPath +
'StringManipulate\CS\StringManipulate\bin\debug\SurrogateStringFunction.d
ll'
WITH PERMISSION_SET = EXTERNAL_ACCESS;
GO

CREATE FUNCTION [dbo].[len_s] (@str nvarchar(4000))
RETURNS bigint
AS EXTERNAL NAME
[SurrogateStringFunction].[Microsoft.Samples.SqlServer.SurrogateStringFun
ction].[LenS];
GO
```