Function

NGUYEN Hong Phuong

Email: phuongnh@soict.hust.edu.vn

Site: https://users.soict.hust.edu.vn/phuongnh

Outline

- Scalar functions in SQL Server
- Syntax
- Modifying a scalar function
- Table variables
- Table functions

Scalar Functions in SQL Server

- Takes as input one or more parameters, returns a single value
- Helps simplify the programmer's code. For example, if a Select query has a complex calculation, you can use a scalar function that encapsulates this formula, and use it in each query.

Syntax to create a function

```
CREATE FUNCTION [schema_name.]function_name
([@parameter [AS] [type_schema_name.] datatype
[= default] [READONLY]
)
RETURNS return_datatype
[WITH { ENCRYPTION
|SCHEMABINDING
|RETURNS NULL ON NULL INPUT
|CALLED ON NULL INPUT
|EXECUTE AS Clause]
[AS]
BEGIN
[declaration_section]
executable_section
RETURN return_value
END;
```

Description

- schema_name: The name of the schema (schema) that owns the function.
- function_name: The name assigned to the function.
- @parameter: One or more parameters passed to the function.
- type_schema_name: The data type of the schema (if any).
- Datatype: The data type for @parameters.
- Default: Default value assigned to @parameter.
- READONLY: @parameters cannot be overridden by functions.
- return_datatype: The data type of the return value.

Description (cont'd)

- ENCRYPTION: The function's source code will not be stored as text in the system.
- SCHEMABINDING: Ensures objects are not modified to affect the function.
- RETURNS NULL ON NULL INPUT: The function will return NULL if any parameter is NULL.
- CALL ON NULL INPUT: The function will execute even if the parameter is NULL.
- EXECUTE AS clause: Defines the security context to execute the function.
- return_value: The value to be returned.

Example 1

```
CREATE FUNCTION fStaff
(@staff_id INT)
RETURNS VARCHAR(50)
AS
BEGIN
	DECLARE @staff_name VARCHAR(50);
	IF @staff_id < 10
	SET @staff_name = 'Smith';
	ELSE
	SET @staff_name = 'Lawrence';
	RETURN @staff_name;
END;
```

```
SELECT dbo.fStaff(8);
```

Example 2: BikeStores database

- BikeStores

```
CREATE FUNCTION sales.fNetSale
(
    @quantity INT,
    @list_price DEC(10,2),
    @discount DEC(4,2)
)
RETURNS DEC(10,2)
AS
BEGIN
    RETURN @quantity * @list_price * (1 - @discount);
END;
```

```
SELECT sales.fNetSale(10,100,0.1) net_sale;
```

Example 2: BikeStores database (cont'd)

```sql
SELECT order_id,
    SUM(sales.fNetSale(quantity, list_price, discount)) net_amount
FROM sales.order_items
GROUP BY order_id
ORDER BY net_amount DESC;
```

Modifying a scalar function

```
ALTER FUNCTION [schema_name.]function_name
(
    parameter_list
)
RETURN data_type AS
BEGIN
    statements
    RETURN value
END
```

Drop a scalar function

- `DROP FUNCTION [schema_name.]function_name;`

- `DROP FUNCTION sales.fNetSale;`

Some key points about scalar function

- Can be used almost anywhere in T-SQL statements.
- Accepts one or more parameters but only returns a single value, so one RETURN statement is required.
- Can use logic like IF block or WHILE loop
- Unable to UPDATE data. Data should not be accessed
- Can call another function

Table Variables in SQL Server

- Table variables allow data records to be stored, similar to the temporary tables
- Declare a table variable:

```
DECLARE @table_variable_name TABLE
(
    column_list
);
```

- Table variable scope
  - No longer exists after the end of the command block
  - If you define a table variable in an SP or Function, it won't exist after the SP or Function ends

Table Variables in SQL Server (cont'd)

- Example

```
DECLARE @product_table TABLE
(
    product_name VARCHAR(MAX) NOT NULL,
    brand_id INT NOT NULL,
    list_price DEC(11,2) NOT NULL
);
```

Insert data into a table variable

```
DECLARE @product_table TABLE (
    product_name VARCHAR(MAX) NOT NULL,
    brand_id INT NOT NULL,
    list_price DEC(11,2) NOT NULL
);

INSERT INTO @product_table
SELECT product_name, brand_id, list_price
FROM production.products
WHERE category_id = 1;

SELECT * FROM @product_table;
```

Limitations of table variables

- The structure of a table variable must be defined, and cannot be changed after it has been declared, not like a regular table or a temporary table.
- Table variables don't contain statistics, so it doesn't help the query optimizer to come up with a good query execution plan. Table variables should only be used to store few records.

Limitations of table variables (cont'd)

- Do not use table variables as input and output parameters. However, it is possible to return a table variable from function
- A non-clustered index cannot be created for a table variable. From SQL Server 2014 it is possible to add a non-clustered index as part of a table variable declaration

Limitations of table variables (cont'd)

- If you are using a table variable with JOIN, you need to alias the table
- For example:

```sql
SELECT brand_name, product_name, list_price
FROM production.brands b INNER JOIN
@product_table pt ON b.brand_id = pt.brand_id;
```

Performance of Table Variables in SQL Server

- Using a table variable in SP will recompile less than using the temporary table
- Uses less resources than a temporary table with less locking and logging overhead.
- Table variables execute in the tempdb database, not in memory (similar to the temporary table)

Using table variables in user-defined function

```
CREATE FUNCTION udfSplit
(
    @string VARCHAR(MAX),
    @delimiter VARCHAR(50) = ' ')
RETURNS @parts TABLE
(
idx INT IDENTITY PRIMARY KEY,
val VARCHAR(MAX)
)
AS
BEGIN
DECLARE @index INT = -1;
WHILE (LEN(@string) > 0)
BEGIN
    SET @index = CHARINDEX(@delimiter , @string);
    IF (@index = 0) AND (LEN(@string) > 0)
```

Using table variables in user-defined function

```
            BEGIN
            INSERT INTO @parts
            VALUES (@string);
            BREAK
            END
        IF (@index > 1)
        BEGIN
            INSERT INTO @parts
            VALUES (LEFT(@string, @index - 1));
            SET @string = RIGHT(@string, (LEN(@string) - @index));
        END
        ELSE
        SET @string = RIGHT(@string, (LEN(@string) - @index));
        END
    RETURN
    END
                SELECT * FROM udfSplit('foo,bar,baz',',');
```

Table function in SQL Server

- A table function is a user-defined function that returns a table data type. The return type of the table function is a table, so the table function can be used in the same way as the table.

Create and execute the table functions

```
CREATE FUNCTION udfProductInYear
(
    @model_year INT
)
RETURNS TABLE
AS
RETURN
    SELECT
        product_name,
        model_year,
        list_price
    FROM
        production.products
    WHERE
        model_year = @model_year;
```

```
SELECT *
FROM udfProductInYear(2017);

SELECT product_name, list_price
FROM udfProductInYear(2018);
```

Modifying the table functions

```
ALTER FUNCTION udfProductInYear (
    @start_year INT,
    @end_year INT
)
RETURNS TABLE
AS
RETURN
    SELECT
        product_name,
        model_year,
        list_price
    FROM
        production.products
    WHERE
        model_year BETWEEN @start_year AND @end_year
```

```
SELECT
    product_name,
    model_year,
    list_price
FROM
    udfProductInYear(2017,2018)
ORDER BY
    product_name;
```

Multi-statement table function

- A function that has many statements and returns a table value
- Useful, because it is possible to execute multiple queries within the function and aggregate the results into the returned table
- To define a table function, use a table variable as the return value. Inside the function, execute one/more insert queries into this table variable

Multi-statement table function

- Example

```
CREATE FUNCTION udfContacts()
    RETURNS @contacts TABLE (
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        email VARCHAR(255),
        phone VARCHAR(25),
        contact_type VARCHAR(20)
    )
AS
BEGIN
    INSERT INTO @contacts
    SELECT first_name, last_name,
        email, phone, 'Staff'
    FROM sales.staffs;
```

```
    INSERT INTO @contacts
    SELECT
        first_name,
        last_name,
        email,
        phone,
       'Customer'
    FROM sales.customers;
    RETURN;
END;


SELECT *
FROM udfContacts();
```

When to use table functions?

- Use table function as a view with parameters (dynamic view)
- Compared to stored procedures, table functions are more flexible, because table functions can be used anywhere where tables are used.

Drop function

- Syntax

```
DROP FUNCTION [IF EXISTS] [schema_name.] function_name;
```

- Note, this delete statement will fail:
  - If the function is referenced in a view or another function is created with the WITH SCHEMABINDING option
  - If there are constraints CHECK, DEFAULT and computed columns related to function

Drop function (cont'd)

- Remove multiple functions

```
DROP FUNCTION [IF EXISTS]
    schema_name.function_name1,
    schema_name.function_name2,
    ...;
```

Example

```sql
CREATE FUNCTION sales.udf_get_discount_amount
(
    @quantity INT,
    @list_price DEC(10,2),
    @discount DEC(4,2)
)
RETURNS DEC(10,2)
AS
BEGIN
    RETURN @quantity * @list_price * @discount
END

DROP FUNCTION sales.udf_get_discount_amount;
```

- Delete function with WITH SCHEMABINDING

```
CREATE FUNCTION sales.udf_get_discount_amount
(
    @quantity INT,
    @list_price DEC(10,2),
    @discount DEC(4,2)
)
RETURNS DEC(10,2)
WITH SCHEMABINDING
AS
BEGIN
    RETURN @quantity * @list_price * @discount
END
```

```sql
CREATE VIEW sales.discounts
WITH SCHEMABINDING
AS
SELECT
    order_id,
    SUM(sales.udf_get_discount_amount(
        quantity,
        list_price,
        discount
    )) AS discount_amount
FROM
    sales.order_items i
GROUP BY
    order_id;
```

- Try deleting, there will be an error

```sql
DROP FUNCTION sales.udf_get_discount_amount;
```

- To delete, you must delete the view first

```
DROP VIEW sales.discounts;

DROP FUNCTION sales.udf_get_discount_amount;
```