DBMaker

SQL Function User's Guide

Version: 01.00

Document No: 54/DBM547-T06262025-01-SQLF
Author: Production Team
Syscom Computer Engineering CO.

Print Date: June 26, 2025

凌羣電腦
THE SYSCOM GROUP

Table of Content

1. SQL Function Introduction

Welcome to SQL Function User's Guide, this document will introduce users about SQL Functions. Include create/delete SQL functions, execute SQL functions.

SQL Function is a type of user-defined function (a subroutine that can return a single value) can be executed regularly in the database. SQL function provided by DBMaker is based on the syntax of stored procedure, but different from stored procedure, SQL functions cannot do DML/DDL operations except SELECT.

SQL Functions are similar to operators in that they operate on data items and return results. Functions differ from operators in the format of their arguments. This format enables them to operate on zero, one, two or more arguments

➲Example:

```
function(arg1, arg2, ...) return ...
```

SQL Function is built into the DBMaker database and can be used in various appropriate SQL statements. DBMaker SQL function is one of the DBMaker user-defined function, besides SQL function, there are C function and Lua function. The followings are the definition of different functions:

Built-in Function: A function already been built and can be used when database is created.

➲Example: CURRENT_DATE is a built-in function. Users can use it without creation.

```
dmSQL> select CURRENT_DATE();

CURRENT_DATE()
==============
2025-02-14
```

User-Defined Function: A function is designed and created by user. Include C function, Lua function and SQL function. If users want to execute these functions, they have to create it manually, or run the SQL script located under the same directory.

NOTE: There are two functions AES_DECRYPT and AES_ENCRYPT stored in database but haven't been created. If they are created, they will be defined as a user-defined function, please refer to ***SQL Command and Function Reference Chapter 4. User-Defined Function*** for more information.

SQL Functions: A type of user-defined function, users can use the CREATE FUNCTION syntax to design function body and create their own SQL functions. Please refer to the following chapters for SQL function information.

2. Stored Procedure and Function

SQL Function and SQL Procedure are both SQL programmable routine. They both support the same execution logic and syntax usage. This chapter will simply explain the difference between SQL stored procedure and SQL function.

	SQL Stored Procedure	SQL Function
Purpose	Perform actions or operations	Perform computations and return values
Return Value	Doesn't return a value directly. May return output parameters.	Always returns a value or table.
Parameter	Accept input and output parameters	Accept input parameters
Modify data	Can modify data: Insert, Update, Delete can be used in procedure	Cannot modify data
Transaction	Can manage transactions	Cannot manage transactions
Call Method	Executed with CALL command.	Often used in SQL queries

3. SQL Function Syntax

This chapter will introduce users how to create/check/delete SQL functions.

3.1. CREATE FUNCTION SYNTAX

To use the CREATE FUNCTION statement, the user must have system authority DBA, SYSDBA or SYSADM.

The syntax diagram for creating SQL Function is as follows:

```
<[CREATE FUNCTION]>
    create-function-statement ::= create-function-file | create-function-
script

create-function-file ::= CREATE FUNCTION FROM source-file-path

create-function-script ::= {block delimiter} create_function {block
delimiter}

create_function ::=
CREATE [ BUILTIN | INTERNAL ] FUNCTION function_name [ ( parameter_def [,
parameter_def]... ) ]
  RETURN datatype
  LANGUAGE SQL
  [ defined_clause ]
  { IS | AS }
  function_body
```

The sections in the diagram:

[BUILTIN | INTERNAL]:

Native UDF built-in function tags

function_name:

Since there is no concept of module for SQL Function, there is no need to use the module name of user-defined-function. If the syntax of function name AAA.BBB is encountered, it will be handled according to the error. Please note that the name of the Function needs to be consistent with the name of the SQL Procedure.

Ensure that function_name is unique, if there is a naming conflict with the current SQL stored procedure and user-defined-function, please rename it.

If encounter the problem of duplicate names of any objects during the creation process, users will be handled in accordance with an error.

parameter_def:

```
parameter_def :: = parameter_name parameter_type
```

Used to describe the input parameter type of the SQL function. The parameters are consistent with user-defined function. Also supports the function-specific types STRING, NSTRING and VARTYPE. The output data is processed directly by "RETURN" syntax.

Therefore, the input and output parameter definitions of SQL Function are stored in the SYSUSERFUNC system table and SYSPROCPARAM system table respectively.

parameter_name :
The name of the parameter, this name will be a variable and can be used in the function body. Supporting SQL column data, user constants, operation expressions, recursive calls, etc.

Input parameters support dynamic input. If the input parameters are less than the defined parameters, the empty parameters will be treated as NULL values to make up for the shortcomings of the input parameters.

Due to Function limitations, SQL Function supports up to 8 input parameters.

parameter_type:
Specifies the type of input parameters. Support common data types Function-specific data types: VARTYPE, STRING, NSTRING.

Supported types: SMALLINT, INTEGER, INT, BIGINT, FLOAT, DOUBLE, REAL, TIMESTAMP, DATE, TIME, DECIMAL, DECIMAL(num), DECIMAL(num,num), BINARY(num), CHAR(num), VARCHAR(num), NCHAR(num), NVARCHAR(num), BLOB, CLOB, VARTYPE, STRING, NSTRING

RETURN DATATYPE:

Define the datatype of the return value. Please notice:

- VARTYPE cannot be defined as a RETURN DATATYPE.
- If the RETURN DATATYPE is BLOB/CLOB, system will create a temporary BLOB.

LANGUAGE SQL:

LANGUAGE SQL is a syntax definition supported by CREATE FUNCTION. Indicates that the current Function language is SQL. LANGUAGE SQL must exist in the CREATE FUNCTION statement.

defined_clause:

Custom clause reserved fields

IS/AS:

The IS/AS token serves as a guide to the function body of the SQL Function.

function_body:

Function body of SQL Function. The function body is a compound statement. Compound statements are bounded by the keywords BEGIN and END. The following is the architecture of SQL function statement:

```
function_body ::=
BEGIN
 DECLARE ...
 |EXCEPTION HANDLER ...
 |SQL_STATEMENT ...
END;
```

3.1.1. CREATE SQL FUNCTION

SQL Function can be created in two ways:

1. **Create from local files in client-side.**
2. **Create by scripts**

Users can choose the right method.

The following is the syntax of CREATE FUNCTION command:

```
<[CREATE FUNCTION]>
    create-function-statement ::= create-function-file | create-function-
script

create-function-file ::= CREATE FUNCTION FROM source-file-path

create-function-script ::= {block delimiter} create_function {block
delimiter}
```

This chapter will only show the creation of the function. The detail of the function body will be introduced in the following chapters.

3.1.1.1. Create from file

Creating from a file requires ensuring that the file path is stored on the client side, not the server side. The content of the SQL function is stored in the file in the form of text. For the creation syntax in the text, please refer to the subsequent explanation of the creation syntax.

To create a SQL Function in the database, or to create a SQL Function for other users, the user must have system authority DBA, SYSBDA or SYSADM.

The following naming rules apply to the parameter names and variable names in SQL

stored procedures and SQL functions:

- SQL Function names can contain at most 128 characters
- SQL Function names can contain any alphanumeric characters, including the underscore
- Character may be in any position
- SQL Function names are not case-sensitive
- The names of Functions and Procedures must be unique.

➲Example:
The following shows how to create a function by prepared file a.sql.

```
create function from 'a.sql';
```

Create a file following ascii encoding and write the syntax and logic of creating the SQL function into the file.

```
CREATE FUNTION FUNC1(VAL string)
RETURN int
LANGUAGE SQL
AS
BEGIN
  DECLARE CNUM INT;
  SET CNUM = 100;
  RETURN CNUM;
END;
```

3.1.1.2. Create by script

Create SQL Function by script sends the creation file directly from the client to the server. Create by script requires using the block delimiter, please customize the appropriate delimiter through "SET BLOCK DELIMITER". Such as *SET BLOCK DELIMITER @@;*

The syntax of script creation is consistent with the text of <(create-function-file)>. For the writing method, please refer to the subsequent explanation of the creation syntax.

➲Example:

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL>
@@
CREATE FUNTION FUNC1(VAL string) RETURN int
LANGUAGE SQL
AS
BEGIN
  DECLARE CNUM INT;
  RETURN CNUM;
END;
@@
```

3.1.2. PASS INPUT PARAMETERS

This chapter will teach users how to create a function with passing input parameters. The following is the syntax to define input parameters when creating functions:

parameter_def:

```
parameter_def :: = parameter_name parameter_type
```

Used to describe the input parameter type of the SQL function. The parameters are consistent with user-defined function. Also supports the function-specific types STRING, NSTRING and VARTYPE. The output data is processed directly by "RETURN" syntax.

Therefore, the input and output parameter definitions of SQL Function are stored in the SYSUSERFUNC system table and SYSPROCPARAM system table respectively.

parameter_name :

The name of the parameter, this name will be a variable and can be used in the function body. Supporting SQL column data, user constants, operation expressions, recursive calls, etc.

Input parameters support dynamic input. If the input parameters are less than the defined parameters, the empty parameters will be treated as NULL values to make up for the shortcomings of the input parameters.

Due to Function limitations, SQL Function supports up to 8 input parameters.

parameter_type:

Specifies the type of input parameters. Support common data types and Function-specific data types: VARTYPE, STRING, NSTRING.

Supported types: SMALLINT, INTEGER, INT, BIGINT, FLOAT, DOUBLE, REAL, TIMESTAMP, DATE, TIME, DECIMAL, DECIMAL(num), DECIMAL(num,num), BINARY(num), CHAR(num), VARCHAR(num), NCHAR(num), NVARCHAR(num), BLOB, CLOB, VARTYPE, STRING, NSTRING

➲Example 1:

The following example will combine two input parameters and return.

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL> @@
CREATE function FUNC3(N1 varchar(128), N2 varchar(128))
RETURN string
LANGUAGE SQL
AS
BEGIN
 IF N1 IS NULL THEN
   SET N1 = '';
 END IF;
 IF N2 IS NULL THEN
   SET N2 = '';
 END IF;
  return N1||N2;
END;
@@
dmSQL> SELECT FUNC3('B','B');

                                FUNC3('B','B')
======================================================================
BB
1 rows selected
```

➲Example 2:

The following function will return the input parameter CC1 and CC2.

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL> @@
CREATE FUNCTION PARAM1(CC1 VARCHAR(128),CC2 VARCHAR(128))
RETURN VARCHAR(256)
LANGUAGE SQL
AS
BEGIN
  RETURN CC1||CC2;
END;
@@
dmSQL> SELECT PARAM1('B','B');

                              PARAM1('B','B')
======================================================================
BB
1 rows selected
```

➲Example 3:

The following function shows function parameter types need to be consistent. If incorrect should report error.

```
CREATE FUNCTION PARAM2(C1 INT,C2 VARCHAR(128))
RETURN VARCHAR(128)
LANGUAGE SQL
AS
BEGIN
 RETURN C1; /* RETURN C1; TYPE INCORRECT */
END;
dmSQL> select PARAM2(1); ->ERROR (6150): [DBMaker] the insert/update value
type is incompatible with column data type ...
```

➲Example 4:

The following example shows function parameters support dynamic input. If the input

parameters are less than the defined parameters, the empty parameters will be treated as NULL values to make up for the shortcomings of the input parameters.

```
CREATE FUNCTION PARAM3(C1 INT,C2 VARCHAR(128))
RETURN VARCHAR(128)
LANGUAGE SQL
AS
BEGIN
 RETURN C2;
END;

dmSQL> select PARAM3(1);

                                PARAM3(1)
================================================================================
                                   NULL

dmSQL> select PARAM3(1,'abc');

                             PARAM3(1,'ABC')
================================================================================
abc
```

➲Example 5:

Function parameter support as result of other calls.

```
CREATE FUNCTION PARAM4(C1 INT,C2 INT,C3 INT)
RETURN INT
LANGUAGE SQL
AS
BEGIN
  RETURN C1+C2+C3;
END;

dmSQL> SELECT PARAM4( PARAM4(1,3,5) , 2 , 4);

PARAM4( PARAM4(1,3,5) , 2 , 4)
==============================
                            15
```

Among them, "parameter_type" also supports three Function-specific parameter types: STRING, NSTRING, VARTYPE. Please also be careful when using these types.

3.1.2.1. string

The string type can handle the input of any character parameter, including char/varchar, etc. It is essentially a variable-length string.

➲Example:

The following function shows function created with string.

```
CREATE FUNCTION t_string(C1 string,C2 string)
RETURN string
LANGUAGE SQL
AS
BEGIN
 RETURN C1|C2;
END;
```

3.1.2.2. nstring

The unicode string type can handle the input of any unicode character parameter, including nchar/nvarchar, etc., which is essentially a variable-length unicode string.

➲Example:

```
CREATE FUNCTION t_nstring(C1 nstring,C2 nstring)
RETURN nstring
LANGUAGE SQL
AS
BEGIN
 RETURN C1|C2;
END;
```

3.1.2.3. vartype

Function can adapt to the input parameter type, and can determine the stored variable form based on the input parameters. But the data still needs to be processed. Please note that vartype cannot be used in "RETURN" types.

➲Example:

```
CREATE FUNCTION t_vartype(C1 vartype,C2 vartype)
RETURN string
LANGUAGE SQL
AS
BEGIN
 RETURN cast(C1 as varchar(128))|cast(C2 as varchar(128));
END;
```

3.1.3. RETURN OUTPUT VALUES

SQL Function will immediately return the current operation result when using RETURN object.

When SQL Function encounters RETURN syntax, it will immediately jump out of the current execution logic and send the value of the object that needs to be returned. According to the definition of user-define-function, UDF returns and only returns one object. The returned object supports the definition of variables, parameters, constants and expressions.

Please note that if the data type of the parameter used when calling SQL Function is not the data type expected by SQL Function, an error will be reported.

➲Example 1:

The following function shows the return object can be an input parameter.

```
CREATE FUNCTION FUNCRET1(C1 INT)
RETURN INT
LANGUAGE SQL
AS
BEGIN
  SET C1=C1+1;
  RETURN C1;
END;

dmSQL> select funcret1(3);

FUNCRET1(3)
===========
          4
```

➲Example 2:

The following function shows RETURN INT doesn't allow returning string, only return int.

```
CREATE FUNCTION FUNCRET2(C1 INT)
RETURN INT
LANGUAGE SQL
AS
BEGIN
```

```
  IF C1=1 THEN
    RETURN 100;
  ELSEIF C1=2 THEN
    RETURN -1;
  ELSE
    RETURN 'dddd';
  END IF;
END;

dmSQL> select funcret2(1);

FUNCRET2(1)
===========
        100

dmSQL> select funcret2(2);

FUNCRET2(2)
===========
         -1

dmSQL> select funcret2(3);

FUNCRET2(3)
===========
ERROR (6150): [DBMaker] the insert/update value type is incompatible with
column data type or compare/operand value is incompatible with column data
type in expression/predicate
```

➲Example 3:

The following function shows the return object can be a defined variable.

```
CREATE FUNCTION FUNCRET3(C1 INT)
RETURN INT
LANGUAGE SQL
AS
BEGIN
  DECLARE C_OUT INT;
  Set C_OUT = C1+200;
  RETURN C_OUT;
END;

dmSQL> select funcret3(1);

FUNCRET3(1)
===========
        201
```

➲Example 4:

The following function shows the return object can be an operation expression.

```
CREATE FUNCTION FUNCRET4(C1 INT)
RETURN INT
LANGUAGE SQL
AS
BEGIN
  DECLARE C_OUT INT;
  Set C_OUT = C1+200;
  RETURN C1 + C_OUT + 1;
END;

dmSQL> select funcret4(1);
```

```
FUNCRET4(1)
===========
        203
```

➲Example 5:

The following function shows the return object can be a complex arithmetic expressions.

```
CREATE FUNCTION FUNCRET5(C1 INT)
RETURN INT
LANGUAGE SQL
AS
BEGIN
  DECLARE C_OUT INT;
  Set C_OUT = C1 + 1000;
  RETURN C1 + C_OUT + ABS(-100) +  FUNCRET3(10000);
END;

dmSQL> select funcret5(1);

FUNCRET5(1)
===========
      11302
```

Usually users will complete data logic processing in SQL Function, the results that need to be returned have been returned in the following way. The returned data will be directly returned to the execution result of the function.

➲Example 6:

The following example shows the returned of the second result will be ignored because the first result has returned and exit the function.

```
CREATE FUNCTION func1(C1 INT)
RETURN INT
LANGUAGE SQL
AS
BEGIN
  DECLARE C_OUT INT;
  Set C_OUT = C1+100;
  RETURN C_OUT;      /* return and exit function */
  Set C_OUT = C1+200;
  RETURN C_OUT;      /* not attach */
END;

dmSQL> select func1(100);

FUNC1(100)
===========
        200

dmSQL> select func1(300);

FUNC1(300)
===========
        400
```

Usually, multiple RETURN commands can exist within one SQL Function. Example 6 will appear because the previous RETURN will jump out of the logic and return the result. So the later RETURN command will not be called. Please refer to the actual logic situation for your SQL function.

➲Example 7:
If there is no RETURN data at the end of execution or is no RETURN on the logical branch and exits normally. By default, a NULL value of the specified RETURN type will be returned.

```
create function func3
RETURN int
LANGUAGE SQL
AS
BEGIN
  DECLARE C_OUT INT;
END;

dmSQL> select func3();

  FUNC3()
===========
   NULL
```

3.1.4. FUNCTION BODY

The execution block of SQL Function is determined by the syntax mark "AS"/"IS", usually it starts from "BEGIN" and end by "END". All execution syntax is supported within the BEGIN-END block, basic parameter, variable operations, expression, flow operations, condition operations, also includes SELECT/cursors/exception handling and so on. But different from stored procedure, DML/DDL commands and transaction are not supported.

The following is the architecture of SQL Function's function body statement:

```
BEGIN
 DECLARE ...
 |EXCEPTION HANDLER ...
 |SQL_STATEMENT ...
END;
```

Through grammatical flow and conditional judgment statements such as CASE/IF/GOTO etc, user can define execution logic.

➲Example 1:
The following function shows using process statements or conditional statements to define the logic procedure.

```
CREATE function func2(C1 int,C2 varchar(128),C3 varchar(128))
RETURN string
LANGUAGE SQL
IS
BEGIN
  IF C1 IS NULL THEN
    RETURN 'NULL EXCEPTION';
  ELSEIF C1 > 0 THEN
    RETURN C2;
  ELSE
    RETURN C3;
  END IF;
END;

dmSQL> select func2(NULL,'31','33');

                              FUNC2(NULL,'31','33')
================================================================================
NULL EXCEPTION

dmSQL> select func2(10,'31','33');
```

```
                                FUNC2(10,'31','33')
========================================================================
31

dmSQL> select func2(-1,'31','33');

                                FUNC2(-1,'31','33')
========================================================================
33
```

➲Example 2:
The following function shows users can use exception and cursor operations within SQL Function

```
CREATE function FUNCS6
RETURN int
LANGUAGE SQL
AS
BEGIN
    DECLARE val INT;
    DECLARE sum INT;
    DECLARE CONTINUE HANDLER FOR NOT FOUND;
    DECLARE cur CURSOR FOR SELECT v1 FROM t1;

    SET sum = 0;
    OPEN cur;

    WHILE SQLCODE = 0 DO
        SET sum = sum + val;
        FETCH cur INTO val;
    END WHILE;

    CLOSE cur;
    RETURN sum;
END;
```

3.2. EXECUTE SQL FUNCTION

To execute SQL Function, if users encounter a user with insufficient permissions, users may need additional execution permissions, such as "EXECUTE" permissions that require "GRANT" to create objects. If the function selects data from table, users may also need the "RESOURCE" or higher privilege of the table.

This chapter will teach users how to execute SQL Function. The above examples have showed how to use the SELECT statement to get return results. Furthermore, SQL Function can be embedded into SELECT/INSERT/UPDATE and other statements, so when it is embedded into other statements, the corresponding execution results can be guaranteed.

➲Example 1:
The following examples show SQL Function can be embedded in DML and DDL statement. The following function will be used in the examples.

```
create function FUNCS(C1 int)
RETURN int
LANGUAGE SQL
IS
BEGIN
  return C1+100;
END;
```

Embedded in SELECT statement.

```
dmSQL> select FUNCS(c1) from src1;

 FUNCS(C1)
===========
         101

1 rows selected
```

Embedded in INSERT statement.

```
dmSQL> insert into t1 values(funcs(1));
1 rows inserted

dmSQL> select * from t1;

    C1
===========
         101

1 rows selected
```

Embedded in UPDATE statement.

```
dmSQL> update t1 set c1=funcs(2);
1 rows updated

dmSQL> select * from t1;

    C1
===========
         102

1 rows selected
```

Embedded in CREATE TABLE statement.

```
dmSQL> create table t2 as select funcs(c1) from t1;
1 rows affected

dmSQL> select * from t2;

 FUNCS(C1)
===========
         202

1 rows selected
```

3.3. FUNCTION INFORMATION IN SYSTEM TABLE

After function is created, the function information will be stored in system table SYSUSERFUNC and SYSPROCPARAM, users can check these two tables for function and parameters information.

3.3.1. SYSUSERFUNC

The SYSUSERFUNC table contains information on user-defined and built-in functions. Because users need data in SYSUSERFUNC to execute UDF function, users that has at least the CONNECT authority can read all information.

Column Name	Description
MODE	Type of function: **1,2** — built-in function **0** — User-defined function
FILE_NAME	The name of the file that the built-in function is in. Or the function creator if the function is a user-defined function.
FUNC_NAME	The name of the function.
LANGUAGE_TYPE	Type of function: **C** — C function **LUA** — LUA function **SQL** — SQL function
RETURN_TYPE	Data type the function returns.
NUM_OF_PARAMETER	The number of parameters in the function.
PARAMETER	Data type of each parameter. The number of parameters is given by the value of NUM_OF_PARAMETER.

3.3.2. SYSPROCPARAM

The SYSPROCPARAM table contains information on stored procedure parameters and function parameters. The user with RESOURCE and CONNECT authority can read parameter information about the procedure/function which the user creates and is granted from other user. The user with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	Description
QUALIFIER	Qualifier.
OWNER	Owner of the procedure/function.
PROC_NAME	Name of the procedure/function.
PARAM_NAME	Parameter name.
PARAM_TYPE	Parameter type: **1** (SQL_PARAM_INPUT) — Input.

	3 (SQL_PARAM_OUTPUT) — Output. **4** (SQL_RETURN_VALUE) — Return value. **5** (SQL_RESULT_COL) — Result set.
DATA_TYPE	Data type.
TYPE_NAME	Type name.
PRECISION	Precision.
LENGTH	Length.
SCALE	Scale.
RADIX	Radix.
NULLABLE	Nullable column: **1** — Allows null values. **0** — Does not allow null values.
REMARKS	Remarks.
SID	Connection ID.

3.4. DROP FUNCTION SYNTAX

The following will show users how to delete an existing SQL function.

The syntax diagram for deleting SQL Function is as follows

```
<[DROP FUNCTION]>

    drop-function-statement ::= DROP FUNCTION function_name
```

The delete operation is consistent with the current delete function. The delete operation will delete the corresponding records of the two related system tables at the same time.

➲Example:
Delete SQL Function FUNC.

```
dmSQL> DROP FUNCTION FUNC;
```