Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #7

Today's Topics:

- Functions
- Procedures
- Cursors
- Type

FUNCTIONS

A user-defined function is a Transact-SQL routine that 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.

```
CREATE [ OR ALTER ] 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
[ ; ]
```

Where,

- **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.

- ***@parameter_name***: Is a parameter in the user-defined function. One or more parameters can be declared. 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.
- ***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 user-defined types and user-defined table types, are allowed except the **timestamp** data type.

  Note: 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.
- **return_data_type**: Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including user-defined types, are allowed except the **timestamp** data type.

Data types

Exact Numerics

Data type	Range	Storage
bigint	-2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807)	8 Bytes
int	-2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	4 Bytes
smallint	-2^15 (-32,768) to 2^15-1 (32,767)	2 Bytes
tinyint	0 to 255	1 Byte

Approximate Numerics

Data type	Range	Storage
float [**(***n***)**]	- 1.79E+308 to -2.23E-308, 0 and 2.23E-308 to 1.79E+308	Depends on the value of *n*
real	- 3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 to 3.40E + 38	4 Bytes

Date and Time

Data type	Default string literal forma
time	hh:mm:ss[.nnnnnnn]
date	YYYY-MM-DD
Datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm

Character Strings

Data type	Significance
char [(*n*)]	Fixed-length, non-Unicode string data. *n* defines the string length and must be a value from 1 through 8,000. The storage size is *n* bytes. The ISO synonym for char is character.
varchar [(*n* | max)]	Variable-length, non-Unicode string data. *n* defines the string length and can be a value from 1 through 8,000. max indicates that the maximum storage size is 2^31-1 bytes (2 GB). The ISO synonyms for varchar are charvarying or charactervarying.

Example 1: Total number of rows in Employee table

```
create function no_of_emp()
RETURNS integer
as
begin
declare @count integer
select @count = count(*) from employees;
return @count;
end

To display :
select dbo.no_of_emp() as 'number_of_employees';
```

Example 2: Maximum of two numbers

```
create function maximum
(@x integer , @y integer)
returns integer
as
begin
      declare @z integer
      if @x > @y
            set @z = @x;
      else
            set @z = @y;
      return @z;
end

To display : select dbo.maximum(5 , 10) as 'maximum';
```

Recursive Functions

Example 3: Recursive factorial

```
create function factorial
(@x integer)
returns integer
as
      begin
            declare @z integer
            if(@x = 0)
                  set @z =  1;
            else
                  set @z = @x * dbo.factorial(@x - 1);
            return @z;
      end

To display : select dbo.factorial(5) as 'factorial';
```

Example 4: Iterative factorial

```
create function factorial_iterative
(@x integer)
returns integer
as
      begin
            declare @z integer = 1
            if(@x = 0)
                  return 1;
            while(@x > 1)
            begin
                  set @z = @z * @x;
                  set @x = @x - 1;
            end
            return @z;
      end

TO display : select dbo.factorial_iterative(3) as 'factorial';
```

PROCEDURES

In SQL Server, a procedure is a stored program that you can pass parameters into. It does not return a value like a function does.

Creating your own stored procedures offers several advantages:

- You can avoid having to store all your T-SQL code in files. Stored procedures are stored in the database itself, so you never have to search through files to find the code you want to use.
- You can execute a stored procedure as often as you like from any machine that can connect to the database server.
- If you have a report that needs to be run frequently, you can create a stored procedure that produces the report. Anyone who has access to the database and permission to execute the stored procedure will be able to produce the report at will. They don't have to understand the T-SQL statements in the stored procedure. All they have to know is how to execute the stored procedure.
- You can enforce database security through stored procedures. You can grant users permission to execute a stored procedure but not permission to access to the underlying tables.
- Centralizing code in a stored procedure lets you reduce the amount of redundant code in your applications and insulate the applications from the effects of database schema changes.

The syntax to create a stored procedure in SQL Server (Transact-SQL) is:

```
CREATE { PROCEDURE | PROC } [schema_name.]procedure_name
   [ @parameter [type_schema_name.] datatype
     [ VARYING ] [ = default ] [ OUT | OUTPUT | READONLY ]
   , @parameter [type_schema_name.] datatype
     [ VARYING ] [ = default ] [ OUT | OUTPUT | READONLY ]
]

[ WITH { ENCRYPTION | RECOMPILE | EXECUTE AS Clause } ]
[ FOR REPLICATION ]
AS
BEGIN
   [declaration_section]
   executable_section
END;
```

Where,

- **schema_name:** The name of the schema that owns the stored procedure.
- **procedure_name:** The name to assign to this procedure in SQL Server.
- **@parameter:** One or more parameters passed into the procedure. 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.

- **type_schema_name:** The schema that owns the data type, if applicable.
- **Datatype:** The data type for @parameter.
- **VARYING:** It is specified for cursor parameters when the result set is an output parameter.
- **Default:** The default value to assign to @parameter.
- **OUT:** It means that @parameter is an output parameter.
- **OUTPUT:** It means that @parameter is an output parameter.
- **READONLY:** It means that @parameter cannot be overwritten by the stored procedure.
- **ENCRYPTION:** It means that the source for the stored procedure will not be stored as plain text in the system views in SQL Server.
- **RECOMPILE:** It means that a query plan will not be cached for this stored procedure.
- **EXECUTE AS clause:** It sets the security context to execute the stored procedure.
- **FOR REPLICATION:** It means that the stored procedure is executed only during replication.

Executing procedures

```
exec procedure_name @parameter [= value] [output];
```

Deleting Procedures

```
drop procedure procedure_name;
```

Example 1: Minimum of the two values

```
create procedure findmin (@x integer , @y integer , @z integer
output)
as
begin
      if(@x < @y)
      begin
            set @z = @x;
      end
      else
      begin
            set @z = @y;
      end
end
declare @a integer , @b integer, @c integer , @d integer;
begin
      set @a = 10;
```

```
        set @b = 20;
        exec findmin @a , @b , @c output;
        set @d = @c;
print 'The minimum of ' + cast(@a as varchar) + ' and ' +
cast(@b as varchar) + ' is ' + cast(@c as varchar);
end
```

Example 2: Square a number given as input

```
create procedure square_input(@x integer output)
as
begin
      set @x = @x * @x;
end
declare @a integer = 6;
begin
      exec square_input @a output;
      print @a;
end
```

CURSOR

Microsoft SQL Server statements produce a complete result set, but there are times when the results are best processed one row at a time. Opening a cursor on a result set allows processing the result set one row at a time. You can assign a cursor to a variable or parameter with a **cursor** data type.

```
Syntax :
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
     [ FORWARD_ONLY | SCROLL ]
     [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
     [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
     [ TYPE_WARNING ]
     FOR select_statement
     [ FOR UPDATE [ OF column_name [ ,...n ] ] ]
[;]
```

Where,

- **LOCAL:** Specifies that the scope of the cursor is local to the batch, stored procedure, or trigger in which the cursor was created. The cursor name is only valid within this

scope. The cursor can be referenced by local cursor variables in the batch, stored procedure, or trigger, or a stored procedure OUTPUT parameter. An OUTPUT parameter is used to pass the local cursor back to the calling batch, stored procedure, or trigger, which can assign the parameter to a cursor variable to reference the cursor after the stored procedure terminates. The cursor is implicitly deallocated when the batch, stored procedure, or trigger terminates, unless the cursor was passed back in an OUTPUT parameter. If it is passed back in an OUTPUT parameter, the cursor is deallocated when the last variable referencing it is deallocated or goes out of scope.

- **GLOBAL**: Specifies that the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection. The cursor is only implicitly deallocated at disconnect.
- **SCROLL:** Specifies that all fetch options (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) are available. If SCROLL is not specified in an ISO DECLARE CURSOR, NEXT is the only fetch option supported.
- **FORWARD_ONLY**: Specifies that the cursor can only be scrolled from the first to the last row. FETCH NEXT is the only supported fetch option. If FORWARD_ONLY is specified without the STATIC, KEYSET, or DYNAMIC keywords, the cursor operates as a DYNAMIC cursor. When neither FORWARD_ONLY nor SCROLL is specified, FORWARD_ONLY is the default, unless the keywords STATIC, KEYSET, or DYNAMIC are specified.
- **STATIC**: Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in **tempdb**; therefore, modifications made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow modifications.
- **KEYSET**: Specifies that the membership and order of rows in the cursor are fixed when the cursor is opened. The set of keys that uniquely identify the rows is built into a table in **tempdb** known as the **keyset**.
- **DYNAMIC**: Defines a cursor that reflects all data changes made to the rows in its result set as you scroll around the cursor. The data values, order, and membership of the rows can change on each fetch. The ABSOLUTE fetch option is not supported with dynamic cursors.
- **FAST_FORWARD**: Specifies a FORWARD_ONLY, READ_ONLY cursor with performance optimizations enabled. FAST_FORWARD cannot be specified if SCROLL or FOR_UPDATE is also specified.
- **READ_ONLY:** Prevents updates made through this cursor. The cursor cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. This option overrides the default capability of a cursor to be updated.
- **SCROLL_LOCKS**: Specifies that positioned updates or deletes made through the cursor are guaranteed to succeed. SQL Server locks the rows as they are read into the cursor to ensure their availability for later modifications. SCROLL_LOCKS cannot be specified if FAST_FORWARD or STATIC is also specified.
- **OPTIMISTIC**: Specifies that positioned updates or deletes made through the cursor do not succeed if the row has been updated since it was read into the cursor. SQL Server

does not lock rows as they are read into the cursor. It instead uses comparisons of **timestamp** column values, or a checksum value if the table has no **timestamp** column, to determine whether the row was modified after it was read into the cursor. If the row was modified, the attempted positioned update or delete fails. OPTIMISTIC cannot be specified if FAST_FORWARD is also specified.

- **TYPE_WARNING:** Specifies that a warning message is sent to the client when the cursor is implicitly converted from the requested type to another.
- ***select_statement***: Is a standard SELECT statement that defines the result set of the cursor.

Declaring a cursor

```
DECLARE ingredients_cursor CURSOR
FOR SELECT * FROM  ingredients;
```

Opening a cursor

```
open ingredients_cursor;
```

Fetching the cursor

```
FETCH NEXT FROM ingredients_cursor;
```

Closing the cursor

```
close ingredients_cursor;
```

Deleting cursor

```
deallocate ingredient_cursor;
```

Example 1: Displaying ingredientid, name and unit of all ingredients whose unit is not NULL

```
DECLARE ingredient_cursor CURSOR
FOR SELECT ingredientid, name, unit FROM ingredients where unit
is not null;
declare @id varchar(5) ,
@name varchar(30) ,
@unit varchar(10);
open ingredient_cursor;
FETCH NEXT FROM ingredient_cursor into @id, @name, @unit;
print(@id + ' ' + @name + ' ' + @unit);
WHILE @@FETCH_STATUS = 0
   BEGIN
      FETCH NEXT FROM ingredient_cursor into @id , @name ,
@unit;
      print(@id + ' ' + @name + ' ' + @unit);
```

```
    END;
CLOSE ingredient_cursor;
```

CURSOR UTILITIES

@@FETCH STATUS

Return value	Description
0	The FETCH statement was successful.
-1	The FETCH statement failed or the row was beyond the result set.
-2	The row fetched is missing.
-9	The cursor is not performing a fetch operation.

Because @@FETCH_STATUS is global to all cursors on a connection, use @@FETCH_STATUS carefully. After a FETCH statement is executed, the test for @@FETCH_STATUS must occur before any other FETCH statement is executed against another cursor. The value of @@FETCH_STATUS is undefined before any fetches have occurred on the connection.

@@Cursor Rows

Returns the number of qualifying rows currently in the last cursor opened on the connection

Return value	Description
-m	The cursor is populated asynchronously. The value returned (*-m*) is the number of rows currently in the keyset.
-1	The cursor is dynamic. Because dynamic cursors reflect all changes, the number of rows that qualify for the cursor is constantly changing. It can never be definitely stated that all qualified rows have been retrieved.
0	No cursors have been opened, no rows qualified for the last opened cursor, or the last-opened cursor is closed or deallocated.
n	The cursor is fully populated. The value returned (*n*) is the total number of rows in the cursor.

Type

User-defined table type is a user-defined type that represents the definition of a table structure. You can use a user-defined table type to declare table-valued parameters for stored procedures or functions, or to declare

table variables that you want to use in a batch or in the body of a stored procedure or function. To create a user-defined table type, use the CREATE TYPE statement. To ensure that the data in a user-defined table type meets specific requirements, you can create unique constraints and primary keys on the user-defined table type.

Syntax:

```
CREATE TYPE [ schema_name. ] type_name  {

   FROM base_type

   [ ( precision [ , scale ] ) ]

   [ NULL | NOT NULL ]

  | EXTERNAL NAME assembly_name [ .class_name ]

  | AS TABLE ( { <column_definition> | <computed_column_definition> }

      [ <table_constraint> ] [ ,...n ] )

} [ ; ]
```

Here is the way you would declare the Book record

```
create type books as table

(title varchar(50), author varchar(50), subject varchar(100),
book_id int);

Modifying and viewing the user defined table type

begin

declare @book1 dbo.books;

declare @book2 dbo.books;

insert into @book1(title , author,subject,book_id) values ('C
Programming'  , 'Nuha Ali' , 'C programming Tutorials' ,
123456);

insert into @book2(title , author,subject,book_id) values
('Telecom Billing'  , 'Zara Ali' , 'Telecom Billing Tutorial' ,
654321);

select * from @book1;

select * from @book2;

end;
```

----------&--------