Seminar 3. Stored procedures. Functions. Dynamic Execution. Cursors

Transact-SQL Server Stored Procedures

A stored procedure is a group of Transact-SQL statements compiled into a single execution plan.

The simplest stored procedure syntax:

```
CREATE PROCEDURE <Name> AS
   -- sequence of SQL commands
GO
```

A stored procedure is executed using EXEC:

```
EXEC <Name>
```

or just simply:

```
<Name>
```

Sample of a simple stored procedure:

```
CREATE PROCEDURE uspGetCourseNames
AS
      SELECT Cname
      FROM Courses
GO
```

and how it is executed:

```
EXEC  uspGetCourseNames
```

Sample of a simple stored procedure with a parameter:

```
ALTER PROCEDURE uspGetCourseNames(@credits INT)
AS
      SELECT Cname FROM Courses
      WHERE Credits = @credits
GO
```

and how it is executed:

```
EXEC  uspGetCourseNames 6
```

Sample of a simple stored procedure with an output parameter:

```
ALTER PROCEDURE uspGetCourseNames(@credits INT, @Number INT OUTPUT)
```

```
AS
	SELECT @Number = COUNT(*)
	FROM Courses
	WHERE Credits = @credits
GO
```

and how it is executed to have access to the value of the output parameter:

```
DECLARE @Number INT
SET @Number = 0

EXEC uspGetCourseNames 6, @Number=@Number OUTPUT

PRINT @Number
```

Another sample of a stored procedure containing a RAISERROR statement:

```
ALTER PROCEDURE uspGetCourseNames(@credits INT, @Number INT OUTPUT)
AS
BEGIN
	SELECT @Number = COUNT(*)
	FROM Courses
	WHERE Credits = @credits

	IF @Number = 0
		RAISERROR ('No courses found', 10, 1)
END
GO
```

The RAISERROR statement generates an error message and initiates error processing for the session. The syntax of the RAISERROR statement is:

```
RAISERROR ( { msg_id | msg_str | @local_var}
    { ,severity ,state } )
```

Transact-SQL Views

A view creates a *virtual table* that represents the data in one or more tables in an alternative way. A view contains rows and columns, just like a real table and can reference a maximum of 1,024 columns.
The SQL CREATE VIEW statement has the following syntax:

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name(s)
WHERE condition
```

Transact-SQL User Defined Functions

User defined functions allow developers to define their own functions to be used in SQL queries. There are three types of user defined functions in MS SQL Server:

- Scalar;
- Inline Table-Valued;
- Multi-Statement Table-Valued.

Scalar functions
Scalar functions return a single value (it does not matter what type it is, as long as it is only a single value).
The biggest drawback of scalar functions is that they may not be automatically inlined. For a scalar function that operates on multiple rows, SQL Server will execute the function once for every row in the result set and this can have a huge performance impact.

```
CREATE FUNCTION ufGetCourseNumber (@credits INT)
RETURNS INT AS

BEGIN
      DECLARE @Return INT
      SET @Return = 0

      SELECT @Return = COUNT(*)
      FROM Courses
      WHERE Credits = @credits

      RETURN @Return
END
-----------------------------------------------
PRINT dbo.ufGetCourseNumber(6)
```

Inline Table-Valued Functions
A table-valued user defined function returns a table instead of a single value. It can be used anywhere a table can be used – typically in the FROM clause of a query.

```
CREATE FUNCTION ufGetCourseNames (@credits INT)
RETURNS TABLE
AS
      RETURN
            SELECT CName
            FROM Courses
            WHERE Credits = @credits
---------------------------------------
SELECT * FROM dbo.ufGetCourseNames(6)
```

Multi-statement Table-Valued Functions
The difference between a multi-statement table-valued function and an inline table-valued function is that the first one contains more than one statement in the function body.

```
CREATE FUNCTION GetAuthorsByState ( @state CHAR(2) )
RETURNS @AuthorsByState table (au_id VARCHAR(11),
            au_fname VARCHAR(20))
AS
BEGIN

      INSERT INTO @AuthorsByState
      SELECT  au_id, au_fname FROM Authors
      WHERE state = @state

      IF @@ROWCOUNT = 0
      BEGIN
            INSERT INTO @AuthorsByState
            VALUES ('','No Authors Found')
      END

RETURN
END
GO
-----------------------------------------
SELECT * FROM dbo.GetAuthorsByState('CA')
```

Global Variables

SQL Server provides a massive number of global variables, which represent a special type of variable:

- the server always maintains the values of these variables;
- all the global variables represent information specific to the server or a current user session
- global variable names begin with the `@@` prefix;
- global variables need not be declared, since the server constantly maintains them (actually they are system-defined functions).

Examples of global variables:

@@ERROR - Contains the error number of the most recent TSQL error. 0 indicates that no error has occurred.
@@IDENTITY - Contains the IDENTITY field value of the most recently inserted row.
@@ROWCOUNT - Contains the number of rows affected by the most recent SELECT, INSERT, UPDATE, or DELETE command.
@@SERVERNAME - Contains the instance name.
@@SPID - Contains the session ID of the current user process.
@@VERSION - Contains system and build information for the current installation of the server.

System Tables

The information about all the objects (tables, fields, indexes, stored procedures, user defined functions, views, etc) created in a database is stored in special tables known as system tables. System tables should not be altered directly by any user, they being maintained exclusively by the server.
Examples of system tables:

sys.objects - contains one row for each object (constraint, stored procedure, view, etc) created within a database.
sys.columns - contains one row for every column of an object that has columns, e.g., tables, views.
sys.sql_modules - returns a row for each object that is an SQL language-defined module.

Dynamic Execution

Syntax:

```
EXEC (<command>)
```

Samples:

```
EXEC('SELECT OrderID, CustomerID FROM Orders WHERE OrderID = 1')
GO

DECLARE @var VARCHAR(MAX)
SET @var = 'SELECT OrderID, CustomerID FROM Orders WHERE OrderID = 1'
EXEC(@var)
GO
```

The main drawbacks of using dynamic execution are performance and (potential) security issues. An alternative to the EXEC statement is the stored procedure *sp_executesql* :

- it avoids many of the SQL injection problems;
- sometimes it is much faster than EXEC.

```
EXECUTE sp_executesql N'SELECT OrderID, CustomerID FROM Orders WHERE OrderID =
@ID ', N'@ID INT', @ID = 1;
```

Transact-SQL Cursors

A cursor is an entity that maps over a result set and establishes a position on a single row within the result set. After the cursor is positioned on a row, operations can be performed on that row or on a block of rows starting at that position.

Operations in a relational database act on a complete set of rows. The set of rows returned by a SELECT statement consists of all the rows that satisfy the conditions in the WHERE clause of the statement. This complete set of rows returned by the statement is known as the result set. Applications – especially interactive, online applications – cannot always work effectively with the entire result set as a unit. These applications need a mechanism to work with one row or with a small block of rows at a time. Cursors are an extension to result sets that provide this mechanism.

Cursors extend result processing by supporting the following functionalities:

- allowing positioning at specific rows of the result set;
- retrieving one row or block of rows from the current position in the result set;
- providing access to the data in a result set for Transact-SQL statements in scripts, stored procedures, and triggers.

Note that if the cursor performs the same operation against **every** row retrieved by cursor, a set-based operation is more efficient to use.

Transact-SQL Server cursors are based on the DECLARE CURSOR statement and are used mainly in Transact-SQL scripts, stored procedures, and triggers.

Transact-SQL cursors are implemented on the server and are managed by Transact-SQL statements sent from the client to the server. They are also contained in batches, stored procedures, or triggers.

When working with Transact-SQL cursors, you use a set of Transact-SQL statements to declare, populate, and retrieve data (as outlined in the following steps):

1. Use a DECLARE CURSOR statement to declare the cursor. When you declare the cursor, you should specify the SELECT statement that will produce the cursor's result set.
2. Use an OPEN statement to populate the cursor. This statement executes the SELECT statement embedded in the DECLARE CURSOR statement.
3. Use a FETCH statement to retrieve individual rows from the result set. Typically, a FETCH statement is executed many times (at least once for each row in the result set).
4. If appropriate, use an UPDATE or DELETE statement to modify the row. This step is optional.
5. Use a CLOSE statement to close the cursor. This process ends the active cursor operation and frees some resources (such as the cursor's result set and its locks on the current row). The cursor is still declared, so you can use an OPEN statement to reopen it.
6. Use a DEALLOCATE statement to remove the cursor reference from the current session. This process completely frees all the resources allocated to the cursor (including the cursor name). After a cursor is deallocated, you must issue a DECLARE statement to rebuild the

cursor (inside a stored procedure it is not necessary to close or deallocate a cursor. They are automatically executed when the stored procedure exits).

Referencing Transact-SQL Cursors

Only Transact-SQL statements can reference Transact-SQL cursor names and variables. The API functions of OLE DB, ODBC, ADO, and DB-Library cannot reference Transact-SQL cursor names and variables. Applications that need cursor processing and are using these APIs should use the cursor support built into the database API, instead of Transact-SQL cursors.

You can use Transact-SQL cursors in applications by using FETCH and by binding each column returned by the FETCH to a program variable. The Transact-SQL FETCH does not support batches, however, so this method is the least efficient way to return data to an application: fetching each row requires a round trip to the server. A more efficient way to fetch rows is to use the cursor functionality built into the database APIs that support batches.

Transact-SQL cursors are extremely efficient when contained in stored procedures and triggers. Everything is compiled into one execution plan on the server, and there is no network traffic associated with fetching rows.

Fetching and Scrolling

The operation to retrieve a row from a cursor is called a fetch. When working with Transact-SQL cursors, you can use FETCH statements to retrieve rows from a cursor's result set. A FETCH statement supports a number of options that enable you to retrieve specific rows:

- FETCH FIRST - fetches the first row in the cursor
- FETCH NEXT - fetches the row after the last row fetched
- FETCH PRIOR - fetches the row before the last row fetched
- FETCH LAST - fetches the last row in the cursor
- FETCH ABSOLUTE *n* - fetches the *n*th row from the first row in the cursor if *n* is a positive integer. If *n* is a negative integer, the row that is *n* rows before the end of the cursor is fetched. If *n* is 0, no rows are fetched.
- FETCH RELATIVE *n* - fetches the row that is *n* rows from the last row fetched. If *n* is positive, the row that is *n* rows after the last row fetched is fetched. If *n* is negative, the row that is *n* rows before the last row fetched is fetched. If *n* is 0, the same row is fetched again.

Controlling Cursor Behavior

There are two models for specifying the behavior of a cursor:

· Cursor types. The database APIs usually specify the behavior of cursors by dividing them into four cursor types: forward-only, static (sometimes called snapshot or insensitive), keyset-driven, and dynamic.

· Cursor behaviors. The SQL-92 standard defines the DECLARE CURSOR keywords SCROLL and INSENSITIVE to specify the behavior of cursors. Some database APIs also support defining cursor behavior in terms of scrollability and sensitivity.

Cursor 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 ] ] ]
```

Cursor Sample

```
DECLARE @ProductID      INT,
        @ProductName    VARCHAR(50),
        @ListPrice      MONEY

DECLARE cursorproducts CURSOR FOR
    SELECT ProductID, Name, ListPrice
    FROM Production.Product
    FOR READ ONLY

OPEN cursorproducts

FETCH cursorproducts INTO @ProductID, @ProductName, @ListPrice

WHILE @@FETCH_STATUS = 0
  BEGIN
     ……  --code for processing @ProductID,@ProductName, @ListPrice
  FETCH cursorproducts INTO @ProductID, @ProductName, @ListPrice
  END

CLOSE cursorproducts
DEALLOCATE cursorproducts
```