Procedural T-SQL and Stored Procedures in SQL Server 2005

Procedural T-SQL

- To perform processes that cannot be done using a single Transact-SQL statement, you can group Transact-SQL statements together in several ways:
  - **Batches**: A batch is a group of one or more Transact-SQL statements that are sent from an application to the server as one unit. Microsoft SQL Server 2005 executes each batch as a single executable unit.

Procedural T-SQL

- **Stored procedures**: A stored procedure is a group of Transact-SQL statements that have been predefined and precompiled on the server. The stored procedure can accept parameters, and can return result sets, return codes, and output parameters to the calling application.
- **Triggers**
- **Scripts**: A script is a series of Transact-SQL statements stored in a file. The file can be used as input to the **sqlcmd** utility or SQL Server Management Studio Code editor. The utilities then execute the Transact-SQL statements stored in the file.

Batches

- In ADO, a batch is the string of Transact-SQL statements enclosed in the **CommandText** property of a **Command** object
- In Microsoft SQL Server Management Studio and the **sqlcmd** utility the GO command signals the end of a batch. GO is not a Transact-SQL statement; it simply signals to the utilities how many SQL statements should be included in a batch.

Scripts

- A script is a series of Transact-SQL statements stored in a file.
- The file can be used as input to SQL Server Management Studio Code editor or the **sqlcmd** utility.
- Transact-SQL scripts have one or more batches. The GO command signals the end of a batch. If a Transact-SQL script does not have any GO commands, it is executed as a single batch.

T-SQL Variable

- To pass data between Transact-SQL statements you can use variables
- After a variable has been declared, or defined, one statement in a batch can set the variable to a value and a later statement in the batch can get the value from the variable.

Declaring a Variable

- The DECLARE statement initializes a Transact-SQL variable by:
  - Assigning a name. The name must have a single @ as the first character.
  - Assigning a system-supplied or user-defined data type and a length. For numeric variables, a precision and scale are also assigned. For variables of type XML, an optional schema collection may be assigned.
  - Setting the value to NULL.
- The scope of a variable lasts from the point it is declared until the end of the batch or stored procedure in which it is declared.

Example

```
DECLARE @MyCounter int;

DECLARE @LastName nvarchar(30), @FirstName
    nvarchar(20), @StateProvince nchar(2);
```

Setting a Value in a Variable

- To assign a value to a variable, use the SET statement. This is the preferred method of assigning a value to a variable. Syntax

SET @*local_variable* **=** *expression*

- A variable can also have a value assigned by being referenced in the select list of a SELECT statement.
  - If a variable is referenced in a select list, the SELECT statement should only return one row.

SET Example

```
USE AdventureWorks;
GO
-- Declare two variables.
DECLARE @FirstNameVariable nvarchar(50),
  @PostalCodeVariable nvarchar(15);
-- Set their values.
SET @FirstNameVariable = N'Amy';
SET @PostalCodeVariable = N'BA5 3HX';
-- Use them in the WHERE clause of a SELECT statement.
SELECT LastName, FirstName, JobTitle, City,
    StateProvinceName, CountryRegionName
FROM HumanResources.vEmployee
WHERE FirstName = @FirstNameVariable
  OR PostalCode = @PostalCodeVariable;
GO
```

SET Example

```
USE AdventureWorks;
GO
DECLARE @rows int;
SET @rows = (SELECT COUNT(*) FROM
    Sales.Customer);
SELECT @rows;
```

Example

```
USE AdventureWorks;
GO
DECLARE @EmpIDVariable int;

SELECT @EmpIDVariable = MAX(EmployeeID)
FROM HumanResources.Employee;
GO
```

Setting a Variable with a SELECT

- If a SELECT statement returns more than one row, the variable is set to the value returned for the expression in the last row of the result set.
- For example, in this batch **@EmpIDVariable** is set to the **EmployeeID** value of the last row returned, which is 1:

```
USE AdventureWorks;
GO
DECLARE @EmpIDVariable int;
SELECT @EmpIDVariable = EmployeeID
FROM HumanResources.Employee
ORDER BY EmployeeID DESC;
SELECT @EmpIDVariable;
GO
```

Control of Flow

- BEGIN...END
- GOTO
- IF...ELSE
- RETURN
- WAITFOR
- WHILE, BREAK, CONTINUE
- CASE

BEGIN...END

- The BEGIN and END statements are used to group multiple Transact-SQL statements into a logical block.
- Use the BEGIN and END statements anywhere a control-of-flow statement must execute a block of two or more Transact-SQL statements
- The BEGIN and END statements are used when:
  - A WHILE loop needs to include a block of statements.
  - An element of a CASE function needs to include a block of statements.
  - An IF or ELSE clause needs to include a block of statements.

BEGIN…END

- When an IF statement controls the execution of only one Transact-SQL statement, no BEGIN or END statement is needed:

```
IF (@@ERROR <> 0)
  SET @ErrorSaveVariable = @@ERROR
```

- If more than statement must be executed use BEGIN…END

```
IF (@@ERROR <> 0)
BEGIN
  SET @ErrorSaveVariable = @@ERROR
  PRINT 'Error encountered, ' +
        CAST(@ErrorSaveVariable AS VARCHAR(10))
END
```

GOTO

- The GOTO statement causes the execution of a Transact-SQL batch to jump to a label.
- The label name is defined using the syntax

label_name:

- before a statement
- GOTO is best used for breaking out of deeply nested control-of-flow statements.

GOTO Example

```
IF (SELECT SYSTEM_USER()) = 'payroll'
   GOTO calculate_salary
-- Other program code would appear here.
-- When the IF statement evaluates to TRUE, the statements
-- between the GOTO and the calculate_salary label are
-- ignored. When the IF statement evaluates to FALSE the
-- statements following the GOTO are executed.
calculate_salary:
   -- Statements to calculate a salary would appear after the label.
```

IF…ELSE

- Syntax

IF *Boolean_expression*

{ *sql_statement* | *statement_block* }

[ELSE { *sql_statement* | *statement_block* }]

IF…ELSE Example

```
IF (@ErrorSaveVariable <> 0)
BEGIN
  PRINT 'Errors encountered, rolling back.'
  PRINT 'Last error encountered: ' +
    CAST(@ErrorSaveVariable AS VARCHAR(10))
  ROLLBACK
END
ELSE
BEGIN
  PRINT 'No Errors encountered, committing.'
  COMMIT
END
RETURN @ErrorSaveVariable
```

RETURN

- The RETURN statement unconditionally terminates a script, stored procedure, or batch. None of the statements in a stored procedure or batch following the RETURN statement are executed.
- When used in a stored procedure, the RETURN statement can specify an integer value to return to the calling application, batch, or procedure. If no value is specified on RETURN, a stored procedure returns the value 0.
- Most stored procedures follow the convention of using the return code to indicate the success or failure of the stored procedure. The stored procedures return a value of 0 when no errors were encountered. Any nonzero value indicates that an error occurred.

WAITFOR

- The WAITFOR statement suspends the execution of a batch, stored procedure, or transaction until:
  - A specified time interval has passed.
  - A specified time of day is reached.
- The WAITFOR statement is specified with one of the following clauses:
  - The DELAY keyword followed by a *time_to_pass* before completing the WAITFOR statement. The time to wait before completing the WAITFOR statement can be up to 24 hours.
  - The TIME keyword followed by a *time_to_execute*, which specifies the time that the WAITFOR statement completes

WAITFOR Examples

```
-- waits 2 seconds
WAITFOR DELAY '00:00:02'
SELECT EmployeeID FROM
    AdventureWorks.HumanResources.Employee;

-- executes the query at 22:00
WAITFOR TIME '22:00';
SELECT EmployeeID FROM
    AdventureWorks.HumanResources.Employee;
```

WHILE

- Syntax

WHILE *Boolean_expression*
{ *sql_statement* | *statement_block* }

- The statement block may contain BREAK and/or CONTINUE

BREAK and CONTINUE

- BREAK
  - Causes an exit from the innermost WHILE loop. Any statements that appear after the END keyword, marking the end of the loop, are executed.
- CONTINUE
  - Causes the WHILE loop to restart, ignoring any statements in the loop after the CONTINUE keyword.

Example

```
USE AdventureWorks;
GO
DECLARE abc CURSOR FOR
SELECT * FROM Purchasing.ShipMethod;
OPEN abc;
FETCH NEXT FROM abc
WHILE (@@FETCH_STATUS = 0)
   FETCH NEXT FROM abc;
CLOSE abc;
DEALLOCATE abc;
GO
```

CASE

- Syntax: two forms
  - The simple CASE function compares an expression to a set of simple expressions to determine the result.
  - The searched CASE function evaluates a set of Boolean expressions to determine the result.

- **Simple CASE function:**

```
CASE input_expression
     WHEN when_expression THEN result_expression    [ ...n ]
     [    ELSE else_result_expression     ]
END
```

CASE

- **Searched CASE function:**

```
CASE
    WHEN Boolean_expression THEN result_expression
        [ ...n ]
    [    ELSE else_result_expression      ]
END
```

Simple CASE function

- Evaluates *input_expression*, and then in the order specified, evaluates *input_expression* = *when_expression* for each WHEN clause.
- Returns the *result_expression* of the first *input_expression* = *when_expression* that evaluates to TRUE.
- If no *input_expression* = *when_expression* evaluates to TRUE, the Database Engine returns the *else_result_expression* if an ELSE clause is specified, or a NULL value if no ELSE clause is specified.

Searched CASE function

- Evaluates, in the order specified, *Boolean_expression* for each WHEN clause.
- Returns *result_expression* of the first *Boolean_expression* that evaluates to TRUE.
- If no *Boolean_expression* evaluates to TRUE, the Database Engine returns the *else_result_expression* if an ELSE clause is specified, or a NULL value if no ELSE clause is specified.

Simple CASE Example

```
USE AdventureWorks;
GO
SELECT   ProductNumber, Category =
     CASE ProductLine
        WHEN 'R' THEN 'Road'
        WHEN 'M' THEN 'Mountain'
        WHEN 'T' THEN 'Touring'
        WHEN 'S' THEN 'Other sale items'
        ELSE 'Not for sale'
     END,
  Name
FROM Production.Product
ORDER BY ProductNumber;
GO
```

Simple CASE Example

```
USE AdventureWorks
GO
SELECT Name,
  CASE Name
    WHEN 'Human Resources' THEN 'HR'
    WHEN 'Finance' THEN 'FI'
    WHEN 'Information Services' THEN 'IS'
    WHEN 'Executive' THEN 'EX'
    WHEN 'Facilities and Maintenance' THEN 'FM'
  END AS Abbreviation
FROM AdventureWorks.HumanResources.Department
WHERE GroupName = 'Executive General and Administration'
```

Searched CASE Example

```
USE AdventureWorks;
GO
SELECT   ProductNumber, Name, 'Price Range' =
     CASE
        WHEN ListPrice =  0 THEN 'Mfg item - not for resale'
        WHEN ListPrice < 50 THEN 'Under $50'
        WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under
    $250'
        WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under
    $1000'
        ELSE 'Over $1000'
     END
FROM Production.Product
ORDER BY ProductNumber ;
GO
```

Expressions

- An expression is a combination of identifiers, values, and operators that SQL Server 2005 can evaluate to obtain a result.
- Expressions can be used, for example, as part of the data to retrieve in a query, or as a search condition when looking for data that meets a set of criteria.

Expressions

- An expression can be any of the following:
  - Constant
  - Function
  - Column name
  - Variable
  - Subquery
  - CASE
- An expression can also be built from combinations of these entities joined by operators.

Operators

- Arithmetic Operators
- Logical Operators
- String Concatenation Operator
- Bitwise Operators
- Unary Operators
- Comparison Operators

NULL Values

- A value of NULL indicates that the data is unknown, not applicable, or that the data will be added later
- A value of NULL is different from an empty or zero value.
- No two null values are equal. Comparisons between two null values, or between a NULL and any other value, return unknown because the value of each NULL is unknown.

NULL Values

- To test for null values in a query, use IS NULL or IS NOT NULL in the WHERE clause.
- When query results are viewed in SQL Server Management Studio Code editor, null values are shown as **(null)** in the result set.
- Null values can be inserted into a column by explicitly stating NULL in an INSERT or UPDATE statement, by leaving a column out of an INSERT statement, or when adding a new column to an existing table by using the ALTER TABLE statement.
- Null values cannot be used for information that is required to distinguish one row in a table from another row in a table, for example, foreign or primary keys.

NULL Values

- When null values are present in data, logical and comparison operators can potentially return a third result of UNKNOWN instead of just TRUE or FALSE.
- A row for which the WHERE condition evaluates to UNKNOWN is not returned by the selection
- Truth table for AND

AND	**TRUE**	**UNKNOWN**	**FALSE**
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	UNKNOWN	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE

Truth Table for OR and NOT

OR	TRUE	UNKNOWN	FALSE
TRUE	TRUE	TRUE	TRUE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	UNKNOWN	FALSE

NOT	Evaluates to
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

NULL Values

- Transact-SQL also offers an extension for null processing.
- If the option ANSI_NULLS is set to OFF, comparisons between nulls, such as NULL = NULL, evaluate to TRUE. Comparisons between NULL and any data value evaluate to FALSE.

Stored procedures

- Microsoft’s implementation of SQL-2003 PSM
- When using Transact-SQL programs, two methods are available for storing and executing the programs.
  - You can store the programs in the client and create applications that send the commands to SQL Server and process the results.
  - You can store the programs as stored procedures in SQL Server and create applications that execute the stored procedures and process the results.

Benefits of Stored Procedures

- The benefits of using stored procedures in SQL Server rather than Transact-SQL programs stored locally on client computers are:
  - They can have security attributes (such as permissions). Users can be granted permission to execute a stored procedure without having to have direct permissions on the objects referenced in the procedure.
  - They can enhance the security of your application. Parameterized stored procedures can help protect your application from SQL Injection attacks.

Benefits of Stored Procedures

- They allow modular programming.
  You can create the procedure once, and call it any number of times in your program. This can improve the maintainability of your application and allow applications to access the database in a uniform manner.
- They can reduce network traffic.
  An operation requiring hundreds of lines of Transact-SQL code can be performed through a single statement that executes the code in a procedure, rather than by sending hundreds of lines of code over the network.

Benefits of Stored Procedures

- They allow faster execution. Transact-SQL stored procedures reduce the compilation cost of Transact-SQL code by caching the plans and reusing them for repeated executions. This means the stored procedure does not need to be reparsed and reoptimized with each use resulting in much faster execution times

SQL Injection

- SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution
- The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed

Example

```
var Shipcity;
ShipCity = Request.form ("ShipCity");
var sql = "select * from OrdersTable where ShipCity = '"
    + ShipCity + "'";
```

- The user is prompted to enter the name of a city. If she enters Redmond, the query assembled by the script looks similar to the following:

```
SELECT * FROM OrdersTable WHERE ShipCity =
    'Redmond'
```

Example

- However, assume that the user enters the following:

Redmond'; drop table OrdersTable--

- In this case, the following query is assembled by the script:

SELECT * FROM OrdersTable WHERE ShipCity = 'Redmond';drop table OrdersTable--'

Example

- The semicolon (;) denotes the end of one query and the start of another.
- The double hyphen (--) indicates that the rest of the current line is a comment and should be ignored.
- Since the modified code is syntactically correct, it will be executed by the server. SQL Server will first select all records in OrdersTable where ShipCity is Redmond. Then, SQL Server will drop OrdersTable.

Types of Stored Procedures

- User-defined:
  - Transact-SQL: procedures written in Transact-SQL
  - CLR: A CLR stored procedure is a reference to a Microsoft .NET Framework common language runtime (CLR) method that can take and return user-supplied parameters. They are implemented as public, static methods on a class in a .NET Framework assembly.
- System stored procedures: they have the **sp_** prefix. System stored procedures logically appear in the **sys** schema of every system- and user-defined database.

Stored Procedures

- Almost any Transact-SQL code that can be written as a batch can be used to create a stored procedure. SELECT statements can be used (differently from SQL/PSM)
- To create a stored procedure use CREATE PROCEDURE

Stored Procedures

- When creating a stored procedure, you should specify:
  - Any input parameters and output parameters to the calling procedure or batch.
  - The programming statements that perform operations in the database, including calling other procedures.
  - The status value returned to the calling procedure or batch to indicate success or failure (and the reason for failure).
  - Any error handling statements needed to catch and handle potential errors.

Naming Stored Procedures

- They are schema level objects, so if the schema is not specified the procedure is created in the default schema
- Do not start a procedure with “sp_” to avoid confusion with system stored procedures

CREATE PROCEDURE

CREATE PROCEDURE [*schema_name*.]
procedure_name

[{ **@***parameter* [*type_schema_name*.] *data_type*}
[**=** *default*] [OUT | OUTPUT]   ] [**,**...*n*]

AS { <sql_statement> [;][...*n*] | <method_specifier> } [;]

Example

- The following stored procedure returns the first and last name, the titles, and the department names of employees from a view. This stored procedure does not use any parameters

```
USE AdventureWorks;
GO
CREATE PROCEDURE
   HumanResources.uspGetAllEmployees
AS
   SELECT LastName, FirstName, JobTitle, Department
   FROM HumanResources.vEmployeeDepartment;
GO
```

Procedure Execution

- The uspGetEmployees stored procedure can be executed in three ways:

```
EXECUTE HumanResources.uspGetAllEmployees;
GO
-- Or
EXEC HumanResources.uspGetAllEmployees;
GO
-- Or, if this procedure is the first statement within a
   batch:
HumanResources.uspGetAllEmployees;
```

Parameters

- Their name must start with @ and must follow the rules for object identifiers.
- The parameter name can be used in the stored procedure to obtain and change the value of the parameter.
- Parameters in a stored procedure are defined with a data type, much as a column in a table is defined.
- A stored procedure parameter can be defined with any of the SQL Server 2005 data types, except the **table** data type. Stored procedure parameters can also be defined with CLR user-defined types

Parameters

- The data type of a parameter determines the type and range of values that are accepted for the parameter.
- For example, if you define a parameter with a **tinyint** data type, only numeric values ranging from 0 to 255 are accepted.
- An error is returned if a stored procedure is executed with a value incompatible with the data type.

Direction of a Parameter

- The direction of a parameter is either in, meaning a value is passed to the stored procedure, or out, meaning the stored procedure returns a value to the calling program. The default is an input parameter.
- To specify an output parameter, you must specify the OUTPUT or OUT keyword in the definition of the parameter in the stored procedure.
- The stored procedure returns the current value of the output parameter to the calling program when the stored procedure exits. The calling program must also use the OUTPUT keyword when executing the stored procedure to save the parameter's value in a variable of the calling program.

Calling a Procedure With Parameters

- Values can be passed to stored procedures
  - by explicitly naming the parameters and assigning the appropriate value or
  - by supplying the the values of parameters in the order in which the parameters have been defined in the CREATE PROCEDURE. In this case the parameters are not named

Example

```
USE AdventureWorks;
GO
CREATE PROCEDURE HumanResources.uspGetEmployees
   @LastName nvarchar(50),
   @FirstName nvarchar(50)
AS
   SELECT FirstName, LastName, JobTitle, Department
   FROM HumanResources.vEmployeeDepartment
   WHERE FirstName = @FirstName AND LastName =
    @LastName;
GO
```

Example

- The uspGetEmployees stored procedure can be executed in the following ways:

```
EXEC HumanResources.uspGetEmployees @LastName =
    N'Ackerman', @FirstName = N'Pilar';
GO
-- Or
EXECUTE HumanResources.uspGetEmployees @FirstName =
    N'Pilar', @LastName = N'Ackerman';
GO
-- Or
EXECUTE HumanResources.uspGetEmployees N'Ackerman',
    N'Pilar';
-- Or, if this procedure is the first statement within a batch:
HumanResources.uspGetEmployees N'Ackerman', N'Pilar';
```

Default Values

- If default values are specified for a parameter in the procedure definition, the parameter can be left unspecified in a procedure call

Example

```
USE AdventureWorks;
GO
CREATE PROCEDURE HumanResources.uspGetEmployees2
   @LastName nvarchar(50) = N'D%',
   @FirstName nvarchar(50) = N'%'
AS
   SELECT FirstName, LastName, JobTitle, Department
   FROM HumanResources.vEmployeeDepartment
   WHERE FirstName LIKE @FirstName
      AND LastName LIKE @LastName;
GO
```

Example of Execution

```
EXECUTE HumanResources.uspGetEmployees2;
-- Or
EXECUTE HumanResources.uspGetEmployees2 N'Wi%';
-- Or
EXECUTE HumanResources.uspGetEmployees2 @FirstNname =
    N‘F%';
-- Or
EXECUTE HumanResources.uspGetEmployees2 N'[CK]ars[OE]n';
-- Or
EXECUTE HumanResources.uspGetEmployees2 N'Hesse',
    N'Stefen';
```

OUTPUT Parameters

- If you specify the OUTPUT keyword for a parameter in the procedure definition, the stored procedure can return the current value of the parameter to the calling program when the stored procedure exits.
- To save the output value of the parameter in a variable, the calling program must use the OUTPUT keyword when executing the stored procedure

OUTPUT Parameters Example

```
USE AdventureWorks;
GO
CREATE PROCEDURE Sales.uspGetEmployeeSalesYTD
@SalesPerson nvarchar(50),
@SalesYTD money OUTPUT
AS
SELECT @SalesYTD = SalesYTD
FROM Sales.SalesPerson AS sp
JOIN HumanResources.vEmployee AS e ON e.EmployeeID =
    sp.SalesPersonID
WHERE LastName = @SalesPerson;
RETURN
GO
```

OUTPUT Parameters Example

```
-- Declare the variable to receive the output value of the procedure.
DECLARE @SalesYTDBySalesPerson money;
-- Execute the procedure specifying a last name for the input
-- parameter and saving the output value in the variable
-- @SalesYTD
EXECUTE Sales.uspGetEmployeeSalesYTD
   N'Blythe', @SalesYTD = @SalesYTDBySalesPerson OUTPUT;
-- Display the value returned by the procedure.
PRINT 'Year-to-date sales for this employee is ' +
   convert(varchar(10),@SalesYTDBySalesPerson);
GO
```

OUTPUT Parameters

- Input values can also be specified for OUTPUT parameters when the stored procedure is executed.
- This allows the stored procedure to receive a value from the calling program, change it or perform operations with it, then return the new value to the calling program.
- In the previous example, the @SalesYTDBySalesPerson variable can be assigned a value prior to executing the stored procedure. The @SalesYTD parameter initially contains that value in the body of the stored procedure, and the value of the @SalesYTD variable is returned to the calling program when the stored procedure exits.
- It is like the type INOUT of PSM

Return Value

- A stored procedure can return an integer value called a return code to indicate the execution status of a procedure.
- You specify the return code for a stored procedure using the RETURN statement.
- As with OUTPUT parameters, you must save the return code in a variable when the stored procedure is executed to use the return code value in the calling program. E.g.

```
DECLARE @result int;
EXECUTE @result = my_proc;
```

Nesting

- Stored procedures are nested when one stored procedure calls another or executes managed code by referencing a CLR routine.
- You can nest stored procedures and managed code references up to 32 levels.
- Attempting to exceed the maximum of 32 levels of nesting causes the whole calling chain to fail.

Cursors

- The typical process for using a Transact-SQL cursor in a stored procedure or trigger is:
  1. Declare variables to contain the data returned by the cursor. Declare one variable for each result set column. Declare the variables to be large enough to hold the values returned by the column and with a data type that can be implicitly converted from the data type of the column.
  2. Associate a Transact-SQL cursor with a SELECT statement using the DECLARE CURSOR statement. The DECLARE CURSOR statement also defines the characteristics of the cursor.
  3. Use the OPEN statement to execute the SELECT statement and populate the cursor.

Cursors

4. Use the FETCH INTO statement to fetch individual rows and have the data for each column moved into the specified variables. Other Transact-SQL statements can then reference those variables to access the fetched data values.
5. When you are finished with the cursor, use the CLOSE statement. Closing a cursor frees some resources, such as the cursor's result set and its locks on the current row, but the cursor structure is still available for processing if you reissue an OPEN statement.

Syntax

DECLARE *cursor_name* [INSENSITIVE] [SCROLL]
    CURSOR FOR *select_statement*
    [FOR { READ ONLY | UPDATE [OF *column_name* [,...n]] }]

- INSENSITIVE: Defines a cursor that makes a temporary copy of the data. 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. If INSENSITIVE is omitted, committed deletes and updates made to the underlying tables (by any user) are reflected in subsequent fetches

Syntax

- SCROLL: Specifies that all fetch options (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) are available. If SCROLL is not specified in DECLARE CURSOR, NEXT is the only fetch option supported.

Syntax

- 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.
- UPDATE [OF *column_name* [**,**...*n*]]: Defines updatable columns within the cursor. If OF *column_name* [**,**...*n*] is specified, only the columns listed allow modifications. If UPDATE is specified without a column list, all columns can be updated.

Syntax

FETCH [[NEXT | PRIOR | FIRST | LAST
| ABSOLUTE { *n* | @*nvar* }
| RELATIVE { *n* | @*nvar*}]
FROM]
cursor_name [INTO @*variable_name* [,...*n*]]

The @@FETCH_STATUS function reports the status of the last FETCH statement. It returns 0 if the last FETCH statement was successful

Example

- In the following example the results of the fetch are returned to the user

```
USE AdventureWorks
GO
DECLARE contact_cursor CURSOR FOR
SELECT LastName FROM Person.Contact
WHERE LastName LIKE 'B%'
ORDER BY LastName
OPEN contact_cursor
```

Example

```
-- Perform the first fetch.
FETCH NEXT FROM contact_cursor
-- Check @@FETCH_STATUS to see if there are any more rows
    to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
  -- This is executed as long as the previous fetch succeeds.
  FETCH NEXT FROM contact_cursor
END
CLOSE contact_cursor
DEALLOCATE contact_cursor
GO
```

Example

```
USE AdventureWorks
GO
-- Declare the variables to store the values returned by FETCH.
DECLARE @LastName varchar(50), @FirstName varchar(50)
DECLARE contact_cursor CURSOR FOR
SELECT LastName, FirstName FROM Person.Contact
WHERE LastName LIKE 'B%'
ORDER BY LastName, FirstName
OPEN contact_cursor
-- Perform the first fetch and store the values in variables.
-- Note: The variables are in the same order as the columns
-- in the SELECT statement.
FETCH NEXT FROM contact_cursor
INTO @LastName, @FirstName
```

Example

```
-- Check @@FETCH_STATUS to see if there are any more rows to
   fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
  -- Concatenate and display the current values in the variables.
  PRINT 'Contact Name: ' + @FirstName + ' ' +  @LastName
  -- This is executed as long as the previous fetch succeeds.
  FETCH NEXT FROM contact_cursor
  INTO @LastName, @FirstName
END
CLOSE contact_cursor
DEALLOCATE contact_cursor
GO
```

User Defined Functions

- Like functions in programming languages, Microsoft SQL Server 2005 user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a table.
- The benefits of using user-defined functions in SQL Server are the same as for stored procedures

User Defined Functions

- In SQL Server 2005, user-defined functions can be written in Transact-SQL, or in any .NET programming language
- All user-defined functions have the same two-part structure: a header and a body. The function takes zero or more input parameters.

Types of Functions

- Scalar functions
- Inline table functions
- Multistatement table functions

Invocation

- 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.
- Table-valued functions can be invoked where table expressions are allowed in the FROM clause of SELECT, UPDATE, or DELETE statements.

Syntax of Scalar Functions

```
CREATE FUNCTION [ schema_name. ] function_name
    ( [ { @parameter_name [ AS ][ type_schema_name. ]
          parameter_data_type
          [ = default ] }    [ ,...n ]   ] )
    RETURNS return_data_type
[ AS ]
BEGIN
    function_body
    RETURN scalar_expression
END [ ; ]
```

Example

```
CREATE FUNCTION dbo.GetWeekDay
-- function name
(@Date datetime) -- input parameter name and data
                 -- type
RETURNS int      -- return parameter data type
AS
BEGIN                          -- begin body definition
RETURN DATEPART (weekday, @Date)
   -- action performed
END;
GO
```

Example

```
SELECT
    dbo.GetWeekDay(CONVERT(DATETIME,'20020201'
    ,101)) AS DayOfWeek;
GO
```

- Result

```
DayOfWeek
---------
6
(1 row(s) affected)
```

Example

```
DECLARE @weekday int
EXECUTE @weekday= dbo.GetWeekDay
   @Date=CONVERT(DATETIME,'20020201',101)
```

Syntax of Inline Table Functions

```
CREATE FUNCTION [ schema_name. ] function_name
   ( [ { @parameter_name [ AS ][ type_schema_name. ]
         parameter_data_type
         [ = default ] }    [ ,...n ]   ] )
RETURNS TABLE
[ AS ]
RETURN [ ( ] select_stmt [ ) ] [ ; ]
```

Example

```
USE AdventureWorks;
GO
CREATE FUNCTION Sales.ufn_SalesByCustomer (@custID int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'YTD 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
    WHERE SH.CustomerID = @custID
    GROUP BY P.ProductID, P.Name
);
GO
```

Example of Invocation of the Function

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

Multistatement table functions

```
CREATE FUNCTION [ schema_name. ] function_name
   ( [ { @parameter_name [ AS ][ type_schema_name. ]
         parameter_data_type
         [ = default ] }    [ ,...n ]   ] )
RETURNS @return_variable TABLE
< table_type_definition >
[ AS ]
BEGIN
   function_body
   RETURN
END [ ; ]
```

Multistatement table functions

- The table to be returned is stored into @*return_variable* in the body of the function
- When RETURN is executed, the result is returned to the calling statement

Built-in Functions

- Built-in functions are provided by SQL Server to help you perform a variety of operations. They cannot be modified. You can use built-in functions in Transact-SQL statements to:
  - Access information from SQL Server system tables without accessing the system tables directly (system functions).
  - Perform common tasks such as SUM, GETDATE, or IDENTITY.

Built-in Functions

- Built-in functions return either scalar or **table** data types.
- For example, @@ERROR returns 0 if the last Transact-SQL statement executed successfully. If the statement generated an error, @@ERROR returns the error number. And the function SUM(*parameter*) returns the sum of all the values for the parameter.

System Functions

- The names of some Transact-SQL system functions start with two at signs (@@).
- The @@functions are system functions, and their syntax usage follows the rules for functions.
- Example

```
SELECT SUSER_NAME()
```

- Retrieve the user name for the current user that is logged on by using SQL Server Authentication