# T-SQL

Contents

**1. Declaring Variables** ....2
**2. SET** ....3
**3. Print** ....4
**4. WAITFOR** ....4
**5. IF Statement** ....5
**6. While Statement** ....8
**7. Stored Procedures** ....9
**8. Benefits of Stored Procedures** ....10
**9. Creating Procedure** ....11
**10. Alter Procedure** ....12
**11. Drop Procedure** ....13
**12. Improve Procedure Performance** ....14

## 1. Declaring Variables

- You use variables to temporarily store data values for later use in the same batch in which they were declared. The batches will be described later, but for now, the important thing for you to know is that a batch is one T-SQL statement or more sent to Microsoft SQL Server for execution as a single unit.
- Use a DECLARE statement to declare one or more variables, and use a SET statement to assign a value to a single variable.
  - **DECLARE @VarName Datatype [=Value], @VarName2 Datatype [=Value], ……. ;**

- **Example**
  - *Declare @ID Int;*
  - *Declare @name varchar(20), @address varchar(40);*
- To assign value to specific variable use SET:
  - *SET @ID=1;*
  - *SET @name='Ali;*
- Alternatively, you can declare and initialize a variable in the same statement, like this:
  - *Declare @ID int =1;*
- The SET statement can operate on only one variable at a time, so if you need to assign values to multiple variables, you need to use multiple SET statements. T-SQL also supports a nonstandard assignment SELECT statement, which you use to query data and assign multiple values obtained from the same row to multiple variables by using a single statement. Here's an example:
  - *Declare @ID int, @name varchar(50);*
  - *Select @ID=id, @name=name from info;*

## 2. SET

- Sets the specified local variable, previously created by using the DECLARE @local_variable statement, to the specified value.
  - *SET @Varname = Value;*
  - *DECLARE @ID int;*
  - *SET @ID=1;*
- The following example creates the @myvar variable, puts a string value into the variable, and prints the value of the @myvar variable.
  - *DECLARE @myvar char(20);*
  - *SET @myvar = 'This is a test';*
  - *SELECT @myvar;*
- SET can be used in compound assignment of variables. The following two examples produce the same result. They create a local variable named @NewBalance, multiplies it by 10 and displays the new value of the local variable in a SELECT statement. The second example uses a compound assignment operator.
  - */* Example one */*
  - *DECLARE @NewBalance int ;*
  - *SET @NewBalance = 10;*
  - *SET @NewBalance = @NewBalance * 10;*
  - *SELECT @NewBalance;*
  - */* Example Two */*
  - *DECLARE @NewBalance int = 10;*
  - *SET @NewBalance *= 10;*
  - *SELECT @NewBalance;*
- SET can be used in assigning a value from a query. The following example uses a query to assign a value to a variable.
  - *DECLARE @rows int;*

- *SET @rows = (SELECT COUNT(*) FROM Sales.Customer);*
- *SELECT @rows;*

- Other compound operations (+=, -=, /=, %=, &=, |=)

### 3. Print

- Print in T-SQL can be used to print messages as statements or variable contents.
  - ***PRINT** msg_str | @local_variable | string_expr*
- The following example prints the current day name:
  - *PRINT 'Current Day is '+DATENAME(dw,GETDATE());*
- Print can be used in displaying result of arithmetic operations such as:
  - *PRINT 12/4*
- Print can be used with variables as shown in example:
  - *Declare @ID int =1;*
  - *PRINT @ID;*

### 4. WAITFOR

- **WAITFOR { DELAY 'time_to_pass' | TIME 'time_to_execute'}**

- **DELAY:** is the specified period of time that must pass, up to a maximum of 24 hours, before execution of a batch, stored procedure, or transaction proceeds.
- **'time_to_pass': i**s the period of time to wait. time_to_pass can be specified in one of the acceptable formats for datetime data, or it can be specified as a local variable. Dates cannot be specified; therefore, the date part of the datetime value is not allowed.
- **TIME:** is the specified time when the batch, stored procedure, or transaction runs.
- **'time_to_execute': i**s the time at which the WAITFOR statement finishes. time_to_execute can be specified in one of the acceptable formats for datetime data, or it can be specified as a local variable. Dates cannot be specified; therefore, the date part of the datetime value is not allowed.

- The following example executes the stored procedure after a two-hour delay.
  - *BEGIN*
  - *WAITFOR DELAY '02:00';*
  - *EXECUTE sp_helpdb;*
  - *END;*
- The following example executes the stored procedure sp_dbremove to remove 'test' database at 10:20 P.M. (22:20).
  - *WAITFOR TIME '22:20';*
  - *EXECUTE sp_dbremove 'test',*

## 5. IF Statement

- You use the *IF . . . ELSE* element to control the flow of your code based on the result of a predicate. You specify a statement or statement block that is executed if the predicate is *TRUE*, and optionally a statement or statement block that is executed if the predicate is *FALSE* or *UNKNOWN*.
  - **IF Boolean_expression {sql_statement | statement_block}**
  - **[ELSE { sql_statement | statement_block } ]**
- The following example has a simple Boolean expression (1=1) that is true and, therefore, prints the first statement.
  - *IF 1 = 1 PRINT 'Boolean_expression is true.'*
  - *ELSE PRINT 'Boolean_expression is false.' ;*
- The following example executes a query as part of the Boolean expression. Because there are 10 records in the Product table that meet the WHERE clause, the first print statement will execute. Change > 5 to > 15 to see how the second part of the statement could execute.
  - *IF (SELECT COUNT(*) FROM Product) > 5*
  - *PRINT 'There are more than 5 Touring-3000 bicycles.'*
  - *ELSE PRINT 'There are 5 or less Touring-3000 bicycles.' ;*

- The following example shows how an IF … ELSE statement can be nested inside another. Set the @Number variable to 5, 50, and 500 to test each statement.
  - *DECLARE @Number int;*
  - *SET @Number = 50;*
  - *IF @Number > 100*
  - *PRINT 'The number is large.';*
  - *ELSE*
  - *BEGIN*
  - *IF @Number < 10*
  - *PRINT 'The number is small.';*
  - *ELSE*
  - *PRINT 'The number is medium.';*
  - *END ;*
- The comparison operators are:

| Operator | Meaning |
|---|---|
| = (Equals) | Equal to |
| > (Greater Than) | Greater than |
| < (Less Than) | Less than |
| >= (Greater Than or Equal To) | Greater than or equal to |
| <= (Less Than or Equal To) | Less than or equal to |
| <> (Not Equal To) | Not equal to |
| != (Not Equal To) | Not equal to (not ISO standard) |
| !< (Not Less Than) | Not less than (not ISO standard) |
| !> (Not Greater Than) | Not greater than (not ISO standard) |

- **Examples:**
  - *DECLARE @x1 int = 27;*
  - *SET @x1 += 2 ;*
  - *SELECT @x1 AS Added_2;*

- *DECLARE @x2 int = 27;*
- *SET @x2 -= 2 ;*
- *SELECT @x2 AS Subtracted_2;*

- *DECLARE @x3 int = 27;*
- *SET @x3 *= 2 ;*
- *SELECT @x3 AS Multiplied_by_2;*

- *DECLARE @x4 int = 27;*
- *SET @x4 /= 2 ;*
- *SELECT @x4 AS Divided_by_2;*

- *DECLARE @x5 int = 27;*
- *SET @x5 %= 2 ;*
- *SELECT @x5 AS Modulo_of_27_divided_by_2;*

- *DECLARE @x6 int = 9;*
- *SET @x6 &= 13 ;*
- *SELECT @x6 AS Bitwise_AND;*

- *DECLARE @x7 int = 27;*
- *SET @x7 ^= 2 ;*
- *SELECT @x7 AS Bitwise_Exclusive_OR;*

- *DECLARE @x8 int = 27;*
- *SET @x8 |= 2 ;*
- *SELECT @x8 AS Bitwise_OR;*

## 6. While Statement

- T-SQL provides the *WHILE* element, which you can use to execute code in a loop. The *WHILE* element executes a statement or statement block repeatedly while the predicate you specify after the *WHILE* keyword is *TRUE*. When the predicate is *FALSE* or *UNKNOWN*, the loop terminates. T-SQL doesn't provide a built-in looping element that executes a predetermined number of times, but it's easy to mimic such an element with a *WHILE* loop and a variable.
  - **WHILE Boolean_expression**
  - **{ sql_statement | statement_block | BREAK | 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 after the CONTINUE keyword.

- **Example 6.1**: the following code demonstrates how to write a loop that iterates 10 times:
  - *DECLARE @i AS INT = 1;*
  - *WHILE @i <= 10*
  - *BEGIN*
  - *PRINT @i;*
  - *SET @i = @i + 1;*
  - *END*

- **Example 6.2:** for example, the following code breaks from the loop if the value of @i is equal to 6:
  - *DECLARE @i AS INT = 1;*
  - *WHILE @i <= 10*
  - *BEGIN*

- *IF @i = 6 BREAK;*
- *PRINT @i;*
- *SET @i = @i + 1;*
- *END;*

- **Example 6.3:** if the average list price of a product is less than $300, the WHILE loop doubles the prices and then selects the maximum price. If the maximum price is less than or equal to $500, the WHILE loop restarts and doubles the prices again. This loop continues doubling the prices until the maximum price is greater than $500, and then exits the WHILE loop and prints a message.
  - *WHILE (SELECT AVG(ListPrice) FROM Product) < $300*
  - *BEGIN*
  - *UPDATE Product*
  - *SET ListPrice = ListPrice * 2*
  - *SELECT MAX(ListPrice) FROM Product*
  - *IF (SELECT MAX(ListPrice) FROM Product) > $500*
  - *BREAK*
  - *ELSE*
  - *CONTINUE*
  - *END*
  - *PRINT 'Too much for the market to bear';*

## 7. Stored Procedures

- A stored procedure in SQL Server is a group of one or more Transact-SQL statements or a reference to a Microsoft .NET Framework common runtime language (CLR) method. Procedures resemble constructs in other programming languages because they can:
  - Accept input parameters and return multiple values in the form of output parameters to the calling program.

- Contain programming statements that perform operations in the database. These include calling other procedures.
- Return a status value to a calling program to indicate success or failure (and the reason for failure).

## 8. Benefits of Stored Procedures

- *Reduced server/client network traffic*: the commands in a procedure are executed as a single batch of code. This can significantly reduce network traffic between the server and client because only the call to execute the procedure is sent across the network.
- *Stronger security:* when calling a procedure over the network, only the call to execute the procedure is visible. Therefore, malicious users cannot see table and database object names, embed Transact-SQL statements of their own, or search for critical data. Using procedure parameters helps guard against SQL injection attacks. Since parameter input is treated as a literal value and not as executable code, it is more difficult for an attacker to insert a command into the Transact-SQL statement(s) inside the procedure and compromise security.
- *Reuse of code:* the code for any repetitious database operation is the perfect candidate for encapsulation in procedures. This eliminates needless rewrites of the same code, decreases code inconsistency, and allows the code to be accessed and executed by any user or application possessing the necessary permissions.
- *Easier maintenance:* when client applications call procedures and keep database operations in the data tier, only the procedures must be updated for any changes in the underlying database. The application tier remains separate and does not have to know how about any changes to database layouts, relationships, or processes.
- *Improved performance:* by default, a procedure compiles the first time it is executed and creates an execution plan that is reused for subsequent executions. Since the query processor does not have to create a new plan, it typically takes less time to process the procedure. If there has been significant change to the tables or

data referenced by the procedure, the precompiled plan may actually cause the procedure to perform slower. In this case, recompiling the procedure and forcing a new execution plan can improve performance.

## 9. Creating Procedure

- To create procedures use the following T-SQL statement:
  - **CREATE { PROC | PROCEDURE } [ schema_name.] procedure_name**
    **[ { @parameterdata_type } [ OUT | OUTPUT ] ] [ ,...n ]**
    **AS { [ BEGIN ] sql_statement [;][ ,...n ] [ END ] }[;]**

- **Example 9.1**: design a procedure to print 'Hello World!'
  - *Create Proc dbo.HelloWorld*
  - *As*
  - *Print 'Hello World!'*
  - *Go*
  - *EXEC dbo.HelloWorld*

- **Example 9.2**: design a procedure to return the name of the current database.
  - *CREATE PROC What_DB_is_this*
  - *AS*
  - *SELECT DB_NAME() AS ThisDB;*
  - *EXEC What_DB_is_this;*

- **Example 9.3**: for the previous example, to provide an input parameter to make the procedure more flexible:
  - *CEATE PROCEDURE What_DB_is_that @ID int*
  - *AS*
  - *SELECT DB_NAME(@ID) AS ThatDB;*
  - *EXECUTE What_DB_is_that 1;*

- **Example 9.4**: creates a stored procedure that returns information for a specific employee by passing values for the employee's first name and last name. This procedure accepts only exact matches for the parameters passed.
  - *CREATE PROCEDURE test.employeeRes*
    *@LastName nvarchar(50), @FirstName nvarchar(50)*
  - *AS*
  - *SET NOCOUNT ON;*
  - *SELECT FirstName, LastName, JobTitle, Department*
    *FROM test.vEmployeeDepartment*
    *WHERE FirstName = @FirstName AND LastName = @LastName;*
    *GO*

- **Example 9.5**: design a stored procedure to return records from Address table in test database, but instead of getting back all records we will limit it to just a particular city:
  - *CREATE PROCEDURE dbo.GetAddress @City varchar(30)*
  - *AS*
  - *SELECT * FROM test.Address WHERE City = @City*
  - *GO*
  - *EXEC dbo.GetAddress 'Basrah'*

**10. Alter Procedure**

- To edit or alter stored procedures use the following T-SQL statement:
  - **ALTER { PROC | PROCEDURE } [schema_name.] procedure_name**
    **[ { @parameterdata_type } [= ] ] [ ,...n ]**
  - **AS { [ BEGIN ] sql_statement [ ; ] [ ,...n ] [ END ] } [;]**

- **Example 10.1**: edit the stored procedure (GetAddress) and make it remove the record with specific city name:
  - *Alter Procedure dbo.GetAddress @city varchar(30)*
  - *As*
  - *Delete from test.address where city=@city;*
  - *Go*

- **Example 10.2**: edit the stored procedure del which accept a parameter and make it without parameters and only print 'Hello World'
  - *Alter proc del*
  - *As*
  - *Print 'Hello World'*

## 11. Drop Procedure

- To drop a stored procedure use the following T-SQL statement:
  - **DROP { PROC | PROCEDURE } [ IF EXISTS ] { [ schema_name. ] procedure } [ ,...n ]**

- **Example 11.1**: the following example removes the dbo.uspMyProc stored procedure in the current database.
  - DROP PROCEDURE IF Exists dbo.uspMyProc;
  - GO

- **Example 11.2**: The following example removes several stored procedures in the current database.
  - *DROP PROCEDURE dbo.uspGetSalesbyMonth, dbo.uspUpdateSalesQuotes, dbo.uspGetSalesByYear;*

- **Example 11.3**: The following example drop the stored procedure using if exists:
  - *DROP PROCEDURE IF EXISTS dbo.uspMyProc;*
  - *GO*

## 12. Improve Procedure Performance

- There are many suggestions to improve procedure performance:
  - Use the SET NOCOUNT ON statement as the first statement in the body of the procedure. That is, place it just after the AS keyword. This turns off messages that SQL Server sends back to the client after any SELECT, INSERT, UPDATE, MERGE, and DELETE statements are executed. Overall performance of the database and application is improved by eliminating this unnecessary network overhead.
  - Use schema names when creating or referencing database objects in the procedure. It takes less processing time for the Database Engine to resolve object names if it does not have to search multiple schemas. It also prevents permission and access problems caused by a user's default schema being assigned when objects are created without specifying the schema.
  - Avoid the use of SELECT *. Instead, specify the required column names. This can prevent some Database Engine errors that stop procedure execution. For example, a SELECT * statement that returns data from a 12 column table and then inserts that data into a 12 column temporary table succeeds until the number or order of columns in either table is changed.
  - Avoid using scalar functions in SELECT statements that return many rows of data. Because the scalar function must be applied to every row, the resulting behavior is like row-based processing and degrades performance.