DECLARE (Transact-SQL)

07/24/2017 • 9 minutes to read • +2

In this article

APPLIES TO: SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Variables are declared in the body of a batch or procedure with the DECLARE statement and are assigned values by using either a SET or SELECT statement. Cursor variables can be declared with this statement and used with other cursor-related statements. After declaration, all variables are initialized as NULL, unless a value is provided as part of the declaration.

Transact-SQL Syntax Conventions

Syntax

Copy

```
-- Syntax for SQL Server and Azure SQL Database

DECLARE
{
    { @local_variable [AS] data_type  [ = value ] }
  | { @cursor_variable_name CURSOR }
} [,...n]
| { @table_variable_name [AS] <table_type_definition> }

<table_type_definition> ::=
     TABLE ( { <column_definition> | <table_constraint> } [ ,...n] )

<column_definition> ::=
     column_name { scalar_data_type | AS computed_column_expression }
     [ COLLATE collation_name ]
     [ [ DEFAULT constant_expression ] | IDENTITY [ (seed ,increment ) ] ]
     [ ROWGUIDCOL ]
     [ <column_constraint> ]
```

```
<column_constraint> ::=
     { [ NULL | NOT NULL ]
     | [ PRIMARY KEY | UNIQUE ]
     | CHECK ( logical_expression )
     | WITH ( <index_option > )
     }

<table_constraint> ::=
     { { PRIMARY KEY | UNIQUE } ( column_name [ ,...n] )
     | CHECK ( search_condition )
     }

<index_option> ::=
See CREATE TABLE for index option syntax.
```

Copy

```
-- Syntax for Azure SQL Data Warehouse and Parallel Data Warehouse

DECLARE
{{ @local_variable [AS] data_type } [ =value [ COLLATE <collation_name> ] ]
} [,...n]
```

Arguments

@local_variable
Is the name of a variable. Variable names must begin with an at (@) sign. Local variable names must comply with the rules for identifiers.

data_type
Is any system-supplied, common language runtime (CLR) user-defined table type, or alias data type. A variable cannot be of **text**, **ntext**, or **image** data type.

For more information about system data types, see Data Types (Transact-SQL). For more information about CLR user-defined types or alias data types, see CREATE TYPE (Transact-SQL).

=value
Assigns a value to the variable in-line. The value can be a constant or an expression, but it must either match the variable declaration type or be implicitly convertible to that type. For more information, see Expressions (Transact-SQL).

@cursor_variable_name
Is the name of a cursor variable. Cursor variable names must begin with an at (@) sign

and conform to the rules for identifiers.

CURSOR
Specifies that the variable is a local cursor variable.

@table_variable_name
Is the name of a variable of type **table**. Variable names must begin with an at (@) sign and conform to the rules for identifiers.

<table_type_definition>
Defines the **table** data type. The table declaration includes column definitions, names, data types, and constraints. The only constraint types allowed are PRIMARY KEY, UNIQUE, NULL, and CHECK. An alias data type cannot be used as a column scalar data type if a rule or default definition is bound to the type.

<table_type_definiton> Is a subset of information used to define a table in CREATE TABLE. Elements and essential definitions are included here. For more information, see CREATE TABLE (Transact-SQL).

n
Is a placeholder indicating that multiple variables can be specified and assigned values. When declaring **table** variables, the **table** variable must be the only variable being declared in the DECLARE statement.

column_name
Is the name of the column in the table.

scalar_data_type
Specifies that the column is a scalar data type.

computed_column_expression
Is an expression defining the value of a computed column. It is computed from an expression using other columns in the same table. For example, a computed column can have the definition **cost** AS **price * qty**. The expression can be a noncomputed column name, constant, built-in function, variable, or any combination of these connected by one or more operators. The expression cannot be a subquery or a user-defined function. The expression cannot reference a CLR user-defined type.

[COLLATE *collation_name*]
Specifies the collation for the column. *collation_name* can be either a Windows collation name or an SQL collation name, and is applicable only for columns of the **char**, **varchar**, **text**, **nchar**, **nvarchar**, and **ntext** data types. If not specified, the column is assigned either the collation of the user-defined data type (if the column is of a user-defined data type) or the collation of the current database.

For more information about the Windows and SQL collation names, see COLLATE (Transact-SQL).

DEFAULT
Specifies the value provided for the column when a value is not explicitly supplied during an insert. DEFAULT definitions can be applied to any columns except those defined as **timestamp** or those with the IDENTITY property. DEFAULT definitions are removed when the table is dropped. Only a constant value, such as a character string; a system function, such as a SYSTEM_USER(); or NULL can be used as a default. To maintain compatibility with earlier versions of SQL Server, a constraint name can be assigned to a DEFAULT.

constant_expression
Is a constant, NULL, or a system function used as the default value for the column.

IDENTITY
Indicates that the new column is an identity column. When a new row is added to the table, SQL Server provides a unique incremental value for the column. Identity columns are commonly used in conjunction with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. You must specify both the seed and increment, or neither. If neither is specified, the default is (1,1).

seed
Is the value used for the very first row loaded into the table.

increment
Is the incremental value added to the identity value of the previous row that was loaded.

ROWGUIDCOL
Indicates that the new column is a row global unique identifier column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. The ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column.

NULL | NOT NULL
Indicates if null is allowed in the variable. The default is NULL.

PRIMARY KEY
Is a constraint that enforces entity integrity for a given column or columns through a unique index. Only one PRIMARY KEY constraint can be created per table.

UNIQUE
Is a constraint that provides entity integrity for a given column or columns through a unique index. A table can have multiple UNIQUE constraints.

CHECK
Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns.

logical_expression
Is a logical expression that returns TRUE or FALSE.

Remarks

Variables are often used in a batch or procedure as counters for WHILE, LOOP, or for an IF...ELSE block.

Variables can be used only in expressions, not in place of object names or keywords. To construct dynamic SQL statements, use EXECUTE.

The scope of a local variable is the batch in which it is declared.

A table variable is not necessarily memory resident. Under memory pressure, the pages belonging to a table variable can be pushed out to tempdb.

A cursor variable that currently has a cursor assigned to it can be referenced as a source in a:

- CLOSE statement.
- DEALLOCATE statement.
- FETCH statement.
- OPEN statement.
- Positioned DELETE or UPDATE statement.
- SET CURSOR variable statement (on the right side).

In all of these statements, SQL Server raises an error if a referenced cursor variable exists but does not have a cursor currently allocated to it. If a referenced cursor variable does not exist, SQL Server raises the same error raised for an undeclared variable of another type.

A cursor variable:

- Can be the target of either a cursor type or another cursor variable. For more information, see SET @local_variable (Transact-SQL).
- Can be referenced as the target of an output cursor parameter in an EXECUTE statement if the cursor variable does not have a cursor currently assigned to it.
- Should be regarded as a pointer to the cursor.

Examples

A. Using DECLARE

The following example uses a local variable named `@find` to retrieve contact information for all last names beginning with `Man`.

Copy

```
USE AdventureWorks2012;
GO
DECLARE @find varchar(30);
/* Also allowed:
DECLARE @find varchar(30) = 'Man%';
*/
SET @find = 'Man%';
SELECT p.LastName, p.FirstName, ph.PhoneNumber
FROM Person.Person AS p
JOIN Person.PersonPhone AS ph ON p.BusinessEntityID = ph.BusinessEntityID
WHERE LastName LIKE @find;
```

Here is the result set.

Copy

```
LastName            FirstName               Phone
------------------- ----------------------- -------------------------
Manchepalli         Ajay                    1 (11) 500 555-0174
Manek               Parul                   1 (11) 500 555-0146
Manzanares          Tomas                   1 (11) 500 555-0178

(3 row(s) affected)
```

B. Using DECLARE with two variables

The following example retrieves the names of Adventure Works Cycles sales representatives who are located in the North American sales territory and have at least

$2,000,000 in sales for the year.

Copy

```
USE AdventureWorks2012;
GO
SET NOCOUNT ON;
GO
DECLARE @Group nvarchar(50), @Sales money;
SET @Group = N'North America';
SET @Sales = 2000000;
SET NOCOUNT OFF;
SELECT FirstName, LastName, SalesYTD
FROM Sales.vSalesPerson
WHERE TerritoryGroup = @Group and SalesYTD >= @Sales;
```

C. Declaring a variable of type table

The following example creates a `table` variable that stores the values specified in the OUTPUT clause of the UPDATE statement. Two `SELECT` statements follow that return the values in `@MyTableVar` and the results of the update operation in the `Employee` table. Note that the results in the `INSERTED.ModifiedDate` column differ from the values in the `ModifiedDate` column in the `Employee` table. This is because the `AFTER UPDATE` trigger, which updates the value of `ModifiedDate` to the current date, is defined on the `Employee` table. However, the columns returned from `OUTPUT` reflect the data before triggers are fired. For more information, see OUTPUT Clause (Transact-SQL).

Copy

```
USE AdventureWorks2012;
GO
DECLARE @MyTableVar table(
    EmpID int NOT NULL,
    OldVacationHours int,
    NewVacationHours int,
    ModifiedDate datetime);
UPDATE TOP (10) HumanResources.Employee
SET VacationHours = VacationHours * 1.25
OUTPUT INSERTED.BusinessEntityID,
       DELETED.VacationHours,
       INSERTED.VacationHours,
       INSERTED.ModifiedDate
INTO @MyTableVar;
--Display the result set of the table variable.
SELECT EmpID, OldVacationHours, NewVacationHours, ModifiedDate
FROM @MyTableVar;
GO
--Display the result set of the table.
--Note that ModifiedDate reflects the value generated by an
```

```
--AFTER UPDATE trigger.
SELECT TOP (10) BusinessEntityID, VacationHours, ModifiedDate
FROM HumanResources.Employee;
GO
```

D. Declaring a variable of user-defined table type

The following example creates a table-valued parameter or table variable called `@LocationTVP`. This requires a corresponding user-defined table type called `LocationTableType`. For more information about how to create a user-defined table type, see CREATE TYPE (Transact-SQL). For more information about table-valued parameters, see Use Table-Valued Parameters (Database Engine).

Copy

```
DECLARE @LocationTVP
AS LocationTableType;
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

E. Using DECLARE

The following example uses a local variable named `@find` to retrieve contact information for all last names beginning with `Walt`.

Copy

```
-- Uses AdventureWorks

DECLARE @find varchar(30);
/* Also allowed:
DECLARE @find varchar(30) = 'Man%';
*/
SET @find = 'Walt%';

SELECT LastName, FirstName, Phone
FROM DimEmployee
WHERE LastName LIKE @find;
```

F. Using DECLARE with two variables

The following example retrieves uses variables to specify the first and last names of employees in the `DimEmployee` table.

Copy

```
-- Uses AdventureWorks

DECLARE @lastName varchar(30), @firstName varchar(30);

SET @lastName = 'Walt%';
SET @firstName = 'Bryan';

SELECT LastName, FirstName, Phone
FROM DimEmployee
WHERE LastName LIKE @lastName AND FirstName LIKE @firstName;
```

See Also

EXECUTE (Transact-SQL)
Built-in Functions (Transact-SQL)
SELECT (Transact-SQL)
table (Transact-SQL)
Compare Typed XML to Untyped XML