UPDATE (T-SQL Statement)

Microsoft

TechNet

Table Of Contents

UPDATE (Transact-SQL)

SQL Server 2012

Changes existing data in a table or view in SQL Server 2012. For examples, see Examples.

Transact-SQL Syntax Conventions

Syntax

Transact-SQL

```
[ WITH <common_table_expression> [...n] ]
UPDATE
    [ TOP ( expression ) [ PERCENT ] ]
    { { table_alias | <object> | rowset_function_limited
         [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
      }
      | @table_variable
    }
    SET
        { column_name = { expression | DEFAULT | NULL }
          | { udt_column_name.{ { property_name = expression
                                | field_name = expression }
                                | method_name ( argument [ ,...n ] )
                              }
          }
          | column_name { .WRITE ( expression , @Offset , @Length ) }
          | @variable = expression
          | @variable = column = expression
          | column_name { += | -= | *= | /= | %= | &= | ^= | |= } expression
          | @variable { += | -= | *= | /= | %= | &= | ^= | |= } expression
          | @variable = column { += | -= | *= | /= | %= | &= | ^= | |= } expression
        } [ ,...n ]

    [ <OUTPUT Clause> ]
    [ FROM{ <table_source> } [ ,...n ] ]
    [ WHERE { <search_condition>
            | { [ CURRENT OF
                  { { [ GLOBAL ] cursor_name }
                      | cursor_variable_name
                  }
                ]
              }
            }
    ]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
[ ; ]

<object> ::=
{
    [ server_name . database_name . schema_name .
    | database_name .[ schema_name ] .
    | schema_name .
    ]
    table_or_view_name}
```

Arguments

WITH <common_table_expression>

Specifies the temporary named result set or view, also known as common table expression (CTE), defined within the scope of the UPDATE statement. The CTE result set is derived from a simple query and is referenced by UPDATE statement.

Common table expressions can also be used with the SELECT, INSERT, DELETE, and CREATE VIEW statements. For more information, see WITH common_table_expression (Transact-SQL).

TOP **(** *expression***)** [PERCENT]

Specifies the number or percent of rows that will be updated. *expression* can be either a number or a percent of the rows.

The rows referenced in the TOP expression used with INSERT, UPDATE, or DELETE are not arranged in any order.

Parentheses delimiting *expression* in TOP are required in INSERT, UPDATE, and DELETE statements. For more information, see TOP (Transact-SQL).

table_alias

The alias specified in the FROM clause representing the table or view from which the rows are to be updated.

server_name

Is the name of the server (using a linked server name or the OPENDATASOURCE function as the server name) on which the table or view is located. If *server_name* is specified, *database_name* and *schema_name* are required.

database_name

Is the name of the database.

schema_name
Is the name of the schema to which the table or view belongs.

table_or view_name
Is the name of the table or view from which the rows are to be updated. The view referenced by *table_or_view_name* must be updatable and reference exactly one base table in the FROM clause of the view. For more information about updatable views, see CREATE VIEW (Transact-SQL).

rowset_function_limited
Is either the OPENQUERY or OPENROWSET function, subject to provider capabilities.

WITH **(** <Table_Hint_Limited> **)**
Specifies one or more table hints that are allowed for a target table. The WITH keyword and the parentheses are required. NOLOCK and READUNCOMMITTED are not allowed. For information about table hints, see Table Hints (Transact-SQL).

@table_variable
Specifies a table variable as a table source.

SET
Specifies the list of column or variable names to be updated.

column_name
Is a column that contains the data to be changed. *column_name* must exist in *table_or view_name*. Identity columns cannot be updated.

expression
Is a variable, literal value, expression, or a subselect statement (enclosed with parentheses) that returns a single value. The value returned by *expression* replaces the existing value in *column_name* or *@variable*.

> **Note**
>
> When referencing the Unicode character data types **nchar**, **nvarchar**, and **ntext**, 'expression' should be prefixed with the capital letter 'N'. If 'N' is not specified, SQL Server converts the string to the code page that corresponds to the default collation of the database or column. Any characters not found in this code page are lost.

DEFAULT
Specifies that the default value defined for the column is to replace the existing value in the column. This can also be used to change the column to NULL if the column has no default and is defined to allow null values.

{ **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **^=** | **|=** }
Compound assignment operator:

+= Add and assign

-= Subtract and assign

*= Multiply and assign

/= Divide and assign

%= Modulo and assign

&= Bitwise AND and assign

^= Bitwise XOR and assign

|= Bitwise OR and assign

udt_column_name
Is a user-defined type column.

property_name | *field_name*
Is a public property or public data member of a user-defined type.

method_name **(** *argument* [**,**... *n*] **)**
Is a nonstatic public mutator method of *udt_column_name* that takes one or more arguments.

.WRITE **(***expression***,***@Offset***,***@Length***)**
Specifies that a section of the value of *column_name* is to be modified. *expression* replaces *@Length* units starting from *@Offset* of *column_name*. Only columns of **varchar(max)**, **nvarchar(max)**, or **varbinary(max)** can be specified with this clause. *column_name* cannot be NULL and cannot be qualified with a table name or table alias.

expression is the value that is copied to *column_name*. *expression* must evaluate to or be able to be implicitly cast to the *column_name* type. If *expression* is set to NULL, *@Length* is ignored, and the value in *column_name* is truncated at the specified *@Offset*.

@Offset is the starting point in the value of *column_name* at which *expression* is written. *@Offset* is a zero-based ordinal position, is **bigint**, and cannot be a negative number. If *@Offset* is NULL, the update operation appends *expression* at the end of the existing *column_name* value and *@Length* is ignored. If @Offset is greater than the length of the *column_name* value, the Database Engine returns an error. If *@Offset* plus *@Length* exceeds the end of the underlying value in the column, the deletion occurs up to the last character of the value. If *@Offset* plus LEN(*expression*) is greater than the underlying declared size, an error is raised.

@Length is the length of the section in the column, starting from *@Offset*, that is replaced by *expression*. *@Length* is **bigint** and cannot be a negative number. If *@Length* is NULL, the update operation removes all data from *@Offset* to the end of the *column_name* value.

For more information, see Remarks.

@ *variable*
Is a declared variable that is set to the value returned by *expression*.

SET **@***variable* = *column* = *expression* sets the variable to the same value as the column. This differs from SET **@***variable* = *column*, *column* = *expression*, which sets

the variable to the pre-update value of the column.

<OUTPUT_Clause>
Returns updated data or expressions based on it as part of the UPDATE operation. The OUTPUT clause is not supported in any DML statements that target remote tables or views. For more information, see OUTPUT Clause (Transact-SQL).

FROM <table_source>
Specifies that a table, view, or derived table source is used to provide the criteria for the update operation. For more information, see FROM (Transact-SQL).

If the object being updated is the same as the object in the FROM clause and there is only one reference to the object in the FROM clause, an object alias may or may not be specified. If the object being updated appears more than one time in the FROM clause, one, and only one, reference to the object must not specify a table alias. All other references to the object in the FROM clause must include an object alias.

A view with an INSTEAD OF UPDATE trigger cannot be a target of an UPDATE with a FROM clause.

> **Note**
>
> Any call to OPENDATASOURCE, OPENQUERY, or OPENROWSET in the FROM clause is evaluated separately and independently from any call to these functions used as the target of the update, even if identical arguments are supplied to the two calls. In particular, filter or join conditions applied on the result of one of those calls have no effect on the results of the other.

WHERE
Specifies the conditions that limit the rows that are updated. There are two forms of update based on which form of the WHERE clause is used:

- Searched updates specify a search condition to qualify the rows to delete.
- Positioned updates use the CURRENT OF clause to specify a cursor. The update operation occurs at the current position of the cursor.

<search_condition>
Specifies the condition to be met for the rows to be updated. The search condition can also be the condition upon which a join is based. There is no limit to the number of predicates that can be included in a search condition. For more information about predicates and search conditions, see Search Condition (Transact-SQL).

CURRENT OF
Specifies that the update is performed at the current position of the specified cursor.

A positioned update using a WHERE CURRENT OF clause updates the single row at the current position of the cursor. This can be more accurate than a searched update that uses a WHERE <search_condition> clause to qualify the rows to be updated. A searched update modifies multiple rows when the search condition does not uniquely identify a single row.

GLOBAL
Specifies that *cursor_name* refers to a global cursor.

cursor_name
Is the name of the open cursor from which the fetch should be made. If both a global and a local cursor with the name *cursor_name* exist, this argument refers to the global cursor if GLOBAL is specified; otherwise, it refers to the local cursor. The cursor must allow updates.

cursor_variable_name
Is the name of a cursor variable. *cursor_variable_name* must reference a cursor that allows updates.

OPTION **(** <query_hint> [**,**... *n*] **)**
Specifies that optimizer hints are used to customize the way the Database Engine processes the statement. For more information, see Query Hints (Transact-SQL).

Best Practices

Use the @@ROWCOUNT function to return the number of inserted rows to the client application. For more information, see @@ROWCOUNT (Transact-SQL).

Variable names can be used in UPDATE statements to show the old and new values affected, but this should be used only when the UPDATE statement affects a single record. If the UPDATE statement affects multiple records, to return the old and new values for each record, use the OUTPUT clause.

Use caution when specifying the FROM clause to provide the criteria for the update operation. The results of an UPDATE statement are undefined if the statement includes a FROM clause that is not specified in such a way that only one value is available for each column occurrence that is updated, that is if the UPDATE statement is not deterministic. For example, in the UPDATE statement in the following script, both rows in `Table1` meet the qualifications of the FROM clause in the UPDATE statement; but it is undefined which row from `Table1` is used to update the row in `Table2`.

Transact-SQL

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.Table1', 'U') IS NOT NULL
    DROP TABLE dbo.Table1;
GO
IF OBJECT_ID ('dbo.Table2', 'U') IS NOT NULL
    DROP TABLE dbo.Table2;
GO
CREATE TABLE dbo.Table1
    (ColA int NOT NULL, ColB decimal(10,3) NOT NULL);
GO
CREATE TABLE dbo.Table2
    (ColA int PRIMARY KEY NOT NULL, ColB decimal(10,3) NOT NULL);
GO
INSERT INTO dbo.Table1 VALUES(1, 10.0), (1, 20.0);
INSERT INTO dbo.Table2 VALUES(1, 0.0);
```

```
GO
UPDATE dbo.Table2
SET dbo.Table2.ColB = dbo.Table2.ColB + dbo.Table1.ColB
FROM dbo.Table2
    INNER JOIN dbo.Table1
    ON (dbo.Table2.ColA = dbo.Table1.ColA);
GO
SELECT ColA, ColB
FROM dbo.Table2;
```

The same problem can occur when the FROM and WHERE CURRENT OF clauses are combined. In the following example, both rows in `Table2` meet the qualifications of the `FROM` clause in the `UPDATE` statement. It is undefined which row from `Table2` is to be used to update the row in `Table1`.

Transact-SQL

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.Table1', 'U') IS NOT NULL
    DROP TABLE dbo.Table1;
GO
IF OBJECT_ID ('dbo.Table2', 'U') IS NOT NULL
    DROP TABLE dbo.Table2;
GO
CREATE TABLE dbo.Table1
    (c1 int PRIMARY KEY NOT NULL, c2 int NOT NULL);
GO
CREATE TABLE dbo.Table2
    (d1 int PRIMARY KEY NOT NULL, d2 int NOT NULL);
GO
INSERT INTO dbo.Table1 VALUES (1, 10);
INSERT INTO dbo.Table2 VALUES (1, 20), (2, 30);
GO
DECLARE abc CURSOR LOCAL FOR
    SELECT c1, c2
    FROM dbo.Table1;
OPEN abc;
FETCH abc;
UPDATE dbo.Table1
SET c2 = c2 + d2
FROM dbo.Table2
WHERE CURRENT OF abc;
GO
SELECT c1, c2 FROM dbo.Table1;
GO
```

Compatibility Support

Support for use of the READUNCOMMITTED and NOLOCK hints in the FROM clause that apply to the target table of an UPDATE or DELETE statement will be removed in a future version of SQL Server. Avoid using these hints in this context in new development work, and plan to modify applications that currently use them.

Data Types

All **char** and **nchar** columns are right-padded to the defined length.

If ANSI_PADDING is set to OFF, all trailing spaces are removed from data inserted into **varchar** and **nvarchar** columns, except in strings that contain only spaces. These strings are truncated to an empty string. If ANSI_PADDING is set to ON, trailing spaces are inserted. The Microsoft SQL Server ODBC driver and OLE DB Provider for SQL Server automatically set ANSI_PADDING ON for each connection. This can be configured in ODBC data sources or by setting connection attributes or properties. For more information, see SET ANSI_PADDING (Transact-SQL).

Updating text, ntext, and image Columns

Modifying a **text**, **ntext**, or **image** column with UPDATE initializes the column, assigns a valid text pointer to it, and allocates at least one data page, unless the column is being updated with NULL.

To replace or modify large blocks of **text**, **ntext**, or **image** data, use WRITETEXT or UPDATETEXT instead of the UPDATE statement.

If the UPDATE statement could change more than one row while updating both the clustering key and one or more **text**, **ntext**, or **image** columns, the partial update to these columns is executed as a full replacement of the values.

> **Important**
>
> The **ntext**, **text**, and **image** data types will be removed in a future version of Microsoft SQL Server. Avoid using these data types in new development work, and plan to modify applications that currently use them. Use nvarchar(max), varchar(max), and varbinary(max) instead.

Updating Large Value Data Types

Use the **.**WRITE (*expression***,** *@Offset***,** *@Length*) clause to perform a partial or full update of **varchar(max)**, **nvarchar(max)**, and **varbinary(max)** data types. For example, a partial update of a **varchar(max)** column might delete or modify only the first 200 characters of the column, whereas a full update would delete or modify all the data in the column. **.**WRITE updates that insert or append new data are minimally logged if the database recovery model is set to bulk-logged or simple. Minimal logging is not used when existing values are updated. For more information, see The Transaction Log (SQL Server).

The Database Engine converts a partial update to a full update when the UPDATE statement causes either of these actions:

- Changes a key column of the partitioned view or table.
- Modifies more than one row and also updates the key of a nonunique clustered index to a nonconstant value.

You cannot use the **.**WRITE clause to update a NULL column or set the value of *column_name* to NULL.

@Offset and *@Length* are specified in bytes for **varbinary** and **varchar** data types and in characters for the **nvarchar** data type. The appropriate offsets are computed for double-byte character set (DBCS) collations.

For best performance, we recommend that data be inserted or updated in chunk sizes that are multiples of 8040 bytes.

If the column modified by the **.**WRITE clause is referenced in an OUTPUT clause, the complete value of the column, either the before image in **deleted.***column_name* or the after image in **inserted.***column_name*, is returned to the specified column in the table variable. See example G that follows.

To achieve the same functionality of **.**WRITE with other character or binary data types, use the STUFF (Transact-SQL).

Updating User-defined Type Columns

Updating values in user-defined type columns can be accomplished in one of the following ways:

- Supplying a value in a SQL Server system data type, as long as the user-defined type supports implicit or explicit conversion from that type. The following example shows how to update a value in a column of user-defined type `Point`, by explicitly converting from a string.

```
UPDATE Cities
SET Location = CONVERT(Point, '12.3:46.2')
WHERE Name = 'Anchorage';
```

- Invoking a method, marked as a mutator, of the user-defined type, to perform the update. The following example invokes a mutator method of type `Point` named `SetXY`. This updates the state of the instance of the type.

```
UPDATE Cities
SET Location.SetXY(23.5, 23.5)
WHERE Name = 'Anchorage';
```

> **Note**
>
> SQL Server returns an error if a mutator method is invoked on a Transact-SQL null value, or if a new value produced by a mutator method is null.

- Modifying the value of a registered property or public data member of the user-defined type. The expression supplying the value must be implicitly convertible to the type of the property. The following example modifies the value of property `X` of user-defined type `Point`.

```
UPDATE Cities
SET Location.X = 23.5
WHERE Name = 'Anchorage';
```

  To modify different properties of the same user-defined type column, issue multiple UPDATE statements, or invoke a mutator method of the type.

Updating FILESTREAM Data

You can use the UPDATE statement to update a FILESTREAM field to a null value, empty value, or a relatively small amount of inline data. However, a large amount of data is more efficiently streamed into a file by using Win32 interfaces. When you update a FILESTREAM field, you modify the underlying BLOB data in the file system. When a FILESTREAM field is set to NULL, the BLOB data associated with the field is deleted. You cannot use .WRITE(), to perform partial updates to FILESTREAM data. For more information, see FILESTREAM (SQL Server).

Error Handling

If an update to a row violates a constraint or rule, violates the NULL setting for the column, or the new value is an incompatible data type, the statement is canceled, an error is returned, and no records are updated.

When an UPDATE statement encounters an arithmetic error (overflow, divide by zero, or a domain error) during expression evaluation, the update is not performed. The rest of the batch is not executed, and an error message is returned.

If an update to a column or columns participating in a clustered index causes the size of the clustered index and the row to exceed 8,060 bytes, the update fails and an error message is returned.

Interoperability

UPDATE statements are allowed in the body of user-defined functions only if the table being modified is a table variable.

When an INSTEAD OF trigger is defined on UPDATE actions against a table, the trigger is running instead of the UPDATE statement. Earlier versions of SQL Server only support AFTER triggers defined on UPDATE and other data modification statements. The FROM clause cannot be specified in an UPDATE statement that references, either directly or indirectly, a view with an INSTEAD OF trigger defined on it. For more information about INSTEAD OF triggers, see CREATE TRIGGER (Transact-SQL).

Limitations and Restrictions

The FROM clause cannot be specified in an UPDATE statement that references, either directly or indirectly, a view that has an INSTEAD OF trigger defined on it. For more information about INSTEAD OF triggers, see CREATE TRIGGER (Transact-SQL).

When a common table expression (CTE) is the target of an UPDATE statement, all references to the CTE in the statement must match. For example, if the CTE is assigned an alias in the FROM clause, the alias must be used for all other references to the CTE. Unambiguous CTE references are required because a CTE does not have an object ID, which SQL Server uses to recognize the implicit relationship between an object and its alias. Without this relationship, the query plan may produce unexpected join behavior and unintended query results. The following examples demonstrate correct and incorrect methods of specifying a CTE when the CTE is the target object of the update operation.

Transact-SQL

```
USE tempdb;
GO
-- UPDATE statement with CTE references that are correctly matched.
DECLARE @x TABLE (ID int, Value int);
DECLARE @y TABLE (ID int, Value int);
INSERT @x VALUES (1, 10), (2, 20);
INSERT @y VALUES (1, 100),(2, 200);

WITH cte AS (SELECT * FROM @x)
UPDATE x -- cte is referenced by the alias.
SET Value = y.Value
FROM cte AS x  -- cte is assigned an alias.
INNER JOIN @y AS y ON y.ID = x.ID;
SELECT * FROM @x;
GO
```

Here is the result set.

```
ID     Value

------ -----

1      100

2      200


(2 row(s) affected)
```

Transact-SQL

```
-- UPDATE statement with CTE references that are incorrectly matched.
USE tempdb;
GO
DECLARE @x TABLE (ID int, Value int);
DECLARE @y TABLE (ID int, Value int);
INSERT @x VALUES (1, 10), (2, 20);
INSERT @y VALUES (1, 100),(2, 200);

WITH cte AS (SELECT * FROM @x)
UPDATE cte   -- cte is not referenced by the alias.
SET Value = y.Value
FROM cte AS x  -- cte is assigned an alias.
INNER JOIN @y AS y ON y.ID = x.ID;
SELECT * FROM @x;
GO
```

Here is the result set.

```
ID     Value

------ -----

1      100

2      100


(2 row(s) affected)
```

Locking Behavior

An UPDATE statement always acquires an exclusive (X) lock on the table it modifies, and holds that lock until the transaction completes. With an exclusive lock, no other transactions can modify data. You can specify table hints to override this default behavior for the duration of the UPDATE statement by specifying another locking

method, however, we recommend that hints be used only as a last resort by experienced developers and database administrators. For more information, see Table Hints (Transact-SQL).

Logging Behavior

The UPDATE statement is logged; however, partial updates to large value data types using the **.**WRITE clause are minimally logged. For more information, see "Updating Large Value Data Types" in the earlier section "Data Types".

Security

Permissions

UPDATE permissions are required on the target table. SELECT permissions are also required for the table being updated if the UPDATE statement contains a WHERE clause, or if *expression* in the SET clause uses a column in the table.

UPDATE permissions default to members of the **sysadmin** fixed server role, the **db_owner** and **db_datawriter** fixed database roles, and the table owner. Members of the **sysadmin**, **db_owner**, and **db_securityadmin** roles, and the table owner can transfer permissions to other users.

Examples

Category	Featured syntax elements
Basic Syntax	UPDATE
Limiting the Rows that Are Updated	WHERE • TOP • WITH common table expression • WHERE CURRENT OF
Setting Column Values	computed values • compound operators • default values • subqueries
Specifying Target Objects Other than Standard Tables	views • table variables • table aliases
Updating Data Based on Data From Other Tables	FROM
Updating Rows in a Remote Table	linked server • OPENQUERY • OPENDATASOURCE
Updating Large Object Data Types	.WRITE • OPENROWSET
Updating User-defined Types	user-defined types
Overriding the Default Behavior of the Query Optimizer by Using Hints	table hints • query hints
Capturing the Results of the UPDATE Statement	OUTPUT clause
Using UPDATE in Other Statements	Stored Procedures • TRY...CATCH

Basic Syntax

Examples in this section demonstrate the basic functionality of the UPDATE statement using the minimum required syntax.

A. Using a simple UPDATE statement

The following example updates a single column for all rows in the `Person.Address` table.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Person.Address
SET ModifiedDate = GETDATE();
```

B. Updating multiple columns

The following example updates the values in the `Bonus`, `CommissionPct`, and `SalesQuota` columns for all rows in the `SalesPerson` table.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Sales.SalesPerson
SET Bonus = 6000, CommissionPct = .10, SalesQuota = NULL;
GO
```

Limiting the Rows that Are Updated

Examples in this section demonstrate ways that you can use to limit the number of rows affected by the UPDATE statement.

A. Using the WHERE clause

The following example uses the WHERE clause to specify which rows to update. The statement updates the value in the `Color` column of the `Production.Product` table for all rows that have an existing value of 'Red' in the `Color` column and have a value in the `Name` column that starts with 'Road-250'.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Production.Product
SET Color = N'Metallic Red'
WHERE Name LIKE N'Road-250%' AND Color = N'Red';
GO
```

B. Using the TOP clause

The following examples use the TOP clause to limit the number of rows that are modified in an UPDATE statement. When a TOP (*n*) clause is used with UPDATE, the update operation is performed on a random selection of '*n*' number of rows. The following example updates the `VacationHours` column by 25 percent for 10 random rows in the `Employee` table.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE TOP (10) HumanResources.Employee
SET VacationHours = VacationHours * 1.25 ;
GO
```

If you must use TOP to apply updates in a meaningful chronology, you must use TOP together with ORDER BY in a subselect statement. The following example updates the vacation hours of the 10 employees with the earliest hire dates.

Transact-SQL

```
UPDATE HumanResources.Employee
SET VacationHours = VacationHours + 8
FROM (SELECT TOP 10 BusinessEntityID FROM HumanResources.Employee
     ORDER BY HireDate ASC) AS th
WHERE HumanResources.Employee.BusinessEntityID = th.BusinessEntityID;
GO
```

C. Using the WITH common_table_expression clause

The following example updates the `PerAssemnblyQty` value for all parts and components that are used directly or indirectly to create the `ProductAssemblyID 800`. The common table expression returns a hierarchical list of parts that are used directly to build `ProductAssemblyID 800` and parts that are used to build those components, and so on. Only the rows returned by the common table expression are modified.

Transact-SQL

```
USE AdventureWorks2012;
GO
WITH Parts(AssemblyID, ComponentID, PerAssemblyQty, EndDate, ComponentLevel) AS
(
    SELECT b.ProductAssemblyID, b.ComponentID, b.PerAssemblyQty,
        b.EndDate, 0 AS ComponentLevel
    FROM Production.BillOfMaterials AS b
    WHERE b.ProductAssemblyID = 800
          AND b.EndDate IS NULL
    UNION ALL
    SELECT bom.ProductAssemblyID, bom.ComponentID, p.PerAssemblyQty,
        bom.EndDate, ComponentLevel + 1
    FROM Production.BillOfMaterials AS bom
        INNER JOIN Parts AS p
        ON bom.ProductAssemblyID = p.ComponentID
        AND bom.EndDate IS NULL
)
UPDATE Production.BillOfMaterials
SET PerAssemblyQty = c.PerAssemblyQty * 2
FROM Production.BillOfMaterials AS c
JOIN Parts AS d ON c.ProductAssemblyID = d.AssemblyID
WHERE d.ComponentLevel = 0;
```

D. Using the WHERE CURRENT OF clause

The following example uses the WHERE CURRENT OF clause to update only the row on which the cursor is positioned. When a cursor is based on a join, only the `table_name` specified in the UPDATE statement is modified. Other tables participating in the cursor are not affected.

Transact-SQL

```
USE AdventureWorks2012;
GO
DECLARE complex_cursor CURSOR FOR
    SELECT a.BusinessEntityID
    FROM HumanResources.EmployeePayHistory AS a
    WHERE RateChangeDate <>
          (SELECT MAX(RateChangeDate)
           FROM HumanResources.EmployeePayHistory AS b
           WHERE a.BusinessEntityID = b.BusinessEntityID) ;
OPEN complex_cursor;
```

```
FETCH FROM complex_cursor;
UPDATE HumanResources.EmployeePayHistory
SET PayFrequency = 2
WHERE CURRENT OF complex_cursor;
CLOSE complex_cursor;
DEALLOCATE complex_cursor;
GO
```

Setting Column Values

Examples in this section demonstrate updating columns by using computed values, subqueries, and DEFAULT values.

A. Specifying a computed value

The following examples uses computed values in an UPDATE statement. The example doubles the value in the `ListPrice` column for all rows in the `Product` table.

Transact-SQL

```
USE AdventureWorks2012 ;
GO
UPDATE Production.Product
SET ListPrice = ListPrice * 2;
GO
```

B. Specifying a compound operator

The following example uses the variable `@NewPrice` to increment the price of all red bicycles by taking the current price and adding 10 to it.

Transact-SQL

```
USE AdventureWorks2012;
GO
DECLARE @NewPrice int = 10;
UPDATE Production.Product
SET ListPrice += @NewPrice
WHERE Color = N'Red';
GO
```

The following example uses the compound operator += to append the data `' - tool malfunction'` to the existing value in the column `Name` for rows that have a `ScrapReasonID` between 10 and 12.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Production.ScrapReason
SET Name += ' - tool malfunction'
WHERE ScrapReasonID BETWEEN 10 and 12;
```

C. Specifying a subquery in the SET clause

The following example uses a subquery in the SET clause to determine the value that is used to update the column. The subquery must return only a scalar value (that is, a single value per row). The example modifies the `SalesYTD` column in the `SalesPerson` table to reflect the most recent sales recorded in the `SalesOrderHeader` table. The subquery aggregates the sales for each salesperson in the `UPDATE` statement.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Sales.SalesPerson
SET SalesYTD = SalesYTD +
    (SELECT SUM(so.SubTotal)
     FROM Sales.SalesOrderHeader AS so
     WHERE so.OrderDate = (SELECT MAX(OrderDate)
                           FROM Sales.SalesOrderHeader AS so2
                           WHERE so2.SalesPersonID = so.SalesPersonID)
     AND Sales.SalesPerson.BusinessEntityID = so.SalesPersonID
     GROUP BY so.SalesPersonID);
GO
```

D. Updating rows using DEFAULT values

The following example sets the `CostRate` column to its default value (0.00) for all rows that have a `CostRate` value greater than `20.00`.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Production.Location
SET CostRate = DEFAULT
WHERE CostRate > 20.00;
```

Specifying Target Objects Other Than Standard Tables

Examples in this section demonstrate how to update rows by specifying a view, table alias, or table variable.

A. Specifying a view as the target object

The following example updates rows in a table by specifying a view as the target object. The view definition references multiple tables, however, the UPDATE statement succeeds because it references columns from only one of the underlying tables. The UPDATE statement would fail if columns from both tables were specified. For more information, see Modify Data Through a View.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Person.vStateProvinceCountryRegion
SET CountryRegionName = 'United States of America'
WHERE CountryRegionName = 'United States';
```

B. Specifying a table alias as the target object

The follow example updates rows in the table `Production.ScrapReason`. The table alias assigned to `ScrapReason` in the FROM clause is specified as the target object in the UPDATE clause.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE sr
SET sr.Name += ' - tool malfunction'
FROM Production.ScrapReason AS sr
JOIN Production.WorkOrder AS wo
     ON sr.ScrapReasonID = wo.ScrapReasonID
     AND wo.ScrappedQty > 300;
```

C. Specifying a table variable as the target object

The following example updates rows in a table variable.

Transact-SQL

```
USE AdventureWorks2012;
GO
-- Create the table variable.
DECLARE @MyTableVar table(
    EmpID int NOT NULL,
    NewVacationHours int,
    ModifiedDate datetime);

-- Populate the table variable with employee ID values from HumanResources.Employee.
INSERT INTO @MyTableVar (EmpID)
    SELECT BusinessEntityID FROM HumanResources.Employee;

-- Update columns in the table variable.
UPDATE @MyTableVar
SET NewVacationHours = e.VacationHours + 20,
    ModifiedDate = GETDATE()
FROM HumanResources.Employee AS e
WHERE e.BusinessEntityID = EmpID;

-- Display the results of the UPDATE statement.
SELECT EmpID, NewVacationHours, ModifiedDate FROM @MyTableVar
ORDER BY EmpID;
GO
```

Updating Data Based on Data From Other Tables

Examples in this section demonstrate methods of updating rows from one table based on information in another table.

A. Using the UPDATE statement with information from another table

The following example modifies the `SalesYTD` column in the `SalesPerson` table to reflect the most recent sales recorded in the `SalesOrderHeader` table.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Sales.SalesPerson
SET SalesYTD = SalesYTD + SubTotal
FROM Sales.SalesPerson AS sp
JOIN Sales.SalesOrderHeader AS so
    ON sp.BusinessEntityID = so.SalesPersonID
    AND so.OrderDate = (SELECT MAX(OrderDate)
                        FROM Sales.SalesOrderHeader
                        WHERE SalesPersonID = sp.BusinessEntityID);
GO
```

The previous example assumes that only one sale is recorded for a specified salesperson on a specific date and that updates are current. If more than one sale for a specified salesperson can be recorded on the same day, the example shown does not work correctly. The example runs without error, but each `SalesYTD` value is updated with only one sale, regardless of how many sales actually occurred on that day. This is because a single UPDATE statement never updates the same row two times.

In the situation in which more than one sale for a specified salesperson can occur on the same day, all the sales for each sales person must be aggregated together within the `UPDATE` statement, as shown in the following example:

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Sales.SalesPerson
SET SalesYTD = SalesYTD +
    (SELECT SUM(so.SubTotal)
     FROM Sales.SalesOrderHeader AS so
     WHERE so.OrderDate = (SELECT MAX(OrderDate)
                           FROM Sales.SalesOrderHeader AS so2
                           WHERE so2.SalesPersonID = so.SalesPersonID)
     AND Sales.SalesPerson.BusinessEntityID = so.SalesPersonID
     GROUP BY so.SalesPersonID);
GO
```

Updating Rows in a Remote Table

Examples in this section demonstrate how to update rows in a remote target table by using a linked server or a rowset function to reference the remote table.

A. Updating data in a remote table by using a linked server

The following example updates a table on a remote server. The example begins by creating a link to the remote data source by using sp_addlinkedserver. The linked server name, `MyLinkServer`, is then specified as part of the four-part object name in the form server.catalog.schema.object. Note that you must specify a valid server name for `@datasrc`.

Transact-SQL

```
USE master;
GO
-- Create a link to the remote data source.
-- Specify a valid server name for @datasrc as 'server_name' or 'server_name\instance_name'.

EXEC sp_addlinkedserver @server = N'MyLinkServer',
    @srvproduct = N' ',
    @provider = N'SQLNCLI10',
    @datasrc = N'<server name>',
    @catalog = N'AdventureWorks2012';
GO
USE AdventureWorks2012;
GO
-- Specify the remote data source using a four-part name
-- in the form linked_server.catalog.schema.object.

UPDATE MyLinkServer.AdventureWorks2012.HumanResources.Department
SET GroupName = N'Public Relations'
WHERE DepartmentID = 4;
```

B. Updating data in a remote table by using the OPENQUERY function

The following example updates a row in a remote table by specifying the OPENQUERY rowset function. The linked server name created in the previous example is used in this example.

Transact-SQL

```
UPDATE OPENQUERY (MyLinkServer, 'SELECT GroupName FROM HumanResources.Department WHERE DepartmentID = 4')
SET GroupName = 'Sales and Marketing';
```

C. Updating data in a remote table by using the OPENDATASOURCE function

The following example inserts a row into a remote table by specifying the OPENDATASOURCE rowset function. Specify a valid server name for the data source by using the format *server_name* or *server_name\instance_name*. You may need to configure the instance of SQL Server for Ad Hoc Distributed Queries. For more information, see ad hoc distributed queries Server Configuration Option.

Transact-SQL

```
UPDATE OPENQUERY (MyLinkServer, 'SELECT GroupName FROM HumanResources.Department WHERE DepartmentID = 4')
SET GroupName = 'Sales and Marketing';
```

Updating Large Object Data Types

Examples in this section demonstrate methods of updating values in columns that are defined with large object (LOB) data types.

A. Using UPDATE with .WRITE to modify data in an nvarchar(max) column

The following example uses the .WRITE clause to update a partial value in `DocumentSummary`, an **nvarchar(max)** column in the `Production.Document` table. The word `components` is replaced with the word `features` by specifying the replacement word, the starting location (offset) of the word to be replaced in the existing data, and the number of characters to be replaced (length). The example also uses the OUTPUT clause to return the before and after images of the `DocumentSummary` column to the `@MyTableVar` table variable.

Transact-SQL

```
USE AdventureWorks2012;
GO
DECLARE @MyTableVar table (
```

```
    SummaryBefore nvarchar(max),
    SummaryAfter nvarchar(max));
UPDATE Production.Document
SET DocumentSummary .WRITE (N'features',28,10)
OUTPUT deleted.DocumentSummary,
       inserted.DocumentSummary
    INTO @MyTableVar
WHERE Title = N'Front Reflector Bracket Installation';
SELECT SummaryBefore, SummaryAfter
FROM @MyTableVar;
GO
```

B. Using UPDATE with .WRITE to add and remove data in an nvarchar(max) column

The following examples add and remove data from an **nvarchar(max)** column that has a value currently set to NULL. Because the .WRITE clause cannot be used to modify a NULL column, the column is first populated with temporary data. This data is then replaced with the correct data by using the .WRITE clause. The additional examples append data to the end of the column value, remove (truncate) data from the column and, finally, remove partial data from the column. The SELECT statements display the data modification generated by each UPDATE statement.

Transact-SQL

```
USE AdventureWorks2012;
GO
-- Replacing NULL value with temporary data.
UPDATE Production.Document
SET DocumentSummary = N'Replacing NULL value'
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
SELECT DocumentSummary
FROM Production.Document
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
-- Replacing temporary data with the correct data. Setting @Length to NULL
-- truncates all existing data from the @Offset position.
UPDATE Production.Document
SET DocumentSummary .WRITE(N'Carefully inspect and maintain the tires and crank arms.',0,NULL)
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
SELECT DocumentSummary
FROM Production.Document
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
-- Appending additional data to the end of the column by setting
-- @Offset to NULL.
UPDATE Production.Document
SET DocumentSummary .WRITE (N' Appending data to the end of the column.', NULL, 0)
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
SELECT DocumentSummary
FROM Production.Document
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
-- Removing all data from @Offset to the end of the existing value by
-- setting expression to NULL.
UPDATE Production.Document
SET DocumentSummary .WRITE (NULL, 56, 0)
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
SELECT DocumentSummary
FROM Production.Document
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
-- Removing partial data beginning at position 9 and ending at
-- position 21.
UPDATE Production.Document
SET DocumentSummary .WRITE ('',9, 12)
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
SELECT DocumentSummary
FROM Production.Document
WHERE Title = N'Crank Arm and Tire Maintenance';
GO
```

C. Using UPDATE with OPENROWSET to modify a varbinary(max) column

The following example replaces an existing image stored in a **varbinary(max)** column with a new image. The OPENROWSET function is used with the BULK option to load the image into the column. This example assumes that a file named `Tires.jpg` exists in the specified file path.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Production.ProductPhoto
SET ThumbNailPhoto = (
    SELECT *
    FROM OPENROWSET(BULK 'c:\Tires.jpg', SINGLE_BLOB) AS x )
WHERE ProductPhotoID = 1;
GO
```

D. Using UPDATE to modify FILESTREAM data

The following example uses the UPDATE statement to modify the data in the file system file. We do not recommend this method for streaming large amounts of data to a file. Use the appropriate Win32 interfaces. The following example replaces any text in the file record with the text `Xray 1`. For more information, see FILESTREAM (SQL Server).

Transact-SQL

```
UPDATE Archive.dbo.Records
SET [Chart] = CAST('Xray 1' as varbinary(max))
WHERE [SerialNumber] = 2;
```

Updating User-defined Types

The following examples modify values in CLR user-defined type (UDT) columns. Three methods are demonstrated. For more information about user-defined columns, see CLR User-Defined Types.

A. Using a system data type

You can update a UDT by supplying a value in a SQL Server system data type, as long as the user-defined type supports implicit or explicit conversion from that type. The following example shows how to update a value in a column of user-defined type `Point`, by explicitly converting from a string.

```
UPDATE dbo.Cities
SET Location = CONVERT(Point, '12.3:46.2')
WHERE Name = 'Anchorage';
```

B. Invoking a method

You can update a UDT by invoking a method, marked as a mutator, of the user-defined type, to perform the update. The following example invokes a mutator method of type `Point` named `SetXY`. This updates the state of the instance of the type.

```
UPDATE dbo.Cities
SET Location.SetXY(23.5, 23.5)
WHERE Name = 'Anchorage';
```

C. Modifying the value of a property or data member

You can update a UDT by modifying the value of a registered property or public data member of the user-defined type. The expression supplying the value must be implicitly convertible to the type of the property. The following example modifies the value of property `X` of user-defined type `Point`.

```
UPDATE dbo.Cities
SET Location.X = 23.5
WHERE Name = 'Anchorage';
```

Overriding the Default Behavior of the Query Optimizer by Using Hints

Examples in this section demonstrate how to use table and query hints to temporarily override the default behavior of the query optimizer when processing the UPDATE statement.

Caution

Because the SQL Server query optimizer typically selects the best execution plan for a query, we recommend that hints be used only as a last resort by experienced developers and database administrators.

A. Specifying a table hint

The following example specifies the table hint TABLOCK. This hint specifies that a shared lock is taken on the table `Production.Product` and held until the end of the UPDATE statement.

Transact-SQL

```
USE AdventureWorks2012;
GO
UPDATE Production.Product
WITH (TABLOCK)
SET ListPrice = ListPrice * 1.10
WHERE ProductNumber LIKE 'BK-%';
GO
```

B. Specifying a query hint

The following example specifies the query hint `OPTIMIZE FOR (@variable)` in the UPDATE statement. This hint instructs the query optimizer to use a particular value for a local variable when the query is compiled and optimized. The value is used only during query optimization, and not during query execution.

Transact-SQL

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE Production.uspProductUpdate
```

```
@Product nvarchar(25)
AS
SET NOCOUNT ON;
UPDATE Production.Product
SET ListPrice = ListPrice * 1.10
WHERE ProductNumber LIKE @Product
OPTION (OPTIMIZE FOR (@Product = 'BK-%') );
GO
-- Execute the stored procedure
EXEC Production.uspProductUpdate 'BK-%';
```

Capturing the Results of the UPDATE Statement

Examples in this section demonstrate how to use the OUTPUT Clause to return information from, or expressions based on, each row affected by an UPDATE statement. These results can be returned to the processing application for use in such things as confirmation messages, archiving, and other such application requirements.

A. Using UPDATE with the OUTPUT clause

The following example updates the column `VacationHours` in the `Employee` table by 25 percent for the first 10 rows and also sets the value in the column `ModifiedDate` to the current date. The `OUTPUT` clause returns the value of `VacationHours` that exists before applying the `UPDATE` statement in the `deleted.VacationHours` column and the updated value in the `inserted.VacationHours` column to the `@MyTableVar` table variable.

Two `SELECT` statements follow that return the values in `@MyTableVar` and the results of the update operation in the `Employee` table. For more examples using the OUTPUT clause, see OUTPUT Clause (Transact-SQL).

Transact-SQL

```
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,
    ModifiedDate = GETDATE()
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.
SELECT TOP (10) BusinessEntityID, VacationHours, ModifiedDate
FROM HumanResources.Employee;
GO
```

Using UPDATE in other statements

Examples in this section demonstrate how to use UPDATE in other statements.

A. Using UPDATE in a stored procedure

The following example uses an UPDATE statement in a stored procedure. The procedure takes one input parameter, `@NewHours` and one output parameter `@RowCount`. The `@NewHours` parameter value is used in the UPDATE statement to update the column `VacationHours` in the table `HumanResources.Employee`. The `@RowCount` output parameter is used to return the number of rows affected to a local variable. The CASE expression is used in the SET clause to conditionally determine the value that is set for `VacationHours`. When the employee is paid hourly (`SalariedFlag` = 0), `VacationHours` is set to the current number of hours plus the value specified in `@NewHours`; otherwise, `VacationHours` is set to the value specified in `@NewHours`.

Transact-SQL

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE HumanResources.Update_VacationHours
@NewHours smallint
AS
SET NOCOUNT ON;
UPDATE HumanResources.Employee
SET VacationHours =
    ( CASE
         WHEN SalariedFlag = 0 THEN VacationHours + @NewHours
         ELSE @NewHours
       END
    )
WHERE CurrentFlag = 1;
GO

EXEC HumanResources.Update_VacationHours 40;
```

B. Using UPDATE in a TRY...CATCH Block

The following example uses an UPDATE statement in a TRY...CATCH block to handle execution errors that may occur during the an update operation.

Transact-SQL

```
USE AdventureWorks2012;
GO
BEGIN TRANSACTION;

BEGIN TRY
    -- Intentionally generate a constraint violation error.
    UPDATE HumanResources.Department
    SET Name = N'MyNewName'
    WHERE DepartmentID BETWEEN 1 AND 2;
END TRY
BEGIN CATCH
    SELECT
         ERROR_NUMBER() AS ErrorNumber
        ,ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_STATE() AS ErrorState
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_LINE() AS ErrorLine
        ,ERROR_MESSAGE() AS ErrorMessage;

    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION;
END CATCH;

IF @@TRANCOUNT > 0
    COMMIT TRANSACTION;
GO
```

See Also

Reference

CREATE TABLE (Transact-SQL)
CREATE TRIGGER (Transact-SQL)
Cursors (Transact-SQL)
DELETE (Transact-SQL)
INSERT (Transact-SQL)
Text and Image Functions (Transact-SQL)
WITH common_table_expression (Transact-SQL)

Concepts

FILESTREAM (SQL Server)