DELETE (T-SQL Statement)

Microsoft

TechNet

Table Of Contents

DELETE (Transact-SQL)

SQL Server 2012

Removes one or more rows from a table or view in SQL Server 2012.

Transact-SQL Syntax Conventions

Syntax

Transact-SQL

```
[ WITH <common_table_expression> [ ,...n ] ]
DELETE
    [ TOP ( expression ) [ PERCENT ] ]
    [ FROM ]
    { { table_alias
      | <object>
      | rowset_function_limited
      [ WITH ( table_hint_limited [ ...n ] ) ] }
      | @table_variable
    }
    [ <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, also known as common table expression, defined within the scope of the DELETE statement. The result set is derived from a SELECT statement.

Common table expressions can also be used with the SELECT, INSERT, UPDATE, and CREATE VIEW statements. For more information, see WITH Common.

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

Specifies the number or percent of random rows that will be deleted. *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. For more information, see TOP (Transact-SQL).

FROM

An optional keyword that can be used between the DELETE keyword and the target *table_or_view_name*, or *rowset_function_limited*.

table_alias

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

server_name

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

The name of the database.

schema_name

The name of the schema to which the table or view belongs.

table_or view_name

The name of the table or view from which the rows are to be removed.

A table variable, within its scope, also can be used as a table source in a DELETE statement.

The view referenced by *table_or_view_name* must be updatable and reference exactly one base table in the FROM clause of the view definition. For more information about updatable views, see CREATE VIEW (Transact-SQL).

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

WITH **(** <table_hint_limited> [... *n*] **)**
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 more information about table hints, see Table Hints (Transact-SQL).

<OUTPUT_Clause>
Returns deleted rows, or expressions based on them, as part of the DELETE operation. The OUTPUT clause is not supported in any DML statements targeting views or remote tables. For more information, see OUTPUT Clause (Transact-SQL).

FROM *table_source*
Specifies an additional FROM clause. This Transact-SQL extension to DELETE allows specifying data from <table_source> and deleting the corresponding rows from the table in the first FROM clause.

This extension, specifying a join, can be used instead of a subquery in the WHERE clause to identify rows to be removed.

For more information, see FROM (Transact-SQL).

WHERE
Specifies the conditions used to limit the number of rows that are deleted. If a WHERE clause is not supplied, DELETE removes all the rows from the table.

There are two forms of delete operations based on what is specified in the WHERE clause:

- Searched deletes specify a search condition to qualify the rows to delete. For example, WHERE *column_name* = *value*.
- Positioned deletes use the CURRENT OF clause to specify a cursor. The delete operation occurs at the current position of the cursor. This can be more accurate than a searched DELETE statement that uses a WHERE *search_condition* clause to qualify the rows to be deleted. A searched DELETE statement deletes multiple rows if the search condition does not uniquely identify a single row.

<search_condition>
Specifies the restricting conditions for the rows to be deleted. There is no limit to the number of predicates that can be included in a search condition. For more information, see Search Condition (Transact-SQL).

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

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

cursor_name
Is the name of the open cursor from which the fetch is 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
The name of a cursor variable. The cursor variable must reference a cursor that allows updates.

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

Best Practices

To delete all the rows in a table, use TRUNCATE TABLE. TRUNCATE TABLE is faster than DELETE and uses fewer system and transaction log resources.

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

Error Handling

You can implement error handling for the DELETE statement by specifying the statement in a TRY…CATCH construct.

The DELETE statement may fail if it violates a trigger or tries to remove a row referenced by data in another table with a FOREIGN KEY constraint. If the DELETE removes multiple rows, and any one of the removed rows violates a trigger or constraint, the statement is canceled, an error is returned, and no rows are removed.

When a DELETE statement encounters an arithmetic error (overflow, divide by zero, or a domain error) occurring during expression evaluation, the Database Engine handles these errors as if SET ARITHABORT is set ON. The rest of the batch is canceled, and an error message is returned.

Interoperability

DELETE can be used in the body of a user-defined function if the object modified is a table variable.

When you delete a row that contains a FILESTREAM column, you also delete its underlying file system files. The underlying files are removed by the FILESTREAM garbage collector. For more information, see Access FILESTREAM Data with Transact-SQL.

The FROM clause cannot be specified in a DELETE 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

When TOP is used with DELETE, the referenced rows are not arranged in any order and the ORDER BY clause can not be directly specified in this statement. If you need to use TOP to delete rows in a meaningful chronological order, you must use TOP together with an ORDER BY clause in a subselect statement. See the Examples section that follows in this topic.

TOP cannot be used in a DELETE statement against partitioned views.

Locking Behavior

By default, a DELETE statement always acquires an exclusive (X) lock on the table it modifies, and holds that lock until the transaction completes. With an exclusive (X) lock, no other transactions can modify data; read operations can take place only with the use of the NOLOCK hint or read uncommitted isolation level. You can specify table hints to override this default behavior for the duration of the DELETE 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).

When rows are deleted from a heap the Database Engine may use row or page locking for the operation. As a result, the pages made empty by the delete operation remain allocated to the heap. When empty pages are not deallocated, the associated space cannot be reused by other objects in the database.

To delete rows in a heap and deallocate pages, use one of the following methods.

- Specify the TABLOCK hint in the DELETE statement. Using the TABLOCK hint causes the delete operation to take an exclusive lock on the table instead of a row or page lock. This allows the pages to be deallocated. For more information about the TABLOCK hint, see Table Hints (Transact-SQL).
- Use TRUNCATE TABLE if all rows are to be deleted from the table.
- Create a clustered index on the heap before deleting the rows. You can drop the clustered index after the rows are deleted. This method is more time consuming than the previous methods and uses more temporary resources.

Logging Behavior

The DELETE statement is always fully logged.

Security

Permissions

DELETE permissions are required on the target table. SELECT permissions are also required if the statement contains a WHERE clause.

DELETE 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 the **db_securityadmin** roles, and the table owner can transfer permissions to other users.

Examples

Category	Featured syntax elements
Basic syntax	DELETE
Limiting the rows deleted	WHERE • FROM • cursor •
Deleting rows from a remote table	Linked server • OPENQUERY rowset function • OPENDATASOURCE rowset function
Overriding the default behavior of the query optimizer by using hints	Table hints • query hints
Capturing the results of the DELETE statement	OUTPUT clause

Basic Syntax

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

A. Using DELETE with no WHERE clause

The following example deletes all rows from the `SalesPersonQuotaHistory` table because a WHERE clause is not used to limit the number of rows deleted.

Transact-SQL

```
USE AdventureWorks2012;
GO
DELETE FROM Sales.SalesPersonQuotaHistory;
GO
```

Limiting the Rows Deleted

Examples in this section demonstrate how to limit the number of rows that will be deleted.

A. Using the WHERE clause to delete a set of rows

The following example deletes all rows from the ProductCostHistory table in which the value in the StandardCost column is more than 1000.00.

Transact-SQL

```
USE AdventureWorks2012;
GO
DELETE FROM Production.ProductCostHistory
WHERE StandardCost > 1000.00;
GO
```

The following example shows a more complex WHERE clause. The WHERE clause defines two conditions that must be met to determine the rows to delete. The value in the StandardCost column must be between 12.00 and 14.00 and the value in the column SellEndDate must be null. The example also prints the value from the **@@ROWCOUNT** function to return the number of deleted rows.

Transact-SQL

```
USE AdventureWorks2012;
GO
DELETE Production.ProductCostHistory
WHERE StandardCost BETWEEN 12.00 AND 14.00
      AND EndDate IS NULL;
PRINT 'Number of rows deleted is ' + CAST(@@ROWCOUNT as char(3));
```

B. Using a cursor to determine the row to delete

The following example deletes a single row from the EmployeePayHistory table using a cursor named my_cursor. The delete operation affects only the single row currently fetched from the cursor.

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;
DELETE FROM HumanResources.EmployeePayHistory
WHERE CURRENT OF complex_cursor;
CLOSE complex_cursor;
DEALLOCATE complex_cursor;
GO
```

C. Using joins and subqueries to data in one table to delete rows in another table

The following examples show two ways to delete rows in one table based on data in another table. In both examples, rows from the SalesPersonQuotaHistory table based are deleted based on the year-to-date sales stored in the SalesPerson table. The first DELETE statement shows the ISO-compatible subquery solution, and the second DELETE statement shows the Transact-SQL FROM extension to join the two tables.

Transact-SQL

```
-- SQL-2003 Standard subquery

USE AdventureWorks2012;
GO
DELETE FROM Sales.SalesPersonQuotaHistory
WHERE BusinessEntityID IN
    (SELECT BusinessEntityID
     FROM Sales.SalesPerson
     WHERE SalesYTD > 2500000.00);
GO
```

Transact-SQL

```
-- Transact-SQL extension
USE AdventureWorks2012;
GO
DELETE FROM Sales.SalesPersonQuotaHistory
FROM Sales.SalesPersonQuotaHistory AS spqh
INNER JOIN Sales.SalesPerson AS sp
ON spqh.BusinessEntityID = sp.BusinessEntityID
WHERE sp.SalesYTD > 2500000.00;
GO
```

A. Using TOP to limit the number of rows deleted

When a TOP (*n*) clause is used with DELETE, the delete operation is performed on a random selection of *n* number of rows. The following example deletes 20 random rows from the PurchaseOrderDetail table that have due dates that are earlier than July 1, 2006.

Transact-SQL

```
USE AdventureWorks2012;
GO
DELETE TOP (20)
FROM Purchasing.PurchaseOrderDetail
WHERE DueDate < '20020701';
GO
```

If you have to use TOP to delete rows in a meaningful chronological order, you must use TOP together with ORDER BY in a subselect statement. The following query deletes the 10 rows of the `PurchaseOrderDetail` table that have the earliest due dates. To ensure that only 10 rows are deleted, the column specified in the subselect statement (`PurchaseOrderID`) is the primary key of the table. Using a nonkey column in the subselect statement may result in the deletion of more than 10 rows if the specified column contains duplicate values.

Transact-SQL

```
USE AdventureWorks2012;
GO
DELETE FROM Purchasing.PurchaseOrderDetail
WHERE PurchaseOrderDetailID IN
   (SELECT TOP 10 PurchaseOrderDetailID
    FROM Purchasing.PurchaseOrderDetail
    ORDER BY DueDate ASC);
GO
```

Deleting Rows From a Remote Table

Examples in this section demonstrate how to delete rows from a remote table by using a linked server or a rowset function to reference the remote table. A remote table exists on a different server or instance of SQL Server.

A. Deleting data from a remote table by using a linked server

The following example deletes rows from a remote table. 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*.

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'SQLNCLI',
    @datasrc = N'server_name',
    @catalog = N'AdventureWorks2012';
GO
```

Transact-SQL

```
-- Specify the remote data source using a four-part name
-- in the form linked_server.catalog.schema.object.

DELETE MyLinkServer.AdventureWorks2012.HumanResources.Department WHERE DepartmentID > 16;
GO
```

B. Deleting data from a remote table by using the OPENQUERY function

The following example deletes rows from 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

```
DELETE OPENQUERY (MyLinkServer, 'SELECT Name, GroupName FROM AdventureWorks2012.HumanResources.Department
WHERE DepartmentID = 18');
GO
```

C. Deleting data from a remote table by using the OPENDATASOURCE function

The following example deletes rows from 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*.

Transact-SQL

```
DELETE FROM OPENDATASOURCE('SQLNCLI',
    'Data Source= <server_name>; Integrated Security=SSPI')
    .AdventureWorks2012.HumanResources.Department
WHERE DepartmentID = 17;'
```

Capturing the results of the DELETE statement

A. Using DELETE with the OUTPUT clause

The following example shows how to save the results of a `DELETE` statement to a table variable.

Transact-SQL

```
USE AdventureWorks2012;
GO
DELETE Sales.ShoppingCartItem
OUTPUT DELETED.*
WHERE ShoppingCartID = 20621;

--Verify the rows in the table matching the WHERE clause have been deleted.
SELECT COUNT(*) AS [Rows in Table] FROM Sales.ShoppingCartItem WHERE ShoppingCartID = 20621;
GO
```

B. Using OUTPUT with <from_table_name> in a DELETE statement

The following example deletes rows in the `ProductProductPhoto` table based on search criteria defined in the `FROM` clause of the `DELETE` statement. The `OUTPUT` clause returns columns from the table being deleted, `DELETED.ProductID`, `DELETED.ProductPhotoID`, and columns from the `Product` table. This is used in the `FROM` clause to specify the rows to delete.

Transact-SQL

```
USE AdventureWorks2012;
GO
DECLARE @MyTableVar table (
    ProductID int NOT NULL,
    ProductName nvarchar(50)NOT NULL,
    ProductModelID int NOT NULL,
    PhotoID int NOT NULL);

DELETE Production.ProductProductPhoto
OUTPUT DELETED.ProductID,
       p.Name,
       p.ProductModelID,
       DELETED.ProductPhotoID
    INTO @MyTableVar
FROM Production.ProductProductPhoto AS ph
JOIN Production.Product as p
    ON ph.ProductID = p.ProductID
    WHERE p.ProductModelID BETWEEN 120 and 130;

--Display the results of the table variable.
SELECT ProductID, ProductName, ProductModelID, PhotoID
FROM @MyTableVar
ORDER BY ProductModelID;
GO
```

See Also

Reference

CREATE TRIGGER (Transact-SQL)
INSERT (Transact-SQL)
SELECT (Transact-SQL)
TRUNCATE TABLE (Transact-SQL)
UPDATE (Transact-SQL)
WITH common_table_expression (Transact-SQL)
@@ROWCOUNT (Transact-SQL)