SQL Server 2012 Transact-SQL DML Reference

SQL Server 2012 Books Online

Reference

Microsoft

Microsoft SQL Server 2012 Transact-SQL DML Reference

SQL Server Books Online

Summary: Data Manipulation Language (DML) is a vocabulary used to retrieve and work with data in SQL Server 2012. Use these statements to add, modify, query, or remove data from a SQL Server database.

Category: Reference
Applies to: SQL Server 2012
Source: SQL Server 2012 Books Online(link to source content)
E-book publication date: May 2012
263 pages

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Contents

Data Manipulation Language (DML) Statements

Data Manipulation Language (DML) is a vocabulary used to retrieve and work with data in SQL Server 2012. Use these statements to add, modify, query, or remove data from a SQL Server database.

In This Section

The following table lists the DML statements that SQL Server uses.

BULK INSERT (Transact-SQL)	SELECT (Transact-SQL)
DELETE (Transact-SQL)	UPDATE (Transact-SQL)
INSERT (Transact-SQL)	UPDATETEXT (Transact-SQL)
MERGE (Transact-SQL)	WRITETEXT (Transact-SQL)
READTEXT (Transact-SQL)	

The following table lists the clauses that are used in multiple DML statements or clauses.

Clause	Can be used in these statements
FROM (Transact-SQL)	DELETE, SELECT, UPDATE
Hints (Transact-SQL)	DELETE, INSERT, SELECT, UPDATE
OPTION Clause (Transact-SQL)	DELETE, SELECT, UPDATE
OUTPUT Clause (Transact-SQL)	DELETE, INSERT, MERGE, UPDATE
Search Condition (Transact-SQL)	DELETE, MERGE, SELECT, UPDATE
Table Value Constructor (Transact-SQL)	FROM, INSERT, MERGE
TOP (Transact-SQL)	DELETE, INSERT, MERGE, SELECT, UPDATE
WHERE (Transact-SQL)	DELETE, SELECT, UPDATE
WITH common_table_expression (Transact-SQL)	DELETE, INSERT, MERGE, SELECT, UPDATE

BULK INSERT

Imports a data file into a database table or view in a user-specified format.

Transact-SQL Syntax Conventions

Syntax

```
BULK INSERT
   [ database_name. [ schema_name ] . | schema_name. ] [ table_name | view_name ]
      FROM 'data_file'
     [ WITH
    (
   [ [ , ] BATCHSIZE =batch_size ]
   [ [ , ] CHECK_CONSTRAINTS ]
   [ [ , ] CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'code_page' } ]
   [ [ , ] DATAFILETYPE =
      { 'char' | 'native'| 'widechar' | 'widenative' } ]
   [ [ , ] FIELDTERMINATOR = 'field_terminator' ]
   [ [ , ] FIRSTROW = first_row ]
   [ [ , ] FIRE_TRIGGERS ]
   [ [ , ] FORMATFILE ='format_file_path' ]
   [ [ , ] KEEPIDENTITY ]
   [ [ , ] KEEPNULLS ]
   [ [ , ] KILOBYTES_PER_BATCH =kilobytes_per_batch ]
   [ [ , ] LASTROW =last_row ]
   [ [ , ] MAXERRORS =max_errors ]
   [ [ , ] ORDER ( { column [ ASC | DESC ] } [ ,...n ] ) ]
   [ [ , ] ROWS_PER_BATCH =rows_per_batch ]
   [ [ , ] ROWTERMINATOR ='row_terminator' ]
   [ [ , ] TABLOCK ]
   [ [ , ] ERRORFILE ='file_name' ]
    )]
```

Arguments

database_name

Is the database name in which the specified table or view resides. If not specified, this is the current database.

schema_name

Is the name of the table or view schema.schema_name is optional if the default schema for the user performing the bulk-import operation is schema of the specified table or view. If schema is not specified and the default schema of the user performing the bulk-import operation is different from the specified table or view, SQL Server returns an error message, and the bulk-import operation is canceled.

table_name

Is the name of the table or view to bulk import data into. Only views in which all columns refer to the same base table can be used. For more information about the restrictions for loading data into views, see INSERT (Transact-SQL).

'data_file'

Is the full path of the data file that contains data to import into the specified table or view. BULK INSERT can import data from a disk (including network, floppy disk, hard disk, and so on).

data_file must specify a valid path from the server on which SQL Server is running. If data_file is a remote file, specify the Universal Naming Convention (UNC) name. A UNC name has the form \\*Systemname**ShareName**Path**FileName*. For example, **`\\SystemX\DiskZ\Sales\update.txt`**.

BATCHSIZE = batch_size

Specifies the number of rows in a batch. Each batch is copied to the server as one transaction. If this fails, SQL Server commits or rolls back the transaction for every batch. By default, all data in the specified data file is one batch. For information about performance considerations, see "Remarks," later in this topic.

CHECK_CONSTRAINTS

Specifies that all constraints on the target table or view must be checked during the bulk-import operation. Without the CHECK_CONSTRAINTS option, any CHECK and FOREIGN KEY constraints are ignored, and after the operation, the constraint on the table is marked as not-trusted.

Note

UNIQUE, PRIMARY KEY, and NOT NULL constraints are always enforced.

At some point, you must examine the constraints on the whole table. If the table was non-empty before the bulk-import operation, the cost of revalidating the constraint may exceed the cost of applying CHECK constraints to the incremental data.

A situation in which you might want constraints disabled (the default behavior) is if the input data contains rows that violate constraints. With CHECK constraints disabled, you can import the data and then use Transact-SQL statements to remove the invalid data.

Note
The MAXERRORS option does not apply to constraint checking.

Note
In SQL Server 2005 and later versions, BULK INSERT enforces new data validation and data checks that could cause existing scripts to fail when they are executed on invalid data in a data file.

CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'code_page' }

Specifies the code page of the data in the data file. CODEPAGE is relevant only if the data contains **char**, **varchar**, or **text** columns with character values greater than **127** or less than **32**.

Note
Microsoft recommends that you specify a collation name for each column in a format file.

CODEPAGE value	Description
ACP	Columns of **char**, **varchar**, or **text** data type are converted from the ANSI/Microsoft Windows code page (ISO 1252) to the SQL Server code page.
OEM (default)	Columns of **char**, **varchar**, or **text** data type are converted from the system OEM code page to the SQL Server code page.
RAW	No conversion from one code page to another occurs; this is the fastest option.
code_page	Specific code page number, for example, 850. **iImportant** SQL Server does not support code page 65001 (UTF-8 encoding).

DATAFILETYPE = { 'char' | 'native' | 'widechar' | 'widenative' }

Specifies that BULK INSERT performs the import operation using the specified data-file type value.

DATAFILETYPE value	All data represented in:
char (default)	Character format. For more information, see Using Character Format to Import or Export Data.
native	Native (database) data types. Create the native data file by bulk importing data from SQL Server using the **bcp** utility. The native value offers a higher performance alternative to the char value. For more information, see Using Native Format to Import or Export Data.
widechar	Unicode characters. For more information, see Using Unicode Character Format to Import or Export Data.
widenative	Native (database) data types, except in **char**, **varchar**, and **text** columns, in which data is stored as Unicode. Create the **widenative** data file by bulk importing data from SQL Server using the **bcp** utility. The **widenative** value offers a higher performance alternative to **widechar**. If the data file contains ANSI extended characters, specify **widenative**. For more information, see Using Unicode Native Format to Import or Export Data.

FIELDTERMINATOR = 'field_terminator'

Specifies the field terminator to be used for **char** and **widechar** data files. The default field terminator is \t (tab character). For more information, see Specifying Field and Row Terminators.

FIRSTROW = first_row

Specifies the number of the first row to load. The default is the first row in the specified data file. FIRSTROW is 1-based.

Note

The FIRSTROW attribute is not intended to skip column headers. Skipping headers is not supported by the BULK INSERT statement. When skipping rows, the SQL Server Database Engine looks only at the field terminators, and does not validate the data in the fields of skipped rows.

FIRE_TRIGGERS

Specifies that any insert triggers defined on the destination table execute during the bulk-import operation. If triggers are defined for INSERT operations on the target table, they are fired for every completed batch.

If FIRE_TRIGGERS is not specified, no insert triggers execute.

FORMATFILE = 'format_file_path'

Specifies the full path of a format file. A format file describes the data file that contains stored responses created by using the **bcp** utility on the same table or view. The format file should be used if:

- The data file contains greater or fewer columns than the table or view.
- The columns are in a different order.
- The column delimiters vary.
- There are other changes in the data format. Format files are typically created by using the **bcp** utility and modified with a text editor as needed. For more information, see bcp Utility.

KEEPIDENTITY

Specifies that identity value or values in the imported data file are to be used for the identity column. If KEEPIDENTITY is not specified, the identity values for this column are verified but not imported and SQL Server automatically assigns unique values based on the seed and increment values specified during table creation. If the data file does not contain values for the identity column in the table or view, use a format file to specify that the identity column in the table or view is to be skipped when importing data; SQL Server automatically assigns unique values for the column. For more information, see DBCC CHECKIDENT.

For more information, see about keeping identify values see Keeping Identity Values When Bulk Importing Data.

KEEPNULLS

Specifies that empty columns should retain a null value during the bulk-import operation, instead of having any default values for the columns inserted. For more information, see Keeping Nulls or Using Default Values During Bulk Import.

KILOBYTES_PER_BATCH = kilobytes_per_batch

Specifies the approximate number of kilobytes (KB) of data per batch as kilobytes_per_batch. By default, KILOBYTES_PER_BATCH is unknown. For information about performance considerations, see "Remarks," later in this topic.

LASTROW = last_row

Specifies the number of the last row to load. The default is 0, which indicates the last row in the specified data file.

MAXERRORS = max_errors

Specifies the maximum number of syntax errors allowed in the data before the bulk-import operation is canceled. Each row that cannot be imported by the bulk-import operation is ignored and counted as one error. If max_errors is not specified, the default is 10.

Note

The MAX_ERRORS option does not apply to constraint checks or to converting **money** and **bigint** data types.

ORDER ({ column [ASC | DESC] } [,... n])

Specifies how the data in the data file is sorted. Bulk import performance is improved if the data being imported is sorted according to the clustered index on the table, if any. If the data file is sorted in a different order, that is other than the order of a clustered index key or if there is no clustered index on the table, the ORDER clause is ignored. The column names supplied must be valid column names in the destination table. By default, the bulk insert operation assumes the data file is unordered. For optimized bulk import, SQL Server also validates that the imported data is sorted.

n

Is a placeholder that indicates that multiple columns can be specified.

ROWS_PER_BATCH = rows_per_batch

Indicates the approximate number of rows of data in the data file.

By default, all the data in the data file is sent to the server as a single transaction, and the number of rows in the batch is unknown to the query optimizer. If you specify ROWS_PER_BATCH (with a value > 0) the server uses this value to optimize the bulk-import operation. The value specified for ROWS_PER_BATCH should approximately the same as the actual number of rows. For information about performance considerations, see "Remarks," later in this topic.

ROWTERMINATOR = 'row_terminator'

Specifies the row terminator to be used for **char** and **widechar** data files. The default row terminator is **\r\n** (newline character). For more information, see Specifying Field and Row Terminators.

TABLOCK

Specifies that a table-level lock is acquired for the duration of the bulk-import operation. A table can be loaded concurrently by multiple clients if the table has no indexes and TABLOCK is specified. By default, locking behavior is determined by the table option **table lock on bulk load**. Holding a lock for the duration of the bulk-import operation reduces lock contention on the table, in some cases can significantly improve performance. For information about performance considerations, see "Remarks," later in this topic.

ERRORFILE = 'file_name'

Specifies the file used to collect rows that have formatting errors and cannot be converted to an OLE DB rowset. These rows are copied into this error file from the data file "as is."

The error file is created when the command is executed. An error occurs if the file already exists. Additionally, a control file that has the extension .ERROR.txt is created. This references each row in the error file and provides error diagnostics. As soon as the errors have been corrected, the data can be loaded.

Compatibility

BULK INSERT enforces strict data validation and data checks of data read from a file that could cause existing scripts to fail when they are executed on invalid data. For example, BULK INSERT verifies that:

- The native representations of **float** or **real** data types are valid.
- Unicode data has an even-byte length.

Data Types

String-to-Decimal Data Type Conversions

The string-to-decimal data type conversions used in BULK INSERT follow the same rules as the Transact-SQL CONVERT function, which rejects strings representing numeric values that use scientific notation. Therefore, BULK INSERT treats such strings as invalid values and reports conversion errors.

To work around this behavior, use a format file to bulk import scientific notation **float** data into a decimal column. In the format file, explicitly describe the column as **real** or **float** data. For more information about these data types, see float and real (Transact-SQL).

Note

Format files represent **real** data as the **SQLFLT4** data type and **float** data as the **SQLFLT8** data type. For information about non-XML format files, see Specifying File Storage Type by Usingbcp.

Example of Importing a Numeric Value that Uses Scientific Notation

This example uses the following table:

```
CREATE TABLE t_float(c1 float, c2 decimal (5,4))
```

The user wants to bulk import data into the `t_float` table. The data file, C:\t_float-c.dat, contains scientific notation **float** data; for example:

```
8.0000000000000002E-28.0000000000000002E-2
```

However, BULK INSERT cannot import this data directly into `t_float`, because its second column, `c2`, uses the `decimal` data type. Therefore, a format file is necessary. The format file must map the scientific notation **float** data to the decimal format of column `c2`.

The following format file uses the `SQLFLT8` data type to map the second data field to the second column:

```
<?xml version="1.0"?>

<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/format"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<RECORD>

<FIELD ID="1" xsi:type="CharTerm" TERMINATOR="\t" MAX_LENGTH="30"/>

<FIELD ID="2" xsi:type="CharTerm" TERMINATOR="\r\n" MAX_LENGTH="30"/></RECORD><ROW>

<COLUMN SOURCE="1" NAME="c1" xsi:type="SQLFLT8"/>

<COLUMN SOURCE="2" NAME="c2" xsi:type="SQLFLT8"/></ROW></BCPFORMAT>
```

To use this format file (using the file name `C:\t_floatformat-c-xml.xml`) to import the test data into the test table, issue the following Transact-SQL statement:

```
BULK INSERT bulktest..t_float

FROM 'C:\t_float-c.dat' WITH (FORMATFILE='C:\t_floatformat-c-xml.xml');
```

```
GO
```

Data Types for Bulk Exporting or Importing SQLXML Documents

To bulk export or import SQLXML data, use one of the following data types in your format file:

Data type	Effect
SQLCHAR or SQLVARYCHAR	The data is sent in the client code page or in the code page implied by the collation). The effect is the same as specifying the DATAFILETYPE **='char'** without specifying a format file.
SQLNCHAR or SQLNVARCHAR	The data is sent as Unicode. The effect is the same as specifying the DATAFILETYPE **= 'widechar'** without specifying a format file.
SQLBINARY or SQLVARYBIN	The data is sent without any conversion.

General Remarks

For a comparison of the BULK INSERT statement, the INSERT ... SELECT * FROM OPENROWSET(BULK...) statement, and the **bcp** command, see Bulk Import and Export of Data (SQL Server).

For information about preparing data for bulk import, see Preparing Data for Bulk Export or Import.

The BULK INSERT statement can be executed within a user-defined transaction to import data into a table or view. Optionally, to use multiple matches for bulk importing data, a transaction can specify the BATCHSIZE clause in the BULK INSERT statement. If a multiple-batch transaction is rolled back, every batch that the transaction has sent to SQL Server is rolled back.

Interoperability

Importing Data from a CSV file

Comma-separated value (CSV) files are not supported by SQL Server bulk-import operations. However, in some cases, a CSV file can be used as the data file for a bulk import of data into SQL Server. For information about the requirements for importing data from a CSV data file, see Preparing Data for Bulk Export or Import.

Logging Behavior

For information about when row-insert operations that are performed by bulk import are logged in the transaction log, see Prerequisites for Minimal Logging in Bulk Import.

Restrictions

When using a format file with BULK INSERT, you can specify up to 1024 fields only. This is same as the maximum number of columns allowed in a table. If you use BULK INSERT with a data file that contains more than 1024 fields, BULK INSERT generates the 4822 error. The bcp utility does not have this limitation, so for data files that contain more than 1024 fields, use the **bcp** command.

Performance Considerations

If the number of pages to be flushed in a single batch exceeds an internal threshold, a full scan of the buffer pool might occur to identify which pages to flush when the batch commits. This full scan can hurt bulk-import performance. A likely case of exceeding the internal threshold occurs when a large buffer pool is combined with a slow I/O subsystem. To avoid buffer overflows on large machines, either do not use the TABLOCK hint (which will remove the bulk optimizations) or use a smaller batch size (which preserves the bulk optimizations).

Because computers vary, we recommend that you test various batch sizes with your data load to find out what works best for you.

Security

Security Account Delegation (Impersonation)

If a SQL Server user is logged in using Windows Authentication, the user can read only the files accessible to the user account, independent of the security profile of the SQL Server process.

When executing the BULK INSERT statement by using **sqlcmd** or **osql**, from one computer, inserting data into SQL Server on a second computer, and specifying a data_file on third computer by using a UNC path, you may receive a 4861 error.

To resolve this error, use SQL Server Authentication and specify a SQL Server login that uses the security profile of the SQL Server process account, or configure Windows to enable security account delegation. For information about how to enable a user account to be trusted for delegation, see Windows Help.

For more information about this and other security considerations for using BULK INSERT, see Importing Bulk Data by Using BULK INSERT or OPENROWSET(BULK...).

Permissions

Requires INSERT and ADMINISTER BULK OPERATIONS permissions. Additionally, ALTER TABLE permission is required if one or more of the following is true:

- Constraints exist and the CHECK_CONSTRAINTS option is not specified.

  Note
  Disabling constraints is the default behavior. To check constraints explicitly, use the CHECK_CONSTRAINTS option.

- Triggers exist and the FIRE_TRIGGER option is not specified.

  Note
  By default, triggers are not fired. To fire triggers explicitly, use the FIRE_TRIGGER option.

- You use the KEEPIDENTITY option to import identity value from data file.

Examples

A. Using pipes to import data from a file

The following example imports order detail information into the `AdventureWorks.Sales.SalesOrderDetail` table from the specified data file by using a pipe (`|`) as the field terminator and `|\n` as the row terminator.

```
BULK INSERT AdventureWorks.Sales.SalesOrderDetail

   FROM 'f:\orders\lineitem.tbl'

   WITH

      (

         FIELDTERMINATOR =' |',

         ROWTERMINATOR =' |\n'

      )
```

B. Using the FIRE_TRIGGERS argument

The following example specifies the `FIRE_TRIGGERS` argument.

```
BULK INSERT AdventureWorks.Sales.SalesOrderDetail

   FROM 'f:\orders\lineitem.tbl'

   WITH

     (

        FIELDTERMINATOR =' |',

        ROWTERMINATOR = ':\n',

        FIRE_TRIGGERS

      )
```

C. Using line feed as a row terminator

The following example imports a file that uses the line feed as a row terminator such as a UNIX output:

```
DECLARE @bulk_cmd varchar(1000)

SET @bulk_cmd = 'BULK INSERT AdventureWorks.Sales.SalesOrderDetail

FROM ''<drive>:\<path>\<filename>''

WITH (ROWTERMINATOR = '''+CHAR(10)+''')'

EXEC(@bulk_cmd)
```

Note

Due to how Microsoft Windows treats text files **(\n** automatically gets replaced with **\r\n)**.

Additional Examples

Other BULK INSERT examples are provided in the following topics:

- Bulk Importing and Exporting XML Documents
- Keeping Identity Values When Bulk Importing Data
- Keeping Nulls or Using Default Values During Bulk Import
- Specifying Field and Row Terminators
- Using a Format File to Bulk Import Data
- Using Character Format to Import or Export Data
- Using Native Format to Import or Export Data
- Using Unicode Character Format to Import or Export Data
- Using Unicode Native Format to Import or Export Data
- Using a Format File to Skip a Table Column
- Using a Format File to Map Table Columns to Data-File Fields

See Also

Bulk Import and Export of Data (SQL Server)

bcp Utility

Format Files for Importing or Exporting Data

INSERT (Transact-SQL)

OPENROWSET

Preparing Data for Bulk Export or Import

sp_tableoption

DELETE

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

Transact-SQL Syntax Conventions

Syntax

```
[ 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_orview_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 Hint (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.

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 Hint (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 Managing FILESTREAM Data by Using 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 a shared 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.

```
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`.

```
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.

```
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.

```
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.

```
-- 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 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
```

D. 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.

```
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.

```
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*.

```
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


-- 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 OPENQUERYrowset function. The linked server name created in the previous example is used in this example.

```
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 OPENDATASOURCErowset function. Specify a valid server name for the data source by using the format *server_name* or *server_name\instance_name*.

```
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.

```
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.

```
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

CREATE TRIGGER

INSERT

SELECT

TRUNCATE TABLE

UPDATE

WITH common_table_expression (Transact-SQL)

@@ROWCOUNT (Transact-SQL)

FROM

Specifies the tables, views, derived tables, and joined tables used in DELETE, SELECT, and UPDATE statements in SQL Server 2012. In the SELECT statement, the FROM clause is required except when the select list contains only constants, variables, and arithmetic expressions (no column names).

Transact-SQL Syntax Conventions

Syntax

```
[ FROM { <table_source> } [ ,...n ] ]
<table_source> ::=
{
    table_or_view_name [ [ AS ] table_alias ] [ <tablesample_clause> ]
        [ WITH ( < table_hint > [ [ , ]...n ] ) ]
    | rowset_function [ [ AS ] table_alias ]
        [ ( bulk_column_alias [ ,...n ] ) ]
    | user_defined_function [ [ AS ] table_alias ] ]
    | OPENXML <openxml_clause>
    | derived_table [ AS ] table_alias [ ( column_alias [ ,...n ] ) ]
    | <joined_table>
    | <pivoted_table>
    | <unpivoted_table>
      | @variable [ [ AS ] table_alias ]
        | @variable.function_call ( expression [ ,...n ] ) [ [ AS ] table_alias ] [ (column_alias [ ,...n ] ) ]
}
<tablesample_clause> ::=
    TABLESAMPLE [SYSTEM] ( sample_number [ PERCENT | ROWS ] )
        [ REPEATABLE ( repeat_seed ) ]


<joined_table> ::=
{
    <table_source><join_type><table_source> ON <search_condition>
    | <table_source> CROSS JOIN <table_source>
    | left_table_source { CROSS | OUTER } APPLY right_table_source
    | [ ( ] <joined_table> [ ) ]
}
<join_type> ::=
    [ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } [ <join_hint> ] ]
    JOIN

<pivoted_table> ::=
    table_source PIVOT <pivot_clause> [ AS ] table_alias

<pivot_clause> ::=
(aggregate_function(value_column [ [ , ]...n ])
        FOR pivot_column
        IN (<column_list>)
    )

<unpivoted_table> ::=
    table_source UNPIVOT <unpivot_clause> [ AS ] table_alias

<unpivot_clause> ::=
(value_column FOR pivot_column IN ( <column_list>) )

<column_list> ::=
column_name [ ,...n ]
```

Arguments

<table_source>

Specifies a table, view, table variable, or derived table source, with or without an alias, to use in the Transact-SQL statement. Up to 256 table sources can be used in a statement, although the limit varies depending on available memory and the complexity of other expressions in the query. Individual queries may not support up to 256 table sources.

Note

Query performance may suffer with lots of tables referenced in a query. Compilation and optimization time is also affected by additional factors. These include the presence of indexes and indexed views on each <table_source> and the size of the <select_list> in the SELECT statement.

The order of table sources after the FROM keyword does not affect the result set that is returned. SQL Server returns errors when duplicate names appear in the FROM clause.

table_or_view_name

Is the name of a table or view.

If the table or view exists in another database on the same instance of SQL Server, use a fully qualified name in the form *database*.*schema*.*object_name*.

If the table or view exists outside the instance of SQL Serverl, use a four-part name in the form *linked_server*.*catalog*.*schema*.*object*. For more information, see sp_addlinkedserver (Transact-SQL). A four-part name that is constructed by using the OPENDATASOURCEfunction as the server part of the name can also be used to specify the remote table source. When OPENDATASOURCE is specified, database_name and schema_name may not apply to all data sources and is subject to the capabilities of the OLE DB provider that accesses the remote object.

[AS] table_alias

Is an alias for table_source that can be used either for convenience or to distinguish a table or view in a self-join or subquery. An alias is frequently a shortened table name used to refer to specific columns of the tables in a join. If the same column name exists in more than one table in the join, SQL Server requires that the column name be qualified by a table name, view name, or alias. The table name cannot be used if an alias is defined.

When a derived table, rowset or table-valued function, or operator clause (such as PIVOT or UNPIVOT) is used, the required table_alias at the end of the clause is the associated table name for all columns, including grouping columns, returned.

WITH (<table_hint>)

Specifies that the query optimizer use an optimization or locking strategy with this table and for this statement. For more information, see Table Hints.

rowset_function

Specifies one of the rowset functions, such as OPENROWSET, that returns an object that can be used instead of a table reference. For more information about a list of rowset functions, see Rowset Functions.

Using the OPENROWSET and OPENQUERY functions to specify a remote object depends on the capabilities of the OLE DB provider that accesses the object.

bulk_column_alias

Is an optional alias to replace a column name in the result set. Column aliases are allowed only in SELECT statements that use the OPENROWSET function with the BULK option. When you use bulk_column_alias, specify an alias for every table column in the same order as the columns in the file.

Note

This alias overrides the NAME attribute in the COLUMN elements of an XML format file, if present.

user_defined_function

Specifies a table-valued function.

OPENXML <openxml_clause>

Provides a rowset view over an XML document. For more information, see OPENXML.

derived_table

Is a subquery that retrieves rows from the database.derived_table is used as input to the outer query.

derived_table can use the Transact-SQL table value constructor feature to specify multiple rows. For example, `SELECT * FROM (VALUES (1, 2), (3, 4), (5, 6), (7, 8), (9, 10) ) AS MyTable(a, b);`. For more information, see Table Value Constructor (Transact-SQL).

column_alias

Is an optional alias to replace a column name in the result set of the derived table. Include one column alias for each column in the select list, and enclose the complete list of column aliases in parentheses.

<tablesample_clause>

Specifies that a sample of data from the table is returned. The sample may be approximate. This clause can be used on any primary or joined table in a SELECT, UPDATE, or DELETE statement. TABLESAMPLE cannot be specified with views.

Note

When you use TABLESAMPLE against databases that are upgraded to SQL Server, the compatibility level of the database is set to 110 or higher, PIVOT is not allowed in a recursive common table expression (CTE) query. For more information, see ALTER DATABASE

Compatibility Level (Transact-SQL).

SYSTEM

Is an implementation-dependent sampling method specified by ISO standards. In SQL Server, this is the only sampling method available and is applied by default. SYSTEM applies a page-based sampling method in which a random set of pages from the table is chosen for the sample, and all the rows on those pages are returned as the sample subset.

sample_number

Is an exact or approximate constant numeric expression that represents the percent or number of rows. When specified with PERCENT, sample_number is implicitly converted to a **float** value; otherwise, it is converted to **bigint**. PERCENT is the default.

PERCENT

Specifies that a sample_number percent of the rows of the table should be retrieved from the table. When PERCENT is specified, SQL Server returns an approximate of the percent specified. When PERCENT is specified the sample_number expression must evaluate to a value from 0 to 100.

ROWS

Specifies that approximately sample_number of rows will be retrieved. When ROWS is specified, SQL Server returns an approximation of the number of rows specified. When ROWS is specified, the sample_numberexpression must evaluate to an integer value greater than zero.

REPEATABLE

Indicates that the selected sample can be returned again. When specified with the same repeat_seed value, SQL Server will return the same subset of rows as long as no changes have been made to any rows in the table. When specified with a different repeat_seed value, SQL Server will likely return some different sample of the rows in the table. The following actions to the table are considered changes: insert, update, delete, index rebuild or defragmentation, and database restore or attach.

repeat_seed

Is a constant integer expression used by SQL Server to generate a random number.repeat_seed is **bigint**. If repeat_seed is not specified, SQL Server assigns a value at random. For a specific repeat_seed value, the sampling result is always the same if no changes have been applied to the table. The repeat_seed expression must evaluate to an integer greater than zero.

<joined_table>

Is a result set that is the product of two or more tables. For multiple joins, use parentheses to change the natural order of the joins.

<join_type>

Specifies the type of join operation.

INNER

Specifies all matching pairs of rows are returned. Discards unmatched rows from both tables. When no join type is specified, this is the default.

FULL [OUTER]

Specifies that a row from either the left or right table that does not meet the join condition is included in the result set, and output columns that correspond to the other table are set to NULL. This is in addition to all rows typically returned by the INNER JOIN.

LEFT [OUTER]

Specifies that all rows from the left table not meeting the join condition are included in the result set, and output columns from the other table are set to NULL in addition to all rows returned by the inner join.

RIGHT [OUTER]

Specifies all rows from the right table not meeting the join condition are included in the result set, and output columns that correspond to the other table are set to NULL, in addition to all rows returned by the inner join.

<join_hint>

Specifies that the SQL Server query optimizer use one join hint, or execution algorithm, per join specified in the query FROM clause. For more information, see Join Hints.

JOIN

Indicates that the specified join operation should occur between the specified table sources or views.

ON <search_condition>

Specifies the condition on which the join is based. The condition can specify any predicate, although columns and comparison operators are frequently used, for example:

```
USE AdventureWorks2012 ;
GO
SELECT p.ProductID, v.BusinessEntityID
FROM Production.Product AS p
JOIN Purchasing.ProductVendor AS v
ON (p.ProductID = v.ProductID);
```

When the condition specifies columns, the columns do not have to have the same name or same data type; however, if the data types are not the same, they must be either compatible or types that SQL Server can implicitly convert. If the data types cannot be implicitly converted, the condition must explicitly convert the data type by using the CONVERT function.

There can be predicates that involve only one of the joined tables in the ON clause. Such predicates also can be in the WHERE clause in the query. Although the placement of such predicates does not make a difference for INNER joins, they might cause a different result when OUTER joins are involved. This is because the predicates in the ON clause are applied to the table before the join, whereas the WHERE clause is semantically applied to the result of the join.

For more information about search conditions and predicates, see Search Condition.

CROSS JOIN

Specifies the cross-product of two tables. Returns the same rows as if no WHERE clause was specified in an old-style, non-SQL-92-style join.

left_table_source{ CROSS | OUTER } APPLY right_table_source

Specifies that the right_table_source of the APPLY operator is evaluated against every row of the left_table_source. This functionality is useful when the right_table_source contains a table-valued function that takes column values from the left_table_source as one of its arguments.

Either CROSS or OUTER must be specified with APPLY. When CROSS is specified, no rows are produced when the right_table_source is evaluated against a specified row of the left_table_source and returns an empty result set.

When OUTER is specified, one row is produced for each row of the left_table_sourceeven when the right_table_source evaluates against that row and returns an empty result set.

For more information, see the Remarks section.

left_table_source

Is a table source as defined in the previous argument. For more information, see the Remarks section.

right_table_source

Is a table source as defined in the previous argument. For more information, see the Remarks section.

table_source PIVOT <pivot_clause>

Specifies that the table_source is pivoted based on the pivot_column. table_source is a table or table expression. The output is a table that contains all columns of the table_source except the pivot_column and value_column. The columns of the table_source, except the pivot_column and value_column, are called the grouping columns of the pivot operator.

PIVOT performs a grouping operation on the input table with regard to the grouping columns and returns one row for each group. Additionally, the output contains one column for each value specified in the column_list that appears in the pivot_column of the input_table.

For more information, see the Remarks section that follows.

aggregate_function

Is a system or user-defined aggregate function that accepts one or more inputs. The aggregate function should be invariant to null values. An aggregate function invariant to null values does not consider null values in the group while it is evaluating the aggregate value.

The COUNT(*) system aggregate function is not allowed.

value_column

Is the value column of the PIVOT operator. When used with UNPIVOT, value_column cannot be the name of an existing column in the input table_source.

FOR pivot_column

Is the pivot column of the PIVOT operator.pivot_column must be of a type implicitly or explicitly convertible to **nvarchar()**. This column cannot be **image** or **rowversion**.

When UNPIVOT is used, pivot_column is the name of the output column that becomes narrowed from the table_source. There cannot be an existing column in table_source with that name.

IN (column_list)

In the PIVOT clause, lists the values in the pivot_column that will become the column names of the output table. The list cannot specify any column names that already exist in the input table_source that is being pivoted.

In the UNPIVOT clause, lists the columns in table_source that will be narrowed into a single pivot_column.

table_alias

Is the alias name of the output table.pivot_table_alias must be specified.

UNPIVOT <unpivot_clause>

Specifies that the input table is narrowed from multiple columns in column_list into a single column called pivot_column.

Remarks

The FROM clause supports the SQL-92-SQL syntax for joined tables and derived tables. SQL-92 syntax provides the INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER, and CROSS join operators.

UNION and JOIN within a FROM clause are supported within views and in derived tables and subqueries.

A self-join is a table that is joined to itself. Insert or update operations that are based on a self-join follow the order in the FROM clause.

Because SQL Server considers distribution and cardinality statistics from linked servers that provide column distribution statistics, the REMOTE join hint is not required to force evaluating a join remotely. The SQL Server query processor considers remote statistics and determines whether a remote-join strategy is appropriate. REMOTE join hint is useful for providers that do not provide column distribution statistics.

Using APPLY

Both the left and right operands of the APPLY operator are table expressions. The main difference between these operands is that the right_table_source can use a table-valued function that takes a column from the left_table_source as one of the arguments of the function. The left_table_source can include table-valued functions, but it cannot contain arguments that are columns from the right_table_source.

The APPLY operator works in the following way to produce the table source for the FROM clause:

1. Evaluates right_table_source against each row of the left_table_source to produce rowsets.

   The values in the right_table_source depend on left_table_source. right_table_source can be represented approximately this way: `TVF(left_table_source.row)`, where `TVF` is a table-valued function.

2. Combines the result sets that are produced for each row in the evaluation of right_table_source with the left_table_source by performing a UNION ALL operation.

   The list of columns produced by the result of the APPLY operator is the set of columns from the left_table_source that is combined with the list of columns from the right_table_source.

Using PIVOT and UNPIVOT

The pivot_column and value_column are grouping columns that are used by the PIVOT operator. PIVOT follows the following process to obtain the output result set:

1. Performs a GROUP BY on its input_table against the grouping columns and produces one output row for each group.

   The grouping columns in the output row obtain the corresponding column values for that group in the input_table.

2. Generates values for the columns in the column list for each output row by performing the following:

   a. Grouping additionally the rows generated in the GROUP BY in the previous step against the pivot_column.

      For each output column in the column_list, selecting a subgroup that satisfies the condition:

      ```
      pivot_column = CONVERT(<data type of pivot_column>, 'output_column')
      ```

   b. aggregate_function is evaluated against the value_column on this subgroup and its result is returned as the value of the corresponding output_column. If the subgroup is empty, SQL Server generates a null value for that output_column. If the aggregate function is COUNT and the subgroup is empty, zero (0) is returned.

Permissions

Requires the permissions for the DELETE, SELECT, or UPDATE statement.

Examples

A. Using a simple FROM clause

The following example retrieves the `TerritoryID` and `Name` columns from the `SalesTerritory` table in the AdventureWorks2012 sample database.

```
USE AdventureWorks2012 ;
GO
SELECT TerritoryID, Name
FROM Sales.SalesTerritory
ORDER BY TerritoryID ;
```

Here is the result set.

```
TerritoryID Name
----------- ------------------------------
1           Northwest
2           Northeast
3           Central
4           Southwest
5           Southeast
6           Canada
7           France
8           Germany
9           Australia
10          United Kingdom
(10 row(s) affected)
```

B. Using the TABLOCK and HOLDLOCK optimizer hints

The following partial transaction shows how to place an explicit shared table lock on `Employee` and how to read the index. The lock is held throughout the whole transaction.

```
USE AdventureWorks2012 ;
GO
BEGIN TRAN
SELECT COUNT(*)
FROM HumanResources.Employee WITH (TABLOCK, HOLDLOCK) ;
```

C. Using the SQL-92 CROSS JOIN syntax

The following example returns the cross product of the two tables `Employee` and `Department`. A list of all possible combinations of `BusinessEntityID` rows and all `Department` name rows are returned.

```
USE AdventureWorks2012 ;
```

```
GO
SELECT e.BusinessEntityID, d.Name AS Department
FROM HumanResources.Employee AS e
CROSS JOIN HumanResources.Department AS d
ORDER BY e.BusinessEntityID, d.Name ;
```

D. Using the SQL-92 FULL OUTER JOIN syntax

The following example returns the product name and any corresponding sales orders in the `SalesOrderDetail` table. It also returns any sales orders that have no product listed in the `Product` table, and any products with a sales order other than the one listed in the `Product` table.

```
USE AdventureWorks2012 ;
GO
-- The OUTER keyword following the FULL keyword is optional.
SELECT p.Name, sod.SalesOrderID
FROM Production.Product AS p
FULL OUTER JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
WHERE p.ProductID IS NULL OR sod.ProductID IS NULL
ORDER BY p.Name ;
```

E. Using the SQL-92 LEFT OUTER JOIN syntax

The following example joins two tables on `ProductID` and preserves the unmatched rows from the left table. The `Product` table is matched with the `SalesOrderDetail` table on the `ProductID` columns in each table. All products, ordered and not ordered, appear in the result set.

```
USE AdventureWorks2012 ;
GO
SELECT p.Name, sod.SalesOrderID
FROM Production.Product AS p
LEFT OUTER JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
ORDER BY p.Name ;
```

F. Using the SQL-92 INNER JOIN syntax

The following example returns all product names and sales order IDs.

```
USE AdventureWorks2012 ;
GO
-- By default, SQL Server performs an INNER JOIN if only the JOIN
-- keyword is specified.
SELECT p.Name, sod.SalesOrderID
FROM Production.Product AS p
INNER JOIN Sales.SalesOrderDetail AS sod
```

```
ON p.ProductID = sod.ProductID
ORDER BY p.Name ;
```

\G. Using the SQL-92 RIGHT OUTER JOIN syntax

The following example joins two tables on `TerritoryID` and preserves the unmatched rows from the right table. The `SalesTerritory` table is matched with the `SalesPerson` table on the `TerritoryID` column in each table. All salespersons appear in the result set, whether or not they are assigned a territory.

```
USE AdventureWorks2012 ;
GO
SELECT st.Name AS Territory, sp.BusinessEntityID
FROM Sales.SalesTerritory AS st
RIGHT OUTER JOIN Sales.SalesPerson AS sp
ON st.TerritoryID = sp.TerritoryID ;
```

H. Using HASH and MERGE join hints

The following example performs a three-table join among the `Product`, `ProductVendor`, and `Vendor` tables to produce a list of products and their vendors. The query optimizer joins `Product` and `ProductVendor` (`p` and `pv`) by using a MERGE join. Next, the results of the `Product` and `ProductVendor` MERGE join (`p` and `pv`) are HASH joined to the `Vendor` table to produce (`p` and `pv`) and `v`.

Important
After a join hint is specified, the INNER keyword is no longer optional and must be explicitly stated for an INNER JOIN to be performed.

```
USE AdventureWorks2012 ;
GO
SELECT p.Name AS ProductName, v.Name AS VendorName
FROM Production.Product AS p
INNER MERGE JOIN Purchasing.ProductVendor AS pv
ON p.ProductID = pv.ProductID
INNER HASH JOIN Purchasing.Vendor AS v
ON pv.BusinessEntityID = v.BusinessEntityID
ORDER BY p.Name, v.Name ;
```

I. Using a derived table

The following example uses a derived table, a `SELECT` statement after the `FROM` clause, to return the first and last names of all employees and the cities in which they live.

```
USE AdventureWorks2012 ;
GO
SELECT RTRIM(p.FirstName) + ' ' + LTRIM(p.LastName) AS Name, d.City
```

```
FROM Person.Person AS p
INNER JOIN HumanResources.Employee e ON p.BusinessEntityID = e.BusinessEntityID
INNER JOIN
   (SELECT bea.BusinessEntityID, a.City
    FROM Person.Address AS a
    INNER JOIN Person.BusinessEntityAddress AS bea
    ON a.AddressID = bea.AddressID) AS d
ON p.BusinessEntityID = d.BusinessEntityID
ORDER BY p.LastName, p.FirstName;
```

J. Using TABLESAMPLE to read data from a sample of rows in a table

The following example uses `TABLESAMPLE` in the `FROM` clause to return approximately `10` percent of all the rows in the `Customer` table.

```
USE AdventureWorks2012 ;
GO
SELECT *
FROM Sales.Customer TABLESAMPLE SYSTEM (10 PERCENT) ;
```

K. Using APPLY

The following example assumes that the following tables with the following schema exist in the database:

- `Departments`: `DeptID`, `DivisionID`, `DeptName`, `DeptMgrID`
- `EmpMgr`: `MgrID`, `EmpID`
- `Employees`: `EmpID`, `EmpLastName`, `EmpFirstName`, `EmpSalary`

There is also a table-valued function, `GetReports(MgrID)` that returns the list of all employees (`EmpID`, `EmpLastName`, `EmpSalary`) that report directly or indirectly to the specified `MgrID`.

The example uses `APPLY` to return all departments and all employees in that department. If a particular department does not have any employees, there will not be any rows returned for that department.

```
SELECT DeptID, DeptName, DeptMgrID, EmpID, EmpLastName, EmpSalary
FROM Departments d CROSS APPLY dbo.GetReports(d.DeptMgrID) ;
```

If you want the query to produce rows for those departments without employees, which will produce null values for the `EmpID`, `EmpLastName` and `EmpSalary` columns, use `OUTER APPLY` instead.

```
SELECT DeptID, DeptName, DeptMgrID, EmpID, EmpLastName, EmpSalary
FROM Departments d OUTER APPLY dbo.GetReports(d.DeptMgrID) ;
```

L. Using PIVOT and UNPIVOT

The following example returns the number of purchase orders placed by employee IDs `164`, `198`, `223`, `231`, and `233`, categorized by vendor ID.

```
USE AdventureWorks2012;
GO
SELECT VendorID, [250] AS Emp1, [251] AS Emp2, [256] AS Emp3, [257] AS Emp4, [260] AS
Emp5
FROM
(SELECT PurchaseOrderID, EmployeeID, VendorID
FROM Purchasing.PurchaseOrderHeader) AS p
PIVOT
(
COUNT (PurchaseOrderID)
FOR EmployeeID IN
( [250], [251], [256], [257], [260] )
) AS pvt
ORDER BY VendorID;
```

Here is a partial result set:

```
VendorID    Emp1        Emp2        Emp3        Emp4        Emp5
----------------------------------------------------------------
1           4           3           5           4           4
2           4           1           5           5           5
3           4           3           5           4           4
4           4           2           5           5           4
5           5           1           5           5           5
```

To unpivot the table, assume the result set produced in the previous example is stored as `pvt`. The query looks like the following.

```
--Create the table and insert values as portrayed in the previous example.
CREATE TABLE dbo.pvt (VendorID int, Emp1 int, Emp2 int,
Emp3 int, Emp4 int, Emp5 int);
GO
INSERT INTO dbo.pvt VALUES
 (1,4,3,5,4,4)
,(2,4,1,5,5,5)
,(3,4,3,5,4,4)
,(4,4,2,5,5,4)
,(5,5,1,5,5,5);
GO
--Unpivot the table.
SELECT VendorID, Employee, Orders
FROM
    (SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp5
    FROM dbo.pvt) AS p
```

```
UNPIVOT
    (Orders FOR Employee IN
        (Emp1, Emp2, Emp3, Emp4, Emp5)
)AS unpvt
GO
```

Here is a partial result set:

```
VendorID    Employee    Orders
------------------------------
1           Emp1        4
1           Emp2        3
1           Emp3        5
1           Emp4        4
1           Emp5        4
2           Emp1        4
2           Emp2        1
2           Emp3        5
2           Emp4        5
2           Emp5        5
```

M. Using CROSS APPLY

The following example retrieves a snapshot of all query plans residing in the plan cache, by querying the `sys.dm_exec_cached_plans` dynamic management view to retrieve the plan handles of all query plans in the cache. Then the `CROSS APPLY` operator is specified to pass the plan handles to `sys.dm_exec_query_plan`. The XML Showplan output for each plan currently in the plan cache is in the `query_plan` column of the table that is returned.

```
USE master;
GO
SELECT dbid, object_id, query_plan
FROM sys.dm_exec_cached_plans AS cp
CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle);
GO
```

See Also

CONTAINSTABLE

DELETE

FREETEXTTABLE

INSERT

OPENQUERY

OPENROWSET

Operators

UPDATE

WHERE

Hints

Hints are options or strategies specified for enforcement by the SQL Server query processor on SELECT, INSERT, UPDATE, or DELETE statements. The hints override any execution plan the query optimizer might select for a query.

Caution

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

The following hints are described in this section:

- Join Hints
- Query Hints
- Table Hint

Join Hints

Join hints specify that the query optimizer enforce a join strategy between two tables.

Important

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

Applies to:

DELETE

SELECT

UPDATE

Transact-SQL Syntax Conventions

Syntax

```
<join_hint> ::=
   { LOOP | HASH | MERGE | REMOTE }
```

Arguments

LOOP | HASH | MERGE

Specifies that the join in the query should use looping, hashing, or merging. Using LOOP |HASH | MERGE JOIN enforces a particular join between two tables. LOOP cannot be specified together with RIGHT or FULL as a join type.

REMOTE

Specifies that the join operation is performed on the site of the right table. This is useful when the left table is a local table and the right table is a remote table. REMOTE should be used only when the left table has fewer rows than the right table.

If the right table is local, the join is performed locally. If both tables are remote but from different data sources, REMOTE causes the join to be performed on the site of the right table. If both tables are remote tables from the same data source, REMOTE is not required.

REMOTE cannot be used when one of the values being compared in the join predicate is cast to a different collation using the COLLATE clause.

REMOTE can be used only for INNER JOIN operations.

Remarks

Join hints are specified in the FROM clause of a query. Join hints enforce a join strategy between two tables. If a join hint is specified for any two tables, the query optimizer automatically enforces the join order for all joined tables in the query, based on the position of the ON keywords. When a CROSS JOIN is used without the ON clause, parentheses can be used to indicate the join order.

Examples

A. Using HASH

The following example specifies that the `JOIN` operation in the query is performed by a `HASH` join.

```
USE AdventureWorks2012;
GO
SELECT p.Name, pr.ProductReviewID
FROM Production.Product AS p
LEFT OUTER HASH JOIN Production.ProductReview AS pr
ON p.ProductID = pr.ProductID
ORDER BY ProductReviewID DESC;
```

B. Using LOOP

The following example specifies that the `JOIN` operation in the query is performed by a `LOOP` join.

```
USE AdventureWorks2012;
GO
DELETE FROM Sales.SalesPersonQuotaHistory
FROM Sales.SalesPersonQuotaHistory AS spqh
    INNER LOOP JOIN Sales.SalesPerson AS sp
    ON spqh.SalesPersonID = sp.SalesPersonID
WHERE sp.SalesYTD > 2500000.00;
GO
```

C. Using MERGE

The following example specifies that the `JOIN` operation in the query is performed by a `MERGE` join.

```
USE AdventureWorks2012;
GO
SELECT poh.PurchaseOrderID, poh.OrderDate, pod.ProductID, pod.DueDate, poh.VendorID
FROM Purchasing.PurchaseOrderHeader AS poh
INNER MERGE JOIN Purchasing.PurchaseOrderDetail AS pod
    ON poh.PurchaseOrderID = pod.PurchaseOrderID;
GO
```

See Also

Hints (Transact-SQL)

Query Hints

Query hints specify that the indicated hints should be used throughout the query. They affect all operators in the statement. If UNION is involved in the main query, only the last query involving a UNION operation can have the OPTION clause. Query hints are specified as part of the OPTION clause. If one or more query hints cause the query optimizer not to generate a valid plan, error 8622 is raised.

> **Caution**
> Because the SQL Server query optimizer typically selects the best execution plan for a query, we recommend only using hints as a last resort for experienced developers and database administrators.

Applies to:

DELETE

INSERT

SELECT

UPDATE

MERGE

Transact-SQL Syntax Conventions

Syntax

```
<query_hint > ::=
{ { HASH | ORDER } GROUP
  | { CONCAT | HASH | MERGE } UNION
  | { LOOP | MERGE | HASH } JOIN
  | EXPAND VIEWS
  | FAST number_rows
  | FORCE ORDER
  | IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX
  | KEEP PLAN
  | KEEPFIXED PLAN
  | MAXDOP number_of_processors
  | MAXRECURSION number
  | OPTIMIZE FOR ( @variable_name { UNKNOWN | = literal_constant } [ , ...n ] )
  | OPTIMIZE FOR UNKNOWN
  | PARAMETERIZATION { SIMPLE | FORCED }
  | RECOMPILE
  | ROBUST PLAN
  | USE PLAN N'xml_plan'
  | TABLE HINT ( exposed_object_name [ ,<table_hint> [ [, ]...n ] ] )
}

<table_hint> ::=
[ NOEXPAND ] {
    INDEX (index_value [ ,...n ] ) | INDEX = (index_value)
  | FORCESEEK [(index_value(index_column_name [,... ] )) ]
  | FORCESCAN
  | HOLDLOCK
  | NOLOCK
  | NOWAIT
  | PAGLOCK
  | READCOMMITTED
  | READCOMMITTEDLOCK
  | READPAST
  | READUNCOMMITTED
  | REPEATABLEREAD
  | ROWLOCK
  | SERIALIZABLE
  | SPATIAL_WINDOW_MAX_CELLS = integer
  | TABLOCK
  | TABLOCKX
  | UPDLOCK
  | XLOCK
}
```

Arguments

{ HASH | ORDER } GROUP

Specifies that aggregations described in the GROUP BY, or DISTINCT clause of the query should use hashing or ordering.

{ MERGE | HASH | CONCAT } UNION

Specifies that all UNION operations are performed by merging, hashing, or concatenating UNION sets. If more than one UNION hint is specified, the query optimizer selects the least expensive strategy from those hints specified.

{ LOOP | MERGE | HASH } JOIN

Specifies that all join operations are performed by LOOP JOIN, MERGE JOIN, or HASH JOIN in the whole query. If more than one join hint is specified, the optimizer selects the least expensive join strategy from the allowed ones.

If, in the same query, a join hint is also specified in the FROM clause for a specific pair of tables, this join hint takes precedence in the joining of the two tables, although the query hints still must be honored. Therefore, the join hint for the pair of tables may only restrict the selection of allowed join methods in the query hint. For more information, see Join Hints (Transact-SQL).

EXPAND VIEWS

Specifies that the indexed views are expanded and the query optimizer will not consider any indexed view as a substitute for any part of the query. A view is expanded when the view name is replaced by the view definition in the query text.

This query hint virtually disallows direct use of indexed views and indexes on indexed views in the query plan.

The indexed view is not expanded only if the view is directly referenced in the SELECT part of the query and WITH (NOEXPAND) or WITH (NOEXPAND, INDEX(index_value [,...n])) is specified. For more information about the query hint WITH (NOEXPAND), see FROM.

Only the views in the SELECT part of statements, including those in INSERT, UPDATE, MERGE, and DELETE statements are affected by the hint.

FAST number_rows

Specifies that the query is optimized for fast retrieval of the first number_rows. This is a nonnegative integer. After the first number_rows are returned, the query continues execution and produces its full result set.

FORCE ORDER

Specifies that the join order indicated by the query syntax is preserved during query optimization. Using FORCE ORDER does not affect possible role reversal behavior of the query optimizer.

Note

In a MERGE statement, the source table is accessed before the target table as the default join order, unless the WHEN SOURCE NOT MATCHED clause is specified. Specifying FORCE ORDER preserves this default behavior.

KEEP PLAN

Forces the query optimizer to relax the estimated recompile threshold for a query. The estimated recompile threshold is the point at which a query is automatically recompiled when the estimated number of indexed column changes have been made to a table by running UPDATE, DELETE, MERGE, or INSERT statements. Specifying KEEP PLAN makes sure that a query will not be recompiled as frequently when there are multiple updates to a table.

KEEPFIXED PLAN

Forces the query optimizer not to recompile a query due to changes in statistics. Specifying KEEPFIXED PLAN makes sure that a query will be recompiled only if the schema of the underlying tables is changed or if **sp_recompile** is executed against those tables.

IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX

Prevents the query from using a nonclusteredxVelocity memory optimized columnstore index. If the query contains the query hint to avoid use of the columnstore index and an index hint to use a columnstore index, the hints are in conflict and the query returns an error.

MAXDOP number

Overrides the **max degree of parallelism** configuration option of **sp_configure** and Resource Governor for the query specifying this option. The MAXDOP query hint can exceed the value configured with sp_configure. If MAXDOP exceeds the value configured with Resource Governor, the Database Engine uses the Resource Governor MAXDOP value, described in ALTER WORKLOAD GROUP (Transact-SQL). All semantic rules used with the **max degree of parallelism** configuration option are applicable when you use the MAXDOP query hint. For more information, see Configure the max degree of parallelism Server Configuration Option.

Warning

If MAXDOP is set to zero then the server chooses the max degree of parallelism.

MAXRECURSION number

Specifies the maximum number of recursions allowed for this query. number is a nonnegative integer between 0 and 32767. When 0 is specified, no limit is applied. If this option is not specified, the default limit for the server is 100.

When the specified or default number for MAXRECURSION limit is reached during query execution, the query is ended and an error is returned.

Because of this error, all effects of the statement are rolled back. If the statement is a SELECT statement, partial results or no results may be returned. Any partial results returned may not include all rows on recursion levels beyond the specified maximum recursion level.

For more information, see WITH common_table_expression (Transact-SQL).

OPTIMIZE FOR (@variable_name { UNKNOWN | = literal_constant } [, ...n])

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.

@variable_name

Is the name of a local variable used in a query, to which a value may be assigned for use with the OPTIMIZE FOR query hint.

UNKNOWN

Specifies that the query optimizer use statistical data instead of the initial value to determine the value for a local variable during query optimization.

literal_constant

Is a literal constant value to be assigned @variable_name for use with the OPTIMIZE FOR query hint.literal_constant is used only during query optimization, and not as the value of @variable_name during query execution. literal_constant can be of any SQL Server system data type that can be expressed as a literal constant. The data type of literal_constant must be implicitly convertible to the data type that @variable_name references in the query.

OPTIMIZE FOR can counteract the default parameter detection behavior of the optimizer or can be used when you create plan guides. For more information, see Recompiling a Stored Procedure.

OPTIMIZE FOR UNKNOWN

Instructs the query optimizer to use statistical data instead of the initial values for all local variables when the query is compiled and optimized, including parameters created with forced parameterization.

If OPTIMIZE FOR @variable_name = literal_constant and OPTIMIZE FOR UNKNOWN are used in the same query hint, the query optimizer will use the literal_constant that is specified for a specific value and UNKNOWN for the remaining variable values. The values are used only during query optimization, and not during query execution.

PARAMETERIZATION { SIMPLE | FORCED }

Specifies the parameterization rules that the SQL Server query optimizer applies to the query when it is compiled.

Important

The PARAMETERIZATION query hint can only be specified inside a plan guide. It cannot be specified directly within a query.

SIMPLE instructs the query optimizer to attempt simple parameterization. FORCED instructs the optimizer to attempt forced parameterization. The PARAMETERIZATION query hint is used to override the current setting of the PARAMETERIZATION database SET option inside a plan guide. For more information, see Specifying Query Parameterization Behavior Using Plan Guides.

RECOMPILE

Instructs the SQL Server Database Engine to discard the plan generated for the query after it executes, forcing the query optimizer to recompile a query plan the next time the same query is executed. Without specifying RECOMPILE, the Database Engine caches query plans and reuses them. When compiling query plans, the RECOMPILE query hint uses the current values of any local variables in the query and, if the query is inside a stored procedure, the current values passed to any parameters.

RECOMPILE is a useful alternative to creating a stored procedure that uses the WITH RECOMPILE clause when only a subset of queries inside the stored procedure, instead of the whole stored procedure, must be recompiled. For more information, see Recompiling a Stored Procedure. RECOMPILE is also useful when you create plan guides.

ROBUST PLAN

Forces the query optimizer to try a plan that works for the maximum potential row size, possibly at the expense of performance. When the query is processed, intermediate tables and operators may have to store and process rows that are wider than any one of the input rows. The rows may be so wide that, sometimes, the particular operator cannot process the row. If this occurs, the Database Engine produces an error during query execution. By using ROBUST PLAN, you instruct the query optimizer not to consider any query plans that may encounter this problem.

If such a plan is not possible, the query optimizer returns an error instead of deferring error detection to query execution. Rows may contain variable-length columns; the Database Engine allows for rows to be defined that have a maximum potential size beyond the ability of the Database Engine to process them. Generally, despite the maximum potential size, an application stores rows that have actual sizes within the limits that the Database Engine can process. If the Database Engine encounters a row that is too long, an execution error is returned.

USE PLAN N'xml_plan'

Forces the query optimizer to use an existing query plan for a query that is specified by **'**xml_plan**'**. USE PLAN cannot be specified with INSERT, UPDATE, MERGE, or DELETE statements.

TABLE HINT (exposed_object_name [, <table_hint> [[,]...n]])

Applies the specified table hint to the table or view that corresponds to exposed_object_name. We recommend using a table hint as a query hint only in the context of a plan guide.

exposed_object_name can be one of the following references:

- When an alias is used for the table or view in the FROM clause of the query, exposed_object_name is the alias.
- When an alias is not used, exposed_object_name is the exact match of the table or view referenced in the FROM clause. For example, if the table or view is referenced using a two-part name, exposed_object_name is the same two-part name.

When exposed_object_name is specified without also specifying a table hint, any indexes specified in the query as part of a table hint for the object are disregarded and index usage is determined by the query optimizer. You can use this technique to eliminate the effect of an INDEX table hint when you cannot modify the original query. See Example J.

<table_hint> ::= { [NOEXPAND] { INDEX (index_value [,...n]) | INDEX = (index_value) | FORCESEEK [(index_value (index_column_name [,...]))]| FORCESCAN | HOLDLOCK | NOLOCK | NOWAIT | PAGLOCK | READCOMMITTED | READCOMMITTEDLOCK | READPAST | READUNCOMMITTED | REPEATABLEREAD | ROWLOCK | SERIALIZABLE |SPATIAL_WINDOW_MAX_CELLS | TABLOCK | TABLOCKX | UPDLOCK | XLOCK }

Is the table hint to apply to the table or view that corresponds to exposed_object_name as a query hint. For a description of these hints, see Table Hints (Transact-SQL).

Table hints other than INDEX, FORCESCAN, and FORCESEEK are disallowed as query hints unless the query already has a WITH clause specifying the table hint. For more information, see Remarks.

Caution

Specifying FORCESEEK with parameters limits the number of plans that can be considered by the optimizer more than when specifying FORCESEEK without parameters. This may cause a "Plan cannot be generated" error to occur in more cases. In a future release, internal modifications to the optimizer may allow more plans to be considered.

Remarks

Query hints cannot be specified in an INSERT statement except when a SELECT clause is used inside the statement.

Query hints can be specified only in the top-level query, not in subqueries. When a table hint is specified as a query hint, the hint can be specified in the top-level query or in a subquery; however, the value specified for exposed_object_name in the TABLE HINT clause must match exactly the exposed name in the query or subquery.

Specifying Table Hints as Query Hints

We recommend using the INDEX, FORCESCAN or FORCESEEK table hint as a query hint only in the context of a plan guide. Plan guides are useful when you cannot modify the original query, for example, because it is a third-party application. The query hint specified in the plan guide is added to the query before it is compiled and optimized. For ad-hoc queries, use the TABLE HINT clause only when testing plan guide statements. For all other ad-hoc queries, we recommend specifying these hints only as table hints.

When specified as a query hint, the INDEX, FORCESCAN, and FORCESEEK table hints are valid for the following objects:

- Tables
- Views
- Indexed views
- Common table expressions (the hint must be specified in the SELECT statement whose result set populates the common table expression)
- Dynamic management views
- Named subqueries

The INDEX, FORCESCAN, and FORCESEEK table hints can be specified as query hints for a query that does not have any existing table hints, or they can be used to replace existing INDEX, FORCESCAN or FORCESEEK hints in the query, respectively. Table hints other than INDEX, FORCESCAN, and FORCESEEK are disallowed as query hints unless the query already has a WITH clause specifying the table hint. In this case, a matching hint must also be specified as a query hint by using TABLE HINT in the OPTION clause to preserve the semantics of the query. For example, if the query contains the table hint NOLOCK, the OPTION clause in the **@hints** parameter of the plan guide must also contain the NOLOCK hint. See Example K. When a table hint other than INDEX, FORCESCAN, or FORCESEEK is specified by using TABLE HINT in the OPTION clause without a matching query hint, or vice versa; error 8702 is raised (indicating that the OPTION clause can cause the semantics of the query to change) and the query fails.

Examples

A. Using MERGE JOIN

The following example specifies that the `JOIN` operation in the query is performed by `MERGE JOIN`.

```
USE AdventureWorks2012;
GO
SELECT *
FROM Sales.Customer AS c
INNER JOIN Sales.vStoreWithAddresses AS sa
    ON c.CustomerID = sa.BusinessEntityID
WHERE TerritoryID = 5
OPTION (MERGE JOIN);
GO
```

B. Using OPTIMIZE FOR

The following example instructs the query optimizer to use the value `'Seattle'` for local variable `@city_name` and to use statistical data to determine the value for the local variable `@postal_code` when optimizing the query.

```
USE AdventureWorks2012;
```

```
GO
DECLARE @city_name nvarchar(30);
DECLARE @postal_code nvarchar(15);
SET @city_name = 'Ascheim';
SET @postal_code = 86171;
SELECT * FROM Person.Address
WHERE City = @city_name AND PostalCode = @postal_code
OPTION ( OPTIMIZE FOR (@city_name = 'Seattle', @postal_code UNKNOWN) );
GO
```

C. Using MAXRECURSION

MAXRECURSION can be used to prevent a poorly formed recursive common table expression from entering into an infinite loop. The following example intentionally creates an infinite loop and uses the `MAXRECURSION` hint to limit the number of recursion levels to two.

```
USE AdventureWorks2012;
GO
--Creates an infinite loop
WITH cte (CustomerID, PersonID, StoreID) AS
(
    SELECT CustomerID, PersonID, StoreID
    FROM Sales.Customer
    WHERE PersonID IS NOT NULL
  UNION ALL
    SELECT cte.CustomerID, cte.PersonID, cte.StoreID
    FROM cte
    JOIN  Sales.Customer AS e
        ON cte.PersonID = e.CustomerID
)
--Uses MAXRECURSION to limit the recursive levels to 2
SELECT CustomerID, PersonID, StoreID
FROM cte
OPTION (MAXRECURSION 2);
GO
```

After the coding error is corrected, `MAXRECURSION` is no longer required.

D. Using MERGE UNION

The following example uses the `MERGE UNION` query hint.

```
USE AdventureWorks2012;
```

```
GO
SELECT BusinessEntityID, JobTitle, HireDate, VacationHours, SickLeaveHours
FROM HumanResources.Employee AS e1
UNION
SELECT BusinessEntityID, JobTitle, HireDate, VacationHours, SickLeaveHours
FROM HumanResources.Employee AS e2
OPTION (MERGE UNION);
GO
```

E. Using HASH GROUP and FAST

The following example uses the `HASH GROUP` and `FAST` query hints.

```
USE AdventureWorks2012;
GO
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE UnitPrice < $5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (HASH GROUP, FAST 10);
GO
```

F. Using MAXDOP

The following example uses the `MAXDOP` query hint.

```
USE AdventureWorks2012 ;
GO
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE UnitPrice < $5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (MAXDOP 2);
GO
```

G. Using INDEX

The following examples use the INDEX hint. The first example specifies a single index. The second example specifies multiple indexes for a single table reference. In both examples, because the INDEX hint is applied on a table that uses an alias, the TABLE HINT clause must also specify the same alias as the exposed object name.

```
USE AdventureWorks2012;
GO
EXEC sp_create_plan_guide
```

```
    @name = N'Guide1',
    @stmt = N'SELECT c.LastName, c.FirstName, e.JobTitle
              FROM HumanResources.Employee AS e
              JOIN Person.Person AS c ON e.BusinessEntityID = c.BusinessEntityID
              WHERE e.OrganizationLevel = 2;',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (TABLE HINT(e, INDEX
(IX_Employee_OrganizationLevel_OrganizationNode)))';
GO
EXEC sp_create_plan_guide
    @name = N'Guide2',
    @stmt = N'SELECT c.LastName, c.FirstName, e.JobTitle
              FROM HumanResources.Employee AS e
              JOIN Person.Person AS c ON e.BusinessEntityID = c.BusinessEntityID
              WHERE e.OrganizationLevel = 2;',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (TABLE HINT(e, INDEX(PK_Employee_BusinessEntityID,
IX_Employee_OrganizationLevel_OrganizationNode)))';
GO
```

H. Using FORCESEEK

The following example uses the FORCESEEK table hint. Because the INDEX hint is applied on a table that uses a two-part name, the TABLE HINT clause must also specify the same two-part name as the exposed object name.

```
USE AdventureWorks2012;
GO
EXEC sp_create_plan_guide
    @name = N'Guide3',
    @stmt = N'SELECT c.LastName, c.FirstName, HumanResources.Employee.JobTitle
              FROM HumanResources.Employee
              JOIN Person.Person AS c ON HumanResources.Employee.BusinessEntityID =
c.BusinessEntityID
              WHERE HumanResources.Employee.OrganizationLevel = 3
              ORDER BY c.LastName, c.FirstName;',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (TABLE HINT( HumanResources.Employee, FORCESEEK))';
GO
```

I. Using multiple table hints

The following example applies the INDEX hint to one table and the FORCESEEK hint to another.

```
USE AdventureWorks2012;
GO
EXEC sp_create_plan_guide
    @name = N'Guide4',
    @stmt = N'SELECT c.LastName, c.FirstName, e.JobTitle
              FROM HumanResources.Employee AS e
              JOIN Person.Person AS c ON e.BusinessEntityID = c.BusinessEntityID
              WHERE OrganizationLevel = 3;',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (TABLE HINT ( e, INDEX(
IX_Employee_OrganizationLevel_OrganizationNode ) )
                       , TABLE HINT ( c, FORCESEEK) )';
GO
```

J. Using TABLE HINT to override an existing table hint

The following example shows how to use the TABLE HINT hint without specifying a hint to override the behavior of the INDEX table hint specified in the FROM clause of the query.

```
USE AdventureWorks2012;
GO
EXEC sp_create_plan_guide
    @name = N'Guide5',
    @stmt = N'SELECT c.LastName, c.FirstName, e.JobTitle
              FROM HumanResources.Employee AS e WITH (INDEX
(IX_Employee_OrganizationLevel_OrganizationNode))
              JOIN Person.Person AS c ON e.BusinessEntityID = c.BusinessEntityID
              WHERE OrganizationLevel = 3;',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (TABLE HINT(e))';
GO
```

K. Specifying semantics-affecting table hints

The following example contains two table hints in the query: NOLOCK, which is semantic-affecting, and INDEX, which is non-semantic-affecting. To preserve the semantics of the query, the NOLOCK hint is specified in the OPTIONS clause of the plan guide. In addition to the NOLOCK hint, the INDEX and FORCESEEK hints are specified and replace the non-semantic-affecting INDEX hint in the query when the statement is compiled and optimized.

```
USE AdventureWorks2012;
GO
EXEC sp_create_plan_guide
    @name = N'Guide6',
    @stmt = N'SELECT c.LastName, c.FirstName, e.JobTitle
              FROM HumanResources.Employee AS e
              JOIN Person.Person AS c ON e.BusinessEntityID = c.BusinessEntityID
              WHERE OrganizationLevel = 3;',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (TABLE HINT ( e, INDEX(
IX_Employee_OrganizationLevel_OrganizationNode) , NOLOCK, FORCESEEK ))';
GO
```

The following example shows an alternative method to preserving the semantics of the query and allowing the optimizer to choose an index other than the index specified in the table hint. This is done by specifying the NOLOCK hint in the OPTIONS clause (because it is semantic-affecting) and specifying the TABLE HINT keyword with only a table reference and no INDEX hint.

```
USE AdventureWorks2012;
GO
EXEC sp_create_plan_guide
    @name = N'Guide7',
    @stmt = N'SELECT c.LastName, c.FirstName, e.JobTitle
              FROM HumanResources.Employee AS e
              JOIN Person.Person AS c ON e.BusinessEntityID = c.BusinessEntityID
              WHERE OrganizationLevel = 2;',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (TABLE HINT ( e, NOLOCK))';
GO
```

See Also

Hints (Transact-SQL)

sp_create_plan_guide (Transact-SQL)

sp_control_plan_guide (Transact-SQL)

Table Hints

Table hints override the default behavior of the query optimizer for the duration of the data manipulation language (DML) statement by specifying a locking method, one or more indexes, a query-processing operation such as a table scan or index seek, or other options. Table hints are specified in the FROM clause of the DML statement and affect only the table or view referenced in that clause.

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.

Applies to:

DELETE

INSERT

SELECT

UPDATE

MERGE

Transact-SQL Syntax Conventions

Syntax

```
WITH  (<table_hint> [ [, ]...n ] )


<table_hint> ::=
[ NOEXPAND ] {
    INDEX  (index_value [ ,...n ] ) | INDEX =  (index_value)    | FORCESEEK
[(index_value(index_column_name  [ ,... ] )) ]
  | FORCESCAN
  | FORCESEEK
  | HOLDLOCK
  | NOLOCK
  | NOWAIT
  | PAGLOCK
  | READCOMMITTED
  | READCOMMITTEDLOCK
  | READPAST
  | READUNCOMMITTED
  | REPEATABLEREAD
  | ROWLOCK
  | SERIALIZABLE
  | SPATIAL_WINDOW_MAX_CELLS = integer
  | TABLOCK
  | TABLOCKX
  | UPDLOCK
  | XLOCK
}

<table_hint_limited> ::=
{
    KEEPIDENTITY
  | KEEPDEFAULTS
  | HOLDLOCK
  | IGNORE_CONSTRAINTS
  | IGNORE_TRIGGERS
  | NOLOCK
  | NOWAIT
  | PAGLOCK
  | READCOMMITTED
  | READCOMMITTEDLOCK
  | READPAST
  | REPEATABLEREAD
  | ROWLOCK
  | SERIALIZABLE
  | TABLOCK
  | TABLOCKX
  | UPDLOCK
  | XLOCK
}
```

Arguments

WITH (<table_hint>) [[,]...n]

With some exceptions, table hints are supported in the FROM clause only when the hints are specified with the WITH keyword. Table hints also must be specified with parentheses.

Important

Omitting the WITH keyword is a deprecated feature: This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

The following table hints are allowed with and without the WITH keyword: NOLOCK, READUNCOMMITTED, UPDLOCK, REPEATABLEREAD, SERIALIZABLE, READCOMMITTED, TABLOCK, TABLOCKX, PAGLOCK, ROWLOCK, NOWAIT, READPAST, XLOCK, and NOEXPAND. When these table hints are specified without the WITH keyword, the hints should be specified alone. For example:

```
FROM t (TABLOCK)
```

When the hint is specified with another option, the hint must be specified with the WITH keyword:

```
FROM t WITH (TABLOCK, INDEX(myindex))
```

We recommend using commas between table hints.

Important

Separating hints by spaces rather than commas is a deprecated feature: This feature will be removed in a future version of Microsoft SQL Server. Do not use this feature in new development work, and modify applications that currently use this feature as soon as possible.

The restrictions apply when the hints are used in queries against databases with the compatibility level of 90 and higher.

NOEXPAND

Specifies that any indexed views are not expanded to access underlying tables when the query optimizer processes the query. The query optimizer treats the view like a table with clustered index. NOEXPAND applies only to indexed views. For more information, see Remarks.

INDEX (index_value [,... n]) | INDEX = (index_value)

The INDEX() syntax specifies the names or IDs of one or more indexes to be used by the query optimizer when it processes the statement. The alternative INDEX = syntax specifies a single index value. Only one index hint per table can be specified.

If a clustered index exists, INDEX(0) forces a clustered index scan and INDEX(1) forces a clustered index scan or seek. If no clustered index exists, INDEX(0) forces a table scan and INDEX(1) is interpreted as an error.

If multiple indexes are used in a single hint list, the duplicates are ignored and the rest of the listed indexes are used to retrieve the rows of the table. The order of the indexes in the index hint is significant. A multiple index hint also enforces index ANDing, and the query optimizer applies as many conditions as possible on each index accessed. If the collection of hinted indexes do not include all columns referenced by the query, a fetch is performed to retrieve the remaining columns after the SQL Server Database Engine retrieves all the indexed columns.

Note

When an index hint referring to multiple indexes is used on the fact table in a star join, the optimizer ignores the index hint and returns a warning message. Also, index ORing is not allowed for a table with an index hint specified.

The maximum number of indexes in the table hint is 250 nonclustered indexes.

KEEPIDENTITY

Is applicable only in an INSERT statement when the BULK option is used with OPENROWSET.

Specifies that identity value or values in the imported data file are to be used for the identity column. If KEEPIDENTITY is not specified, the identity values for this column are verified but not imported and the query optimizer automatically assigns unique values based on the seed and increment values specified during table creation.

Important

If the data file does not contain values for the identity column in the table or view, and the identity column is not the last column in the table, you must skip the identity column. For more information, see sys.foreign_keys (Transact-SQL). If an identity column is skipped successfully, the query optimizer automatically assigns unique values for the identity column into the imported table rows.

For an example that uses this hint in an INSERT ... SELECT * FROM OPENROWSET(BULK...) statement, see Keeping Identity Values When Bulk Importing Data.

For information about checking the identity value for a table, see DBCC CHECKIDENT.

KEEPDEFAULTS

Is applicable only in an INSERT statement when the BULK option is used with OPENROWSET.

Specifies insertion of a table column's default value, if any, instead of NULL when the data record lacks a value for the column.

For an example that uses this hint in an INSERT ... SELECT * FROM OPENROWSET(BULK...) statement, see Keeping Nulls or Using Default Values During Bulk Import.

FORCESEEK [(index_value (index_column_name [,... n]))]

Specifies that the query optimizer use only an index seek operation as the access path to the data in the table or view. Starting with SQL Server 2008 R2 SP1, index parameters can also be specified. In that case, the query optimizer considers only index seek operations through the specified index using at least the specified index columns.

index_value

Is the index name or index ID value. The index ID 0 (heap) cannot be specified. To return the index name or ID, query the sys.indexes catalog view.

index_column_name

Is the name of the index column to include in the seek operation. Specifying FORCESEEK with index parameters is similar to using FORCESEEK with an INDEX hint. However, you can achieve greater control over the access path used by the query optimizer by specifying both the index to seek on and the index columns to consider in the seek operation. The optimizer may consider additional columns if needed. For example, if a nonclustered index is specified, the optimizer may choose to use clustered index key columns in addition to the specified columns.

The FORCESEEK hint can be specified in the following ways.

Syntax	Example	Description
Without an index or INDEX hint	`FROM dbo.MyTable WITH (FORCESEEK)`	The query optimizer considers only index seek operations to access the table or view through any relevant index.
Combined with an INDEX hint	`FROM dbo.MyTable WITH (FORCESEEK, INDEX (MyIndex))`	The query optimizer considers only index seek operations to access the table or view through the specified index.
Parameterized by specifying an index and index columns	`FROM dbo.MyTable WITH (FORCESEEK (MyIndex (col1, col2, col3)))`	The query optimizer considers only index seek operations to access the table or view through the specified index using at least the specified index columns.

When using the FORCESEEK hint (with or without index parameters), consider the following guidelines.

- The hint can be specified as a table hint or as a query hint. For more information about query hints, see Query Hints (Transact-SQL).

- To apply FORCESEEK to an indexed view, the NOEXPAND hint must also be specified.
- The hint can be applied at most once per table or view.
- The hint cannot be specified for a remote data source. Error 7377 is returned when FORCESEEK is specified with an index hint and error 8180 is returned when FORCESEEK is used without an index hint.
- If FORCESEEK causes no plan to be found, error 8622 is returned.

When FORCESEEK is specified with index parameters, the following guidelines and restrictions apply.

- The hint cannot be specified in combination with either an INDEX hint or another FORCESEEK hint.
- At least one column must be specified and it must be the leading key column.
- Additional index columns can be specified, however, key columns cannot be skipped. For example, if the specified index contains the key columns `a`, `b`, and `c`, valid syntax would include `FORCESEEK (MyIndex (a))` and `FORCESEEK (MyIndex (a, b)`. Invalid syntax would include `FORCESEEK (MyIndex (c))` and `FORCESEEK (MyIndex (a, c)`.
- The order of column names specified in the hint must match the order of the columns in the referenced index.
- Columns that are not in the index key definition cannot be specified. For example, in a nonclustered index, only the defined index key columns can be specified. Clustered key columns that are automatically included in the index cannot be specified, but may be used by the optimizer.
- An xVelocity memory optimized columnstore index cannot be specified as an index parameter. Error 366 is returned.
- Modifying the index definition (for example, by adding or removing columns) may require modifications to the queries that reference that index.
- The hint prevents the optimizer from considering any spatial or XML indexes on the table.
- The hint cannot be specified in combination with the FORCESCAN hint.
- For partitioned indexes, the partitioning column implicitly added by SQL Server cannot be specified in the FORCESEEK hint.

cCaution

Specifying FORCESEEK with parameters limits the number of plans that can be considered by the optimizer more than when specifying FORCESEEK without parameters. This may cause a "Plan cannot be generated" error to occur in more cases. In a future release, internal modifications to the optimizer may allow more plans to be considered.

FORCESCAN

Introduced in SQL Server 2008 R2 SP1, this hint specifies that the query optimizer use only an index scan operation as the access path to the referenced table or view. The FORCESCAN hint can be useful for queries in which the optimizer underestimates the number of affected rows and chooses a seek operation rather than a scan operation. When this occurs, the amount of memory granted for the operation is too small and query performance is impacted.

FORCESCAN can be specified with or without an INDEX hint. When combined with an index hint, (`INDEX = index_name, FORCESCAN`), the query optimizer considers only scan access paths through the specified index when accessing the referenced table. FORCESCAN can be specified with the index hint INDEX(0) to force a table scan operation on the base table.

For partitioned tables and indexes, FORCESCAN is applied after partitions have been eliminated through query predicate evaluation. This means that the scan is applied only to the remaining partitions and not to the entire table.

The FORCESCAN hint has the following restrictions.

- The hint cannot be specified for a table that is the target of an INSERT, UPDATE, or DELETE statement.
- The hint cannot be used with more than one index hint.
- The hint prevents the optimizer from considering any spatial or XML indexes on the table.
- The hint cannot be specified for a remote data source.
- The hint cannot be specified in combination with the FORCESEEK hint.

HOLDLOCK

Is equivalent to SERIALIZABLE. For more information, see SERIALIZABLE later in this topic. HOLDLOCK applies only to the table or view for which it is specified and only for the duration of the transaction defined by the statement that it is used in. HOLDLOCK cannot be used in a SELECT statement that includes the FOR BROWSE option.

IGNORE_CONSTRAINTS

Is applicable only in an INSERT statement when the BULK option is used with OPENROWSET.

Specifies that any constraints on the table are ignored by the bulk-import operation. By default, INSERT checks Unique Constraints and Check Constraints and Primary and Foreign Key Constraints. When IGNORE_CONSTRAINTS is specified for a bulk-import operation, INSERT must ignore these constraints on a target table. Note that you cannot disable UNIQUE, PRIMARY KEY, or NOT NULL constraints.

You might want to disable CHECK and FOREIGN KEY constraints if the input data contains rows that violate constraints. By disabling the CHECK and FOREIGN KEY constraints, you can import the data and then use Transact-SQL statements to clean up the data.

However, when CHECK and FOREIGN KEY constraints are ignored, each ignored constraint on the table is marked as is_not_trusted in the sys.check_constraints or sys.foreign_keys catalog view after the operation. At some point, you should check the constraints on the whole table. If the table was not empty before the bulk import operation, the cost of revalidating the constraint may exceed the cost of applying CHECK and FOREIGN KEY constraints to the incremental data.

IGNORE_TRIGGERS

Is applicable only in an INSERT statement when the BULK option is used with OPENROWSET.

Specifies that any triggers defined on the table are ignored by the bulk-import operation. By default, INSERT applies triggers.

Use IGNORE_TRIGGERS only if your application does not depend on any triggers and maximizing performance is important.

NOLOCK

Is equivalent to READUNCOMMITTED. For more information, see READUNCOMMITTED later in this topic.

Note

For UPDATE or DELETE statements: This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

NOWAIT

Instructs the Database Engine to return a message as soon as a lock is encountered on the table. NOWAIT is equivalent to specifying SET LOCK_TIMEOUT 0 for a specific table.

PAGLOCK

Takes page locks either where individual locks are ordinarily taken on rows or keys, or where a single table lock is ordinarily taken. By default, uses the lock mode appropriate for the operation. When specified in transactions operating at the SNAPSHOT isolation level, page locks are not taken unless PAGLOCK is combined with other table hints that require locks, such as UPDLOCK and HOLDLOCK.

READCOMMITTED

Specifies that read operations comply with the rules for the READ COMMITTED isolation level by using either locking or row versioning. If the database option READ_COMMITTED_SNAPSHOT is OFF, the Database Engine acquires shared locks as data is read and releases those locks when the read operation is completed. If the database option READ_COMMITTED_SNAPSHOT is ON, the Database Engine does not acquire locks and uses row versioning. For more information about isolation levels, see SET TRANSACTION ISOLATION LEVEL.

Note

For UPDATE or DELETE statements: This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

READCOMMITTEDLOCK

Specifies that read operations comply with the rules for the READ COMMITTED isolation level by using locking. The Database Engine acquires shared locks as data is read and releases those locks when the read operation is completed, regardless of the setting of the READ_COMMITTED_SNAPSHOT database option. For more information about isolation levels, see SET TRANSACTION ISOLATION LEVEL. This hint cannot be specified on the target table of an INSERT statement; error 4140 is returned.

READPAST

Specifies that the Database Engine not read rows that are locked by other transactions. When READPAST is specified, row-level locks are skipped. That is, the Database Engine skips past the rows instead of blocking the current transaction until the locks are released. For example, assume table `T1` contains a single integer column with the values of 1, 2, 3, 4, 5. If transaction A changes the value of 3 to 8 but has not yet committed, a SELECT * FROM T1 (READPAST) yields values 1, 2, 4, 5. READPAST is primarily used to reduce locking contention when implementing a work queue that uses a SQL Server table. A queue reader that uses READPAST skips past queue entries locked by other transactions to the next available queue entry, without having to wait until the other transactions release their locks.

READPAST can be specified for any table referenced in an UPDATE or DELETE statement, and any table referenced in a FROM clause. When specified in an UPDATE statement, READPAST is applied only when reading data to identify which records to update, regardless of where in the statement it is specified. READPAST cannot be specified for tables in the INTO clause of an INSERT statement. Read operations that use READPAST do not block. Update or delete operations that use READPAST may block when reading foreign keys or indexed views, or when modifying secondary indexes.

READPAST can only be specified in transactions operating at the READ COMMITTED or REPEATABLE READ isolation levels. When specified in transactions operating at the SNAPSHOT isolation level, READPAST must be combined with other table hints that require locks, such as UPDLOCK and HOLDLOCK.

The READPAST table hint cannot be specified when the READ_COMMITTED_SNAPSHOT database option is set to ON and either of the following conditions is true.

- The transaction isolation level of the session is READ COMMITTED.
- The READCOMMITTED table hint is also specified in the query.

To specify the READPAST hint in these cases, remove the READCOMMITTED table hint if present, and include the READCOMMITTEDLOCK table hint in the query.

READUNCOMMITTED

Specifies that dirty reads are allowed. No shared locks are issued to prevent other transactions from modifying data read by the current transaction, and exclusive locks set by other transactions do not block the current transaction from reading the locked data. Allowing dirty reads can cause higher concurrency, but at the cost of reading data modifications that then are rolled back by other transactions. This may generate errors for your transaction, present users with data that was never committed, or cause users to see records twice (or not at all).

READUNCOMMITTED and NOLOCK hints apply only to data locks. All queries, including those with READUNCOMMITTED and NOLOCK hints, acquire Sch-S (schema stability) locks during compilation and execution. Because of this, queries are blocked when a concurrent transaction holds a Sch-M (schema modification) lock on the table. For example, a data definition language (DDL) operation acquires a Sch-M lock before it modifies the schema information of the table. Any concurrent queries, including those running with READUNCOMMITTED or NOLOCK hints, are blocked when attempting to acquire a Sch-S lock. Conversely, a query holding a Sch-S lock blocks a concurrent transaction that attempts to acquire a Sch-M lock.

READUNCOMMITTED and NOLOCK cannot be specified for tables modified by insert, update, or delete operations. The SQL Server query optimizer ignores the READUNCOMMITTED and NOLOCK hints in the FROM clause that apply to the target table of an UPDATE or DELETE statement.

Note

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.

You can minimize locking contention while protecting transactions from dirty reads of uncommitted data modifications by using either of the following:

- The READ COMMITTED isolation level with the READ_COMMITTED_SNAPSHOT database option set ON.
- The SNAPSHOT isolation level.

For more information about isolation levels, see SET TRANSACTION ISOLATION LEVEL.

Note

If you receive the error message 601 when READUNCOMMITTED is specified, resolve it as you would a deadlock error (1205), and retry your statement.

REPEATABLEREAD

Specifies that a scan is performed with the same locking semantics as a transaction running at REPEATABLE READ isolation level. For more information about isolation levels, see SET TRANSACTION ISOLATION LEVEL (Transact-SQL).

ROWLOCK

Specifies that row locks are taken when page or table locks are ordinarily taken. When specified in transactions operating at the SNAPSHOT isolation level, row locks are not taken unless ROWLOCK is combined with other table hints that require locks, such as UPDLOCK and HOLDLOCK.

SPATIAL_WINDOW_MAX_CELLS = integer

Specifies the maximum number of cells to use for tessellating a geometry or geography object. number is a value between 1 and 8192.

This option allows for fine-tuning of query execution time by adjusting the tradeoff between primary and secondary filter execution time. A larger number reduces secondary filter execution time, but increases primary execution filter time and a smaller number decreases primary filter execution time, but increase secondary filter execution. For denser spatial data, a higher number should produce a faster execution time by giving a better approximation with the primary filter and reducing secondary filter execution time. For sparser data, a lower number will decrease the primary filter execution time.

This option works for both manual and automatic grid tessellations.

SERIALIZABLE

Is equivalent to HOLDLOCK. Makes shared locks more restrictive by holding them until a transaction is completed, instead of releasing the shared lock as soon as the required table or data page is no longer needed, whether the transaction has been completed or not. The scan is performed with the same semantics as a transaction running at the

SERIALIZABLE isolation level. For more information about isolation levels, see SET TRANSACTION ISOLATION LEVEL.

TABLOCK

Specifies that the acquired lock is applied at the table level. The type of lock that is acquired depends on the statement being executed. For example, a SELECT statement may acquire a shared lock. By specifying TABLOCK, the shared lock is applied to the entire table instead of at the row or page level. If HOLDLOCK is also specified, the table lock is held until the end of the transaction.

When importing data into a heap by using the INSERT INTO <target_table> SELECT <columns> FROM <source_table> statement, you can enable optimized logging and locking for the statement by specifying the TABLOCK hint for the target table. In addition, the recovery model of the database must be set to simple or bulk-logged. For more information, see INSERT (Transact-SQL).

When used with the OPENROWSET bulk rowset provider to import data into a table, TABLOCK enables multiple clients to concurrently load data into the target table with optimized logging and locking. For more information, see Prerequisites for Minimal Logging in Bulk Import.

TABLOCKX

Specifies that an exclusive lock is taken on the table.

UPDLOCK

Specifies that update locks are to be taken and held until the transaction completes. UPDLOCK takes update locks for read operations only at the row-level or page-level. If UPDLOCK is combined with TABLOCK, or a table-level lock is taken for some other reason, an exclusive (X) lock will be taken instead.

When UPDLOCK is specified, the READCOMMITTED and READCOMMITTEDLOCK isolation level hints are ignored. For example, if the isolation level of the session is set to SERIALIZABLE and a query specifies (UPDLOCK, READCOMMITTED), the READCOMMITTED hint is ignored and the transaction is run using the SERIALIZABLE isolation level.

XLOCK

Specifies that exclusive locks are to be taken and held until the transaction completes. If specified with ROWLOCK, PAGLOCK, or TABLOCK, the exclusive locks apply to the appropriate level of granularity.

Remarks

The table hints are ignored if the table is not accessed by the query plan. This may be caused by the optimizer choosing not to access the table at all, or because an indexed view is accessed instead. In the latter case, accessing an indexed view can be prevented by using the OPTION (EXPAND VIEWS) query hint.

All lock hints are propagated to all the tables and views that are accessed by the query plan, including tables and views referenced in a view. Also, SQL Server performs the corresponding lock consistency checks.

Lock hints ROWLOCK, UPDLOCK, AND XLOCK that acquire row-level locks may place locks on index keys rather than the actual data rows. For example, if a table has a nonclustered index, and a SELECT statement using a lock hint is handled by a covering index, a lock is acquired on the index key in the covering index rather than on the data row in the base table.

If a table contains computed columns that are computed by expressions or functions accessing columns in other tables, the table hints are not used on those tables and are not propagated. For example, a NOLOCK table hint is specified on a table in the query. This table has computed columns that are computed by a combination of expressions and functions that access columns in another table. The tables referenced by the expressions and functions do not use the NOLOCK table hint when accessed.

SQL Server does not allow for more than one table hint from each of the following groups for each table in the FROM clause:

- Granularity hints: PAGLOCK, NOLOCK, READCOMMITTEDLOCK, ROWLOCK, TABLOCK, or TABLOCKX.
- Isolation level hints: HOLDLOCK, NOLOCK, READCOMMITTED, REPEATABLEREAD, SERIALIZABLE.

Filtered Index Hints

A filtered index can be used as a table hint, but will cause the query optimizer to generate error 8622 if it does not cover all of the rows that the query selects. The following is an example of an invalid filtered index hint. The example creates the filtered index `FIBillOfMaterialsWithComponentID` and then uses it as an index hint for a SELECT statement. The filtered index predicate includes data rows for ComponentIDs 533, 324, and 753. The query predicate also includes data rows for ComponentIDs 533, 324, and 753 but extends the result set to include ComponentIDs 855 and 924, which are not in the filtered index. Therefore, the query optimizer cannot use the filtered index hint and generates error 8622. For more information, see Filtered Index Design Guidelines.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.indexes
    WHERE name = N'FIBillOfMaterialsWithComponentID'
    AND object_id = OBJECT_ID(N'Production.BillOfMaterials'))
DROP INDEX FIBillOfMaterialsWithComponentID
```

```
    ON Production.BillOfMaterials;
GO
CREATE NONCLUSTERED INDEX "FIBillOfMaterialsWithComponentID"
    ON Production.BillOfMaterials (ComponentID, StartDate, EndDate)
    WHERE ComponentID IN (533, 324, 753);
GO
SELECT StartDate, ComponentID FROM Production.BillOfMaterials
    WITH( INDEX (FIBillOfMaterialsWithComponentID) )
    WHERE ComponentID in (533, 324, 753, 855, 924);
GO
```

The query optimizer will not consider an index hint if the SET options do not have the required values for filtered indexes. For more information, see CREATE INDEX (Transact-SQL).

Using NOEXPAND

NOEXPAND applies only to *indexed views*. An indexed view is a view with a unique clustered index created on it. If a query contains references to columns that are present both in an indexed view and base tables, and the query optimizer determines that using the indexed view provides the best method for executing the query, the query optimizer uses the index on the view. This function is called *indexed view matching*. Automatic use of indexed view by query optimizer is supported only in specific editions of SQL Server. For a list of features that are supported by the editions of SQL Server, see Features Supported by the Editions of SQL Server 2012 (http://go.microsoft.com/fwlink/?linkid=232473).

However, for the optimizer to consider indexed views for matching, or use an indexed view that is referenced with the NOEXPAND hint, the following SET options must be set to ON.

ANSI_NULLS	ANSI_WARNINGS	CONCAT_NULL_YIELDS_NULL
ANSI_PADDING	ARITHABORT[1]	QUOTED_IDENTIFIERS

[1] ARITHABORT is implicitly set to ON when ANSI_WARNINGS is set to ON. Therefore, you do not have to manually adjust this setting.

Also, the NUMERIC_ROUNDABORT option must be set to OFF.

To force the optimizer to use an index for an indexed view, specify the NOEXPAND option. This hint can be used only if the view is also named in the query. SQL Server does not provide a hint to force a particular indexed view to be used in a query that does not name the view directly in the FROM clause; however, the query optimizer considers using indexed views, even if they are not referenced directly in the query.

Using a Table Hint as a Query Hint

Table hints can also be specified as a query hint by using the OPTION (TABLE HINT) clause. We recommend using a table hint as a query hint only in the context of a plan guide. For ad-hoc queries, specify these hints only as table hints. For more information, see Query Hints (Transact-SQL).

Permissions

The KEEPIDENTITY, IGNORE_CONSTRAINTS, and IGNORE_TRIGGERS hints require ALTER permissions on the table.

Examples

A. Using the TABLOCK hint to specify a locking method

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

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

B. Using the FORCESEEK hint to specify an index seek operation

The following example uses the FORCESEEK hint without specifying an index to force the query optimizer to perform an index seek operation on the `Sales.SalesOrderDetail` table.

```
USE AdventureWorks2012;
GO
SELECT *
FROM Sales.SalesOrderHeader AS h
INNER JOIN Sales.SalesOrderDetail AS d WITH (FORCESEEK)
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.TotalDue > 100
AND (d.OrderQty > 5 OR d.LineTotal < 1000.00);
GO
```

The following example uses the FORCESEEK hint with an index to force the query optimizer to perform an index seek operation on the specified index and index column.

```
USE AdventureWorks2012;
GO
```

```
SELECT h.SalesOrderID, h.TotalDue, d.OrderQty
FROM Sales.SalesOrderHeader AS h
    INNER JOIN Sales.SalesOrderDetail AS d
    WITH (FORCESEEK (PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID (SalesOrderID)))
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.TotalDue > 100
AND (d.OrderQty > 5 OR d.LineTotal < 1000.00);
GO
```

C. Using the FORCECAN hint to specify an index scan operation

The following example uses the FORCESCAN hint to force the query optimizer to perform a scan operation on the Sales.SalesOrderDetail table.

```
USE AdventureWorks2012;
GO
SELECT h.SalesOrderID, h.TotalDue, d.OrderQty
FROM Sales.SalesOrderHeader AS h
    INNER JOIN Sales.SalesOrderDetail AS d
    WITH (FORCESCAN)
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.TotalDue > 100
AND (d.OrderQty > 5 OR d.LineTotal < 1000.00);
```

See Also

OPENROWSET

Hints (Transact-SQL)

Query Hints (Transact-SQL)

INSERT

Adds one or more rows to a table or a view in SQL Server 2012. For examples, see Examples.

Transact-SQL Syntax Conventions

Syntax

[WITH <common_table_expression> [,...**n**]]

INSERT

{

```
        [ TOP (expression) [ PERCENT ] ]
        [ INTO ]
        { <object> | rowset_function_limited
          [ WITH (<Table_Hint_Limited> [ ...n ] ) ]
        }
    {
        [ (column_list) ]
        [ <OUTPUT Clause> ]
        { VALUES ( { DEFAULT | NULL | expression } [ ,...n ] ) [ ,...n     ]
        | derived_table
        | execute_statement
        | <dml_table_source>
        | DEFAULT VALUES
        }
    }
}
[;]

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

<dml_table_source> ::=
    SELECT <select_list>
    FROM (<dml_statement_with_output_clause>)
      [AS] table_alias [ ( column_alias [ ,...n ] ) ]
    [ WHERE <search_condition> ]
        [ OPTION ( <query_hint> [ ,...n ] ) ]

<column_definition> ::=
 column_name <data_type>
```

```
    [ COLLATE collation_name ]
    [ NULL | NOT NULL ]

<data type> ::=
[ type_schema_name . ] type_name
    [ ( precision [ , scale ] | max ]
-- External tool only syntax
INSERT
{
    [BULK]
    [ database_name . [ schema_name ] . | schema_name . ]
    [ table_name | view_name ]
    ( <column_definition> )
    [ WITH (
        [ [ , ] CHECK_CONSTRAINTS ]
        [ [ , ] FIRE_TRIGGERS ]
        [ [ , ] KEEP_NULLS ]
        [ [ , ] KILOBYTES_PER_BATCH = kilobytes_per_batch ]
        [ [ , ] ROWS_PER_BATCH = rows_per_batch ]
        [ [ , ] ORDER ( { column [ ASC | DESC ] } [ ,...n ] ) ]
        [ [ , ] TABLOCK ]
    ) ]
}
[; ]
```

Arguments

WITH <common_table_expression>

Specifies the temporary named result set, also known as common table expression, defined within the scope of the INSERT statement. The result set is derived from a SELECT statement. For more information, see WITH common_table_expression (Transact-SQL).

TOP (expression) [PERCENT]

Specifies the number or percent of random rows that will be inserted. expression can be either a number or a percent of the rows. For more information, see TOP (Transact-SQL).

INTO

Is an optional keyword that can be used between INSERT and the target table.

server_name

Is the name of the linked server on which the table or view is located.server_name can be specified as a linked server name, or by using the OPENDATASOURCE function.

When server_name is specified as a linked server, database_name and schema_name are required. When server_name is specified with OPENDATASOURCE, database_name and schema_name may not apply to all data sources and is subject to the capabilities of the OLE DB provider that accesses the remote object.

database_name

Is the name of the database.

schema_name

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

table_orview_name

Is the name of the table or view that is to receive the data.

A table variable, within its scope, can be used as a table source in an INSERT 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. For example, an INSERT into a multi-table view must use a column_list that references only columns from one base table. For more information about updatable views, see CREATE VIEW (Transact-SQL).

rowset_function_limited

Is either the OPENQUERY or OPENROWSET function. Use of these functions is subject to the capabilities of the OLE DB provider that accesses the remote object.

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.

READPAST, NOLOCK, and READUNCOMMITTED are not allowed. For more information about table hints, see Table Hint (Transact-SQL).

Important

The ability to specify the HOLDLOCK, SERIALIZABLE, READCOMMITTED, REPEATABLEREAD, or UPDLOCK hints on tables that are targets of INSERT statements will be removed in a future version of SQL Server. These hints do not affect the performance of INSERT statements. Avoid using them in new development work, and plan to modify applications that currently use them.

Specifying the TABLOCK hint on a table that is the target of an INSERT statement has the same effect as specifying the TABLOCKX hint. An exclusive lock is taken on the table.

(column_list)

Is a list of one or more columns in which to insert data.*column_list* must be enclosed in parentheses and delimited by commas.

If a column is not in *column_list*, the Database Engine must be able to provide a value based on the definition of the column; otherwise, the row cannot be loaded. The Database Engine automatically provides a value for the column if the column:

- Has an IDENTITY property. The next incremental identity value is used.
- Has a default. The default value for the column is used.
- Has a **timestamp** data type. The current timestamp value is used.
- Is nullable. A null value is used.
- Is a computed column. The calculated value is used.

column_list must be used when explicit values are inserted into an identity column, and the SET IDENTITY_INSERT option must be ON for the table.

OUTPUT Clause

Returns inserted rows as part of the insert operation. The results can be returned to the processing application or inserted into a table or table variable for further processing.

The OUTPUT clause is not supported in DML statements that reference local partitioned views, distributed partitioned views, or remote tables, or INSERT statements that contain an *execute_statement*. The OUTPUT INTO clause is not supported in INSERT statements that contain a <dml_table_source> clause.

VALUES

Introduces the list or lists of data values to be inserted. There must be one data value for each column in *column_list*, if specified, or in the table. The value list must be enclosed in parentheses.

If the values in the Value list are not in the same order as the columns in the table or do not have a value for each column in the table, *column_list* must be used to explicitly specify the column that stores each incoming value.

You can use the Transact-SQL row constructor (also called a table value constructor) to specify multiple rows in a single INSERT statement. The row constructor consists of a single VALUES clause with multiple value lists enclosed in parentheses and separated by a comma. For more information, see Table Value Constructor (Transact-SQL).

DEFAULT

Forces the Database Engine to load the default value defined for a column. If a default does not exist for the column and the column allows null values, NULL is inserted. For a column defined with the **timestamp** data type, the next timestamp value is inserted. DEFAULT is not valid for an identity column.

expression

Is a constant, a variable, or an expression. The expression cannot contain an EXECUTE statement.

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.

derived_table

Is any valid SELECT statement that returns rows of data to be loaded into the table. The SELECT statement cannot contain a common table expression (CTE).

execute_statement

Is any valid EXECUTE statement that returns data with SELECT or READTEXT statements. For more information, see EXECUTE (Transact-SQL).

The RESULT SETS options of the EXECUTE statement cannot be specified in an INSERT…EXEC statement.

If execute_statement is used with INSERT, each result set must be compatible with the columns in the table or in column_list.

execute_statement can be used to execute stored procedures on the same server or a remote server. The procedure in the remote server is executed, and the result sets are returned to the local server and loaded into the table in the local server. In a distributed transaction, execute_statement cannot be issued against a loopback linked server when the connection has multiple active result sets (MARS) enabled.

If execute_statement returns data with the READTEXT statement, each READTEXT statement can return a maximum of 1 MB (1024 KB) of data. execute_statement can also be used with extended procedures. execute_statement inserts the data returned by the main thread of the extended procedure; however, output from threads other than the main thread are not inserted.

You cannot specify a table-valued parameter as the target of an INSERT EXEC statement; however, it can be specified as a source in the INSERT EXEC string or stored-procedure. For more information, see Table-valued Parameters (Database Engine).

<dml_table_source>

Specifies that the rows inserted into the target table are those returned by the OUTPUT clause of an INSERT, UPDATE, DELETE, or MERGE statement, optionally filtered by a WHERE clause. If <dml_table_source> is specified, the target of the outer INSERT statement must meet the following restrictions:

- It must be a base table, not a view.
- It cannot be a remote table.
- It cannot have any triggers defined on it.
- It cannot participate in any primary key-foreign key relationships.
- It cannot participate in merge replication or updatable subscriptions for transactional replication.

The compatibility level of the database must be set to 100 or higher. For more information, see OUTPUT Clause (Transact-SQL).

<select_list>

Is a comma-separated list specifying which columns returned by the OUTPUT clause to insert. The columns in <select_list> must be compatible with the columns into which values are being inserted. <select_list> cannot reference aggregate functions or TEXTPTR.

Note

Any variables listed in the SELECT list refer to their original values, regardless of any changes made to them in <dml_statement_with_output_clause>.

<dml_statement_with_output_clause>

Is a valid INSERT, UPDATE, DELETE, or MERGE statement that returns affected rows in an OUTPUT clause. The statement cannot contain a WITH clause, and cannot target remote tables or partitioned views. If UPDATE or DELETE is specified, it cannot be a cursor-based UPDATE or DELETE. Source rows cannot be referenced as nested DML statements.

WHERE <search_condition>

Is any WHERE clause containing a valid <search_condition> that filters the rows returned by <dml_statement_with_output_clause>. For more information, see Search Condition (Transact-SQL). When used in this context, <search_condition> cannot contain subqueries, scalar user-defined functions that perform data access, aggregate functions, TEXTPTR, or full-text search predicates.

DEFAULT VALUES

Forces the new row to contain the default values defined for each column.

BULK

Used by external tools to upload a binary data stream. This option is not intended for use with tools such as SQL Server Management Studio, SQLCMD, OSQL, or data access application programming interfaces such as SQL Server Native Client.

FIRE_TRIGGERS

Specifies that any insert triggers defined on the destination table execute during the binary data stream upload operation. For more information, see BULK INSERT (Transact-SQL).

CHECK_CONSTRAINTS

Specifies that all constraints on the target table or view must be checked during the binary data stream upload operation. For more information, see BULK INSERT (Transact-SQL).

KEEPNULLS

Specifies that empty columns should retain a null value during the binary data stream upload operation. For more information, see Keeping Nulls or Using Default Values During Bulk Import.

KILOBYTES_PER_BATCH = kilobytes_per_batch

Specifies the approximate number of kilobytes (KB) of data per batch as kilobytes_per_batch. For more information, see BULK INSERT (Transact-SQL).

ROWS_PER_BATCH = rows_per_batch

Indicates the approximate number of rows of data in the binary data stream. For more information, see BULK INSERT (Transact-SQL).

Note A syntax error is raised if a column list is not provided.

Best Practices

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

Best Practices for Bulk Importing Data

Using INSERT INTO…SELECT to Bulk Import Data with Minimal Logging

You can use INSERT INTO <target_table> SELECT <columns> FROM <source_table> to efficiently transfer a large number of rows from one table, such as a staging table, to another table with minimal logging. Minimal logging can improve the performance of the statement and reduce the possibility of the operation filling the available transaction log space during the transaction.

Minimal logging for this statement has the following requirements:

- The recovery model of the database is set to simple or bulk-logged.
- The target table is an empty or nonempty heap.
- The target table is not used in replication.
- The TABLOCK hint is specified for the target table.

Rows that are inserted into a heap as the result of an insert action in a MERGE statement may also be minimally logged.

Unlike the BULK INSERT statement, which holds a less restrictive Bulk Update lock, INSERT INTO…SELECT with the TABLOCK hint holds an exclusive (X) lock on the table. This means that you cannot insert rows using parallel insert operations.

Using OPENROWSET and BULK to Bulk Import Data

The OPENROWSET function can accept the following table hints, which provide bulk-load optimizations with the INSERT statement:

- The TABLOCK hint can minimize the number of log records for the insert operation. The recovery model of the database must be set to simple or bulk-logged and the target table cannot be used in replication. For more information, see Prerequisites for Minimal Logging in Bulk Import.
- The IGNORE_CONSTRAINTS hint can temporarily disable FOREIGN KEY and CHECK constraint checking.
- The IGNORE_TRIGGERS hint can temporarily disable trigger execution.
- The KEEPDEFAULTS hint allows the insertion of a table column's default value, if any, instead of NULL when the data record lacks a value for the column.
- The KEEPIDENTITY hint allows the identity values in the imported data file to be used for the identity column in the target table.

These optimizations are similar to those available with the BULK INSERT command. For more information, see Table Hints (Transact-SQL).

Data Types

When you insert rows, consider the following data type behavior:

- If a value is being loaded into columns with a **char**, **varchar**, or **varbinary** data type, the padding or truncation of trailing blanks (spaces for **char** and **varchar**, zeros for **varbinary**) is determined by the SET ANSI_PADDING setting defined for the column when the table was created. For more information, see SET ANSI_PADDING (Transact-SQL).

  The following table shows the default operation for SET ANSI_PADDING OFF.

Data type	Default operation
char	Pad value with spaces to the defined width of column.
varchar	Remove trailing spaces to the last non-space character or to a single-space character for strings made up of only spaces.
varbinary	Remove trailing zeros.

- If an empty string (' ') is loaded into a column with a **varchar** or **text** data type, the default operation is to load a zero-length string.
- Inserting a null value into a **text** or **image** column does not create a valid text pointer, nor does it preallocate an 8-KB text page.
- Columns created with the **uniqueidentifier** data type store specially formatted 16-byte binary values. Unlike with identity columns, the Database Engine does not automatically generate values for columns with the **uniqueidentifier** data type. During an insert operation, variables with a data type of **uniqueidentifier** and string constants in the form *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx* (36 characters including hyphens, where *x* is a hexadecimal digit in the range 0-9 or a-f) can be used for **uniqueidentifier** columns. For example, 6F9619FF-8B86-D011-B42D-00C04FC964FF is a valid value for a **uniqueidentifier** variable or column. Use the NEWID() function to obtain a globally unique ID (GUID).

Inserting Values into User-Defined Type Columns

You can insert values in user-defined type columns by:

- Supplying a value of the user-defined type.
- 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 insert a value in a column of user-defined type `Point`, by explicitly converting from a string.

```
INSERT INTO Cities (Location)

VALUES ( CONVERT(Point, '12.3:46.2') );
```

A binary value can also be supplied without performing explicit conversion, because all user-defined types are implicitly convertible from binary.

- Calling a user-defined function that returns a value of the user-defined type. The following example uses a user-defined function `CreateNewPoint()` to create a new value of user-defined type `Point` and insert the value into the `Cities` table.

```
INSERT INTO Cities (Location)

VALUES ( dbo.CreateNewPoint(x, y) );
```

Error Handling

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

If an INSERT statement violates a constraint or rule, or if it has a value incompatible with the data type of the column, the statement fails and an error message is returned.

If INSERT is loading multiple rows with SELECT or EXECUTE, any violation of a rule or constraint that occurs from the values being loaded causes the statement to be stopped, and no rows are loaded.

When an INSERT 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 to ON. The batch is stopped, and an error message is returned. During expression evaluation when SET ARITHABORT and SET ANSI_WARNINGS are OFF, if an INSERT, DELETE or UPDATE statement encounters an arithmetic error, overflow, divide-by-zero, or a domain error, SQL Server inserts or updates a NULL value. If the target column is not nullable, the insert or update action fails and the user receives an error.

Interoperability

When an INSTEAD OF trigger is defined on INSERT actions against a table or view, the trigger executes instead of the INSERT statement. For more information about INSTEAD OF triggers, see CREATE TRIGGER (Transact-SQL).

Limitations and Restrictions

When you insert values into remote tables and not all values for all columns are specified, you must identify the columns to which the specified values are to be inserted.

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

Locking Behavior

An INSERT 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; 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 INSERT 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 INSERT statement is always fully logged except when using the OPENROWSET function with the BULK keyword or when using INSERT INTO <target_table> SELECT <columns> FROM <source_table>. These operations can be minimally logged. For more information, see the section "Best Practices for Bulk Loading Data" earlier in this topic.

Security

During a linked server connection, the sending server provides a login name and password to connect to the receiving server on its behalf. For this connection to work, you must create a login mapping between the linked servers by using sp_addlinkedsrvlogin.

When you use OPENROWSET(BULK…), it is important to understand how SQL Server handles impersonation. For more information, see "Security Considerations" in Importing Bulk Data by Using BULK INSERT or OPENROWSET(BULK...).

Permissions

INSERT permission is required on the target table.

INSERT 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.

To execute INSERT with the OPENROWSET function BULK option, you must be a member of the **sysadmin** fixed server role or of the **bulkadmin** fixed server role.

Examples

Category	Featured syntax elements
Basic syntax	INSERT • table value constructor
Handling column values	IDENTITY • NEWID • default values • user-defined types
Inserting data from other tables	INSERT…SELECT • INSERT…EXECUTE •

Category	Featured syntax elements
	WITH common table expression • TOP • OFFSET FETCH
Specifying target objects other than standard tables	Views • table variables
Inserting rows into a remote table	Linked server • OPENQUERY rowset function • OPENDATASOURCE rowset function
Bulk loading data from tables or data files	INSERT…SELECT • OPENROWSET function
Overriding the default behavior of the query optimizer by using hints	Table hints
Capturing the results of the INSERT statement	OUTPUT clause

Basic Syntax

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

A. Inserting a single row of data

The following example inserts one row into the `Production.UnitMeasure` table. The columns in this table are `UnitMeasureCode`, `Name`, and `ModifiedDate`. Because values for all columns are supplied and are listed in the same order as the columns in the table, the column names do not have to be specified in the column list.

```
USE AdventureWorks2012;
GO
INSERT INTO Production.UnitMeasure
VALUES (N'FT', N'Feet', '20080414');
GO
```

B. Inserting multiple rows of data

The following example uses the table value constructor to insert three rows into the `Production.UnitMeasure` table in a single INSERT statement. Because values for all columns are supplied and are listed in the same order as the columns in the table, the column names do not have to be specified in the column list.

```
USE AdventureWorks2012;
GO
INSERT INTO Production.UnitMeasure
VALUES (N'FT2', N'Square Feet ', '20080923'), (N'Y', N'Yards', '20080923'), (N'Y3',
N'Cubic Yards', '20080923');
```

```
GO
```

C. Inserting data that is not in the same order as the table columns

The following example uses a column list to explicitly specify the values that are inserted into each column. The column order in the `Production.UnitMeasure` table is `UnitMeasureCode`, `Name`, `ModifiedDate`; however, the columns are not listed in that order in column_list.

```
USE AdventureWorks2012;
GO
INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode,
    ModifiedDate)
VALUES (N'Square Yards', N'Y2', GETDATE());
GO
```

Handling Column Values

Examples in this section demonstrate methods of inserting values into columns that are defined with an IDENTITY property, DEFAULT value, or are defined with data types such as **uniqueidentifer** or user-defined type columns.

A. Inserting data into a table with columns that have default values

The following example shows inserting rows into a table with columns that automatically generate a value or have a default value. `Column_1` is a computed column that automatically generates a value by concatenating a string with the value inserted into `column_2`. `Column_2` is defined with a default constraint. If a value is not specified for this column, the default value is used. `Column_3` is defined with the **rowversion** data type, which automatically generates a unique, incrementing binary number. `Column_4` does not automatically generate a value. When a value for this column is not specified, NULL is inserted. The INSERT statements insert rows that contain values for some of the columns but not all. In the last INSERT statement, no columns are specified and only the default values are inserted by using the DEFAULT VALUES clause.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.T1', 'U') IS NOT NULL
    DROP TABLE dbo.T1;
GO
CREATE TABLE dbo.T1
(
    column_1 AS 'Computed column ' + column_2,
    column_2 varchar(30)
        CONSTRAINT default_name DEFAULT ('my column default'),
    column_3 rowversion,
    column_4 varchar(40) NULL
);
GO
INSERT INTO dbo.T1 (column_4)
```

```
    VALUES ('Explicit value');
INSERT INTO dbo.T1 (column_2, column_4)
    VALUES ('Explicit value', 'Explicit value');
INSERT INTO dbo.T1 (column_2)
    VALUES ('Explicit value');
INSERT INTO T1 DEFAULT VALUES;
GO
SELECT column_1, column_2, column_3, column_4
FROM dbo.T1;
GO
```

B. Inserting data into a table with an identity column

The following example shows different methods of inserting data into an identity column. The first two INSERT statements allow identity values to be generated for the new rows. The third INSERT statement overrides the IDENTITY property for the column with the SET IDENTITY_INSERT statement and inserts an explicit value into the identity column.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.T1', 'U') IS NOT NULL
    DROP TABLE dbo.T1;
GO
CREATE TABLE dbo.T1 ( column_1 int IDENTITY, column_2 VARCHAR(30));
GO
INSERT T1 VALUES ('Row #1');
INSERT T1 (column_2) VALUES ('Row #2');
GO
SET IDENTITY_INSERT T1 ON;
GO
INSERT INTO T1 (column_1,column_2)
    VALUES (-99, 'Explicit identity value');
GO
SELECT column_1, column_2
FROM T1;
GO
```

C. Inserting data into a uniqueidentifier column by using NEWID()

The following example uses the NEWID() function to obtain a GUID for `column_2`. Unlike for identity columns, the Database Engine does not automatically generate values for columns with the uniqueidentifier data type, as shown by the second `INSERT` statement.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.T1', 'U') IS NOT NULL
    DROP TABLE dbo.T1;
```

```
GO
CREATE TABLE dbo.T1
(
    column_1 int IDENTITY,
    column_2 uniqueidentifier,
);
GO
INSERT INTO dbo.T1 (column_2)
    VALUES (NEWID());
INSERT INTO T1 DEFAULT VALUES;
GO
SELECT column_1, column_2
FROM dbo.T1;
GO
```

D. Inserting data into user-defined type columns

The following Transact-SQL statements insert three rows into the `PointValue` column of the `Points` table. This column uses a CLR user-defined type (UDT). The `Point` data type consists of X and Y integer values that are exposed as properties of the UDT. You must use either the CAST or CONVERT function to cast the comma-delimited X and Y values to the `Point` type. The first two statements use the CONVERT function to convert a string value to the `Point` type, and the third statement uses the CAST function. For more information, see Manipulating UDT Data.

```
INSERT INTO dbo.Points (PointValue) VALUES (CONVERT(Point, '3,4'));
INSERT INTO dbo.Points (PointValue) VALUES (CONVERT(Point, '1,5'));
INSERT INTO dbo.Points (PointValue) VALUES (CAST ('1,99' AS Point));
```

Inserting Data from Other Tables

Examples in this section demonstrate methods of inserting rows from one table into another table.

A. Using the SELECT and EXECUTE options to insert data from other tables

The following example shows how to insert data from one table into another table by using INSERT…SELECT or INSERT…EXECUTE. Each is based on a multi-table SELECT statement that includes an expression and a literal value in the column list.

The first INSERT statement uses a SELECT statement to derive the data from the source tables (`Employee`, `SalesPerson`, and `Person`) and store the result set in the `EmployeeSales` table. The second INSERT statement uses the EXECUTE clause to call a stored procedure that contains the SELECT statement, and the third INSERT uses the EXECUTE clause to reference the SELECT statement as a literal string.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.EmployeeSales', 'U') IS NOT NULL
    DROP TABLE dbo.EmployeeSales;
```

```
GO
IF OBJECT_ID ('dbo.uspGetEmployeeSales', 'P') IS NOT NULL
    DROP PROCEDURE uspGetEmployeeSales;
GO
CREATE TABLE dbo.EmployeeSales
( DataSource   varchar(20) NOT NULL,
  BusinessEntityID   varchar(11) NOT NULL,
  LastName     varchar(40) NOT NULL,
  SalesDollars money NOT NULL
);
GO
CREATE PROCEDURE dbo.uspGetEmployeeSales
AS
    SET NOCOUNT ON;
    SELECT 'PROCEDURE', sp.BusinessEntityID, c.LastName,
        sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.BusinessEntityID LIKE '2%'
    ORDER BY sp.BusinessEntityID, c.LastName;
GO
--INSERT...SELECT example
INSERT INTO dbo.EmployeeSales
    SELECT 'SELECT', sp.BusinessEntityID, c.LastName, sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.BusinessEntityID LIKE '2%'
    ORDER BY sp.BusinessEntityID, c.LastName;
GO
--INSERT...EXECUTE procedure example
INSERT INTO dbo.EmployeeSales
EXECUTE dbo.uspGetEmployeeSales;
GO
--INSERT...EXECUTE('string') example
INSERT INTO dbo.EmployeeSales
EXECUTE
('
SELECT ''EXEC STRING'', sp.BusinessEntityID, c.LastName,
    sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
```

```
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.BusinessEntityID LIKE ''2%''
    ORDER BY sp.BusinessEntityID, c.LastName
');
GO
--Show results.
SELECT DataSource,BusinessEntityID,LastName,SalesDollars
FROM dbo.EmployeeSales;
GO
```

B. Using WITH common table expression to define the data inserted

The following example creates the `NewEmployee` table. A common table expression (`EmployeeTemp`) defines the rows from one or more tables to be inserted into the `NewEmployee` table. The INSERT statement references the columns in the common table expression.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'HumanResources.NewEmployee', N'U') IS NOT NULL
    DROP TABLE HumanResources.NewEmployee;
GO
CREATE TABLE HumanResources.NewEmployee
(
    EmployeeID int NOT NULL,
    LastName nvarchar(50) NOT NULL,
    FirstName nvarchar(50) NOT NULL,
    PhoneNumber Phone NULL,
    AddressLine1 nvarchar(60) NOT NULL,
    City nvarchar(30) NOT NULL,
    State nchar(3) NOT NULL,
    PostalCode nvarchar(15) NOT NULL,
    CurrentFlag Flag
);
GO
WITH EmployeeTemp (EmpID, LastName, FirstName, Phone,
                   Address, City, StateProvince,
                   PostalCode, CurrentFlag)
AS (SELECT
       e.BusinessEntityID, c.LastName, c.FirstName, pp.PhoneNumber,
       a.AddressLine1, a.City, sp.StateProvinceCode,
       a.PostalCode, e.CurrentFlag
    FROM HumanResources.Employee e
        INNER JOIN Person.BusinessEntityAddress AS bea
        ON e.BusinessEntityID = bea.BusinessEntityID
        INNER JOIN Person.Address AS a
        ON bea.AddressID = a.AddressID
        INNER JOIN Person.PersonPhone AS pp
        ON e.BusinessEntityID = pp.BusinessEntityID
        INNER JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
        INNER JOIN Person.Person as c
        ON e.BusinessEntityID = c.BusinessEntityID
    )
INSERT INTO HumanResources.NewEmployee
    SELECT EmpID, LastName, FirstName, Phone,
           Address, City, StateProvince, PostalCode, CurrentFlag
    FROM EmployeeTemp;
GO
```

C. Using TOP to limit the data inserted from the source table

The following example creates the table `EmployeeSales` and inserts the name and year-to-date sales data for the top 5 random employees from the table `HumanResources.Employee`. The INSERT statement chooses any 5 rows returned by the `SELECT` statement. The OUTPUT clause displays the rows that are inserted into the `EmployeeSales` table. Notice that the ORDER BY clause in the SELECT statement is not used to determine the top 5 employees.

```
USE AdventureWorks2012 ;
GO
IF OBJECT_ID ('dbo.EmployeeSales', 'U') IS NOT NULL
    DROP TABLE dbo.EmployeeSales;
GO
CREATE TABLE dbo.EmployeeSales
( EmployeeID   nvarchar(11) NOT NULL,
  LastName     nvarchar(20) NOT NULL,
  FirstName    nvarchar(20) NOT NULL,
  YearlySales  money NOT NULL
 );
GO
INSERT TOP(5)INTO dbo.EmployeeSales
    OUTPUT inserted.EmployeeID, inserted.FirstName, inserted.LastName,
inserted.YearlySales
    SELECT sp.BusinessEntityID, c.LastName, c.FirstName, sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.SalesYTD > 250000.00
    ORDER BY sp.SalesYTD DESC;
```

If you have to use TOP to insert rows in a meaningful chronological order, you must use TOP together with ORDER BY in a subselect statement as shown in the following example. The OUTPUT clause displays the rows that are inserted into the `EmployeeSales` table. Notice that the top 5 employees are now inserted based on the results of the ORDER BY clause instead of random rows.

```
INSERT INTO dbo.EmployeeSales
    OUTPUT inserted.EmployeeID, inserted.FirstName, inserted.LastName,
inserted.YearlySales
    SELECT TOP (5) sp.BusinessEntityID, c.LastName, c.FirstName, sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.SalesYTD > 250000.00
    ORDER BY sp.SalesYTD DESC;
```

Specifying Target Objects Other Than Standard Tables

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

A. Inserting data by specifying a view

The following example specifies a view name as the target object; however, the new row is inserted in the underlying base table. The order of the values in the `INSERT` statement must match the column order of the view. For more information, see Modifying Data Through a View.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.T1', 'U') IS NOT NULL
    DROP TABLE dbo.T1;
GO
IF OBJECT_ID ('dbo.V1', 'V') IS NOT NULL
    DROP VIEW dbo.V1;
GO
CREATE TABLE T1 ( column_1 int, column_2 varchar(30));
GO
CREATE VIEW V1 AS
SELECT column_2, column_1
FROM T1;
GO
INSERT INTO V1
    VALUES ('Row 1',1);
GO
SELECT column_1, column_2
FROM T1;
GO
```

```
SELECT column_1, column_2
FROM V1;
GO
```

B. Inserting data into a table variable

The following example specifies a table variable as the target object.

```
USE AdventureWorks2012;
GO
-- Create the table variable.
DECLARE @MyTableVar table(
    LocationID int NOT NULL,
    CostRate smallmoney NOT NULL,
    NewCostRate AS CostRate * 1.5,
    ModifiedDate datetime);

-- Insert values into the table variable.
INSERT INTO @MyTableVar (LocationID, CostRate, ModifiedDate)
    SELECT LocationID, CostRate, GETDATE() FROM Production.Location
    WHERE CostRate > 0;

-- View the table variable result set.
SELECT * FROM @MyTableVar;
GO
```

Inserting Rows into a Remote Table

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

A. Inserting data into a remote table by using a linked server

The following example inserts rows into 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*.

```
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


USE AdventureWorks2012;
GO
-- Specify the remote data source in the FROM clause using a four-part name
-- in the form linked_server.catalog.schema.object.

INSERT INTO MyLinkServer.AdventureWorks2012.HumanResources.Department (Name, GroupName)
VALUES (N'Public Relations', N'Executive General and Administration');
GO
```

B. Inserting data into a remote table by using the OPENQUERY function

The following example inserts a row into a remote table by specifying the OPENQUERYrowset function. The linked server name created in the previous example is used in this example.

```
-- Use the OPENQUERY function to access the remote data source.

INSERT OPENQUERY (MyLinkServer, 'SELECT Name, GroupName FROM
AdventureWorks2012.HumanResources.Department')
VALUES ('Environmental Impact', 'Engineering');
GO
```

C. Inserting data into a remote table by using the OPENDATASOURCE function

The following example inserts a row into a remote table by specifying the OPENDATASOURCErowset function. Specify a valid server name for the data source by using the format *server_name* or *server_name\instance_name*.

```
-- Use the OPENDATASOURCE function to specify the remote data source.
-- Specify a valid server name for Data Source using the format server_name or
server_name\instance_name.

INSERT INTO OPENDATASOURCE('SQLNCLI',
    'Data Source= <server_name>; Integrated Security=SSPI')
    .AdventureWorks2012.HumanResources.Department (Name, GroupName)
    VALUES (N'Standards and Methods', 'Quality Assurance');
GO
```

Bulk Loading Data from Tables or Data Files

Examples in this section demonstrate two methods to bulk load data into a table by using the INSERT statement.

A. Inserting data into a heap with minimal logging

The following example creates aa new table (a heap) and inserts data from another table into it using minimal logging. The example assumes that the recovery model of the `AdventureWorks2012` database is set to FULL. To ensure minimal logging is used, the recovery model of the `AdventureWorks2012` database is set to BULK_LOGGED before rows are inserted and reset to FULL after the INSERT INTO…SELECT statement. In addition, the TABLOCK hint is specified for the target table `Sales.SalesHistory`. This ensures that the statement uses minimal space in the transaction log and performs efficiently.

```
USE AdventureWorks2012;
GO
-- Create the target heap.
CREATE TABLE Sales.SalesHistory(
    SalesOrderID int NOT NULL,
    SalesOrderDetailID int NOT NULL,
    CarrierTrackingNumber nvarchar(25) NULL,
    OrderQty smallint NOT NULL,
    ProductID int NOT NULL,
    SpecialOfferID int NOT NULL,
    UnitPrice money NOT NULL,
    UnitPriceDiscount money NOT NULL,
    LineTotal money NOT NULL,
    rowguid uniqueidentifier ROWGUIDCOL  NOT NULL,
    ModifiedDate datetime NOT NULL );
GO
-- Temporarily set the recovery model to BULK_LOGGED.
ALTER DATABASE AdventureWorks2012
SET RECOVERY BULK_LOGGED;
GO
-- Transfer data from Sales.SalesOrderDetail to Sales.SalesHistory
INSERT INTO Sales.SalesHistory WITH (TABLOCK)
    (SalesOrderID,
     SalesOrderDetailID,
     CarrierTrackingNumber,
     OrderQty,
     ProductID,
     SpecialOfferID,
     UnitPrice,
     UnitPriceDiscount,
     LineTotal,
     rowguid,
     ModifiedDate)
SELECT * FROM Sales.SalesOrderDetail;
GO
```

```
-- Reset the recovery model.
ALTER DATABASE AdventureWorks2012
SET RECOVERY FULL;
GO
```

B. Using the OPENROWSET function with BULK to bulk load data into a table

The following example inserts rows from a data file into a table by specifying the OPENROWSET function. The IGNORE_TRIGGERS table hint is specified for performance optimization. For more examples, see Importing Bulk Data by Using BULK INSERT or OPENROWSET(BULK...).

```
-- Use the OPENROWSET function to specify the data source and specifies the
IGNORE_TRIGGERS table hint.
INSERT INTO HumanResources.Department WITH (IGNORE_TRIGGERS) (Name, GroupName)
SELECT b.Name, b.GroupName
FROM OPENROWSET (
    BULK 'C:\SQLFiles\DepartmentData.txt',
    FORMATFILE = 'C:\SQLFiles\BulkloadFormatFile.xml',
    ROWS_PER_BATCH = 15000)AS b ;
GO
```

Overriding the Default Behavior of the Query Optimizer by Using Hints

Examples in this section demonstrate how to use table hints to temporarily override the default behavior of the query optimizer when processing the INSERT 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. Using the TABLOCK hint to specify a locking method

The following example specifies that an exclusive (X) lock is taken on the Production.Location table and is held until the end of the INSERT statement.

```
USE AdventureWorks2012;
GO
INSERT INTO Production.Location WITH (XLOCK)
(Name, CostRate, Availability)
VALUES ( N'Final Inventory', 15.00, 80.00);
GO
```

Capturing the Results of the INSERT 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 INSERT 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 OUTPUT with an INSERT statement

The following example inserts a row into the ScrapReason table and uses the OUTPUT clause to return the results of the statement to the @MyTableVar table variable. Because the ScrapReasonID column is defined with an IDENTITY property, a value is not specified in the INSERT statement for that column. However, note that the value generated by the Database Engine for that column is returned in the OUTPUT clause in the INSERTED.ScrapReasonID column.

```
USE AdventureWorks2012;
GO
DECLARE @MyTableVar table( NewScrapReasonID smallint,
                           Name varchar(50),
                           ModifiedDate datetime);
INSERT Production.ScrapReason
    OUTPUT INSERTED.ScrapReasonID, INSERTED.Name, INSERTED.ModifiedDate
        INTO @MyTableVar
VALUES (N'Operator error', GETDATE());

--Display the result set of the table variable.
SELECT NewScrapReasonID, Name, ModifiedDate FROM @MyTableVar;
--Display the result set of the table.
SELECT ScrapReasonID, Name, ModifiedDate
FROM Production.ScrapReason;
GO
```

B. Using OUTPUT with identity and computed columns

The following example creates the EmployeeSales table and then inserts several rows into it using an INSERT statement with a SELECT statement to retrieve data from source tables. The EmployeeSales table contains an identity column (EmployeeID) and a computed column (ProjectedSales). Because these values are generated by the Database Engine during the insert operation, neither of these columns can be defined in @MyTableVar.

```
USE AdventureWorks2012 ;
GO
IF OBJECT_ID ('dbo.EmployeeSales', 'U') IS NOT NULL
    DROP TABLE dbo.EmployeeSales;
GO
CREATE TABLE dbo.EmployeeSales
( EmployeeID   int IDENTITY (1,5)NOT NULL,
  LastName     nvarchar(20) NOT NULL,
  FirstName    nvarchar(20) NOT NULL,
  CurrentSales money NOT NULL,
  ProjectedSales AS CurrentSales * 1.10
);
GO
DECLARE @MyTableVar table(
  LastName     nvarchar(20) NOT NULL,
  FirstName    nvarchar(20) NOT NULL,
  CurrentSales money NOT NULL
  );

INSERT INTO dbo.EmployeeSales (LastName, FirstName, CurrentSales)
  OUTPUT INSERTED.LastName,
         INSERTED.FirstName,
         INSERTED.CurrentSales
  INTO @MyTableVar
    SELECT c.LastName, c.FirstName, sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.BusinessEntityID LIKE '2%'
    ORDER BY c.LastName, c.FirstName;

SELECT LastName, FirstName, CurrentSales
FROM @MyTableVar;
GO
SELECT EmployeeID, LastName, FirstName, CurrentSales, ProjectedSales
FROM dbo.EmployeeSales;
GO
```

C. Inserting data returned from an OUTPUT clause

The following example captures data returned from the OUTPUT clause of a MERGE statement, and inserts that data into another table. The MERGE statement updates the `Quantity` column of the `ProductInventory` table daily, based on orders that are processed in the `SalesOrderDetail` table. It also deletes rows for products whose inventories drop to 0. The example captures the rows that are deleted and inserts them into another table, `ZeroInventory`, which tracks products with no inventory.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID(N'Production.ZeroInventory', N'U') IS NOT NULL
    DROP TABLE Production.ZeroInventory;
GO
--Create ZeroInventory table.
CREATE TABLE Production.ZeroInventory (DeletedProductID int, RemovedOnDate DateTime);
GO
```

```
INSERT INTO Production.ZeroInventory (DeletedProductID, RemovedOnDate)
SELECT ProductID, GETDATE()
FROM
(   MERGE Production.ProductInventory AS pi
    USING (SELECT ProductID, SUM(OrderQty) FROM Sales.SalesOrderDetail AS sod
           JOIN Sales.SalesOrderHeader AS soh
           ON sod.SalesOrderID = soh.SalesOrderID
           AND soh.OrderDate = '20070401'
           GROUP BY ProductID) AS src (ProductID, OrderQty)
    ON (pi.ProductID = src.ProductID)
    WHEN MATCHED AND pi.Quantity - src.OrderQty <= 0
        THEN DELETE
    WHEN MATCHED
        THEN UPDATE SET pi.Quantity = pi.Quantity - src.OrderQty
    OUTPUT $action, deleted.ProductID) AS Changes (Action, ProductID)
WHERE Action = 'DELETE';
IF @@ROWCOUNT = 0
PRINT 'Warning: No rows were inserted';
GO
SELECT DeletedProductID, RemovedOnDate FROM Production.ZeroInventory;
```

See Also

BULK INSERT (Transact-SQL)

DELETE (Transact-SQL)

EXECUTE (Transact-SQL)

FROM (Transact-SQL)

IDENTITY (Property) (Transact-SQL)

NEWID (Transact-SQL)

SELECT (Transact-SQL)

UPDATE (Transact-SQL)

MERGE (Transact-SQL)

OUTPUT Clause (Transact-SQL)

Using the inserted and deleted Tables

MERGE

Performs insert, update, or delete operations on a target table based on the results of a join with a source table. For example, you can synchronize two tables by inserting, updating, or deleting rows in one table based on differences found in the other table.

Transact-SQL Syntax Conventions

Syntax

```
[ WITH <common_table_expression> [,...n] ]
MERGE
    [ TOP ( expression ) [ PERCENT ] ]
    [ INTO ] <target_table> [ WITH ( <merge_hint> ) ] [ [ AS ] table_alias ]
    USING <table_source>
    ON <merge_search_condition>
    [ WHEN MATCHED [ AND <clause_search_condition> ]
        THEN <merge_matched> ] [ ...n ]
    [ WHEN NOT MATCHED [ BY TARGET ] [ AND <clause_search_condition> ]
        THEN <merge_not_matched> ]
    [ WHEN NOT MATCHED BY SOURCE [ AND <clause_search_condition> ]
        THEN <merge_matched> ] [ ...n  ]
    [ <output_clause> ]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
;

<target_table> ::=
{
    [ database_name . schema_name . | schema_name . ]
  target_table
}

<merge_hint>::=
{
    { [ <table_hint_limited> [ ,...n ] ]
    [ [ , ] INDEX ( index_val [ ,...n ] ) ] }
}

<table_source> ::=
{
    table_or_view_name [ [ AS ] table_alias ] [ <tablesample_clause> ]
        [ WITH ( table_hint [ [ , ]...n ] ) ]
  | rowset_function [ [ AS ] table_alias ]
      [ ( bulk_column_alias [ ,...n ] ) ]
  | user_defined_function [ [ AS ] table_alias ]
  | OPENXML <openxml_clause>
  | derived_table [ AS ] table_alias [ ( column_alias [ ,...n ] ) ]
  | <joined_table>
  | <pivoted_table>
  | <unpivoted_table>
}

<merge_search_condition> ::=
    <search_condition>

<merge_matched>::=
    { UPDATE SET <set_clause> | DELETE }

<set_clause>::=
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 ]

<merge_not_matched>::=
{
    INSERT [ (column_list) ]
      { VALUES (values_list)
      | DEFAULT VALUES }
```

```
}

<clause_search_condition> ::=
    <search_condition>

<search condition> ::=
    { [ NOT ] <predicate> | ( <search_condition> ) }
    [ { AND | OR } [ NOT ] { <predicate> | ( <search_condition> ) } ]
[ ,...n ]

<predicate> ::=
    { expression { = | <> | ! = | > | > = | ! > | < | < = | ! < } expression
    | string_expression [ NOT ] LIKE string_expression
  [ ESCAPE 'escape_character' ]
    | expression [ NOT ] BETWEEN expression AND expression
    | expression IS [ NOT ] NULL
    | CONTAINS
  ( { column | * } ,'< contains_search_condition >' )
    | FREETEXT ( { column | * } , 'freetext_string')
    | expression [ NOT ] IN (subquery | expression [ ,...n ] )
    | expression { = | <> | ! = | > | > = | ! > | < | < = | ! < }
  { ALL | SOME | ANY} (subquery)
    | EXISTS (subquery) }

<output_clause>::=
{
    [ OUTPUT <dml_select_list> INTO { @table_variable | output_table }
        [ (column_list) ] ]
    [ OUTPUT <dml_select_list> ]
}

<dml_select_list>::=
    { <column_name> | scalar_expression }
        [ [AS] column_alias_identifier ] [ ,...n ]

<column_name> ::=
    { DELETED | INSERTED | from_table_name } . { * | column_name }
    | $action
```

Arguments

WITH <common_table_expression>

Specifies the temporary named result set or view, also known as common table expression, defined within the scope of the MERGE statement. The result set is derived from a simple query and is referenced by the MERGE statement. For more information, see WITH common_table_expression (Transact-SQL).

TOP (expression) [PERCENT]

Specifies the number or percentage of rows that are affected. expression can be either a number or a percentage of the rows. The rows referenced in the TOP expression are not arranged in any order. For more information, see TOP (Transact-SQL).

The TOP clause is applied after the entire source table and the entire target table are joined and the joined rows that do not qualify for an insert, update, or delete action are removed. The TOP clause further reduces the number of joined rows to the specified value and the insert, update, or delete actions are applied to the remaining joined rows in an unordered fashion. That is, there is no order in which the rows are distributed among the actions defined in the WHEN clauses. For example, specifying TOP (10) affects 10 rows; of these rows, 7 may be updated and 3 inserted, or 1 may be deleted, 5 updated, and 4 inserted and so on.

Because the MERGE statement performs a full table scan of both the source and target tables, I/O performance can be affected when using the TOP clause to modify a large table by creating multiple batches. In this scenario, it is important to ensure that all successive batches target new rows.

database_name

Is the name of the database in which target_table is located.

schema_name

Is the name of the schema to which target_table belongs.

target_table

Is the table or view against which the data rows from <table_source> are matched based on <clause_search_condition>.target_table is the target of any insert, update, or delete operations specified by the WHEN clauses of the MERGE statement.

If target_table is a view, any actions against it must satisfy the conditions for updating views. For more information, see Modifying Data Through a View.

target_table cannot be a remote table. target_table cannot have any rules defined on it.

[AS] table_alias

Is an alternative name used to reference a table.

USING <table_source>

Specifies the data source that is matched with the data rows in target_table based on <merge_search condition>. The result of this match dictates the actions to take by the WHEN clauses of the MERGE statement. <table_source> can be a remote table or a derived table that accesses remote tables.

<table_source> can be a derived table that uses the Transact-SQL table value constructor to construct a table by specifying multiple rows.

For more information about the syntax and arguments of this clause, see FROM (Transact-SQL).

ON <merge_search_condition>

Specifies the conditions on which <table_source> is joined with target_table to determine where they match.

Caution

It is important to specify only the columns from the target table that are used for matching purposes. That is, specify columns from the target table that are compared to the corresponding column of the source table. Do not attempt to improve query performance by filtering out rows in the target table in the ON clause, such as by specifying `AND NOT target_table.column_x = value`. Doing so may return unexpected and incorrect results.

WHEN MATCHED THEN <merge_matched>

Specifies that all rows of target_table that match the rows returned by <table_source> ON <merge_search_condition>, and satisfy any additional search condition, are either updated or deleted according to the <merge_matched> clause.

The MERGE statement can have at most two WHEN MATCHED clauses. If two clauses are specified, then the first clause must be accompanied by an AND <search_condition> clause. For any given row, the second WHEN MATCHED clause is only applied if the first is not. If there are two WHEN MATCHED clauses, then one must specify an UPDATE action and one must specify a DELETE action. If UPDATE is specified in the <merge_matched> clause, and more than one row of <table_source>matches a row in target_table based on <merge_search_condition>, SQL Server returns an error. The MERGE statement cannot update the same row more than once, or update and delete the same row.

WHEN NOT MATCHED [BY TARGET] THEN <merge_not_matched>

Specifies that a row is inserted into target_table for every row returned by <table_source> ON <merge_search_condition> that does not match a row in target_table, but does satisfy an additional search condition, if present. The values to insert are specified by the <merge_not_matched> clause. The MERGE statement can have only one WHEN NOT MATCHED clause.

WHEN NOT MATCHED BY SOURCE THEN <merge_matched>

Specifies that all rows of target_table that do not match the rows returned by <table_source> ON <merge_search_condition>, and that satisfy any additional search condition, are either updated or deleted according to the <merge_matched> clause.

The MERGE statement can have at most two WHEN NOT MATCHED BY SOURCE clauses. If two clauses are specified, then the first clause must be accompanied by an AND <clause_search_condition> clause. For any given row, the second WHEN NOT MATCHED BY SOURCE clause is only applied if the first is not. If there are two WHEN NOT MATCHED BY SOURCE clauses, then one must specify an UPDATE action and one must specify a DELETE action. Only columns from the target table can be referenced in <clause_search_condition>.

When no rows are returned by <table_source>, columns in the source table cannot be accessed. If the update or delete action specified in the <merge_matched> clause references columns in the source table, error 207 (Invalid column name) is returned. For example, the clause `WHEN NOT MATCHED BY SOURCE THEN UPDATE SET TargetTable.Col1 = SourceTable.Col1` may cause the statement to fail because `Col1` in the source table is inaccessible.

AND <clause_search_condition>

Specifies any valid search condition. For more information, see Search Condition (Transact-SQL).

<table_hint_limited>

Specifies one or more table hints that are applied on the target table for each of the insert, update, or delete actions that are performed by the MERGE statement. The WITH keyword and the parentheses are required.

NOLOCK and READUNCOMMITTED are not allowed. For more information about table hints, see Table Hint (Transact-SQL).

Specifying the TABLOCK hint on a table that is the target of an INSERT statement has the same effect as specifying the TABLOCKX hint. An exclusive lock is taken on the table. When FORCESEEK is specified, it is applied to the implicit instance of the target table joined with the source table.

Caution

Specifying READPAST with WHEN NOT MATCHED [BY TARGET] THEN INSERT may result in INSERT operations that violate UNIQUE constraints.

INDEX (index_val [,...n])

Specifies the name or ID of one or more indexes on the target table for performing an implicit join with the source table. For more information, see Table Hint (Transact-SQL).

<output_clause>

Returns a row for every row in target_table that is updated, inserted, or deleted, in no particular order. For more information about the arguments of this clause, see OUTPUT Clause (Transact-SQL).

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 Hint (Transact-SQL).

<merge_matched>

Specifies the update or delete action that is applied to all rows of target_table that do not match the rows returned by <table_source> ON <merge_search_condition>, and that satisfy any additional search condition.

UPDATE SET <set_clause>

Specifies the list of column or variable names to be updated in the target table and the values with which to update them.

For more information about the arguments of this clause, see UPDATE (Transact-SQL). Setting a variable to the same value as a column is not permitted.

DELETE

Specifies that the rows matching rows in target_table are deleted.

<merge_not_matched>

Specifies the values to insert into the target table.

(column_list)

Is a list of one or more columns of the target table in which to insert data. Columns must be specified as a single-part name or else the MERGE statement will fail. column_list must be enclosed in parentheses and delimited by commas.

VALUES (values_list)

Is a comma-separated list of constants, variables, or expressions that return values to insert into the target table. Expressions cannot contain an EXECUTE statement.

DEFAULT VALUES

Forces the inserted row to contain the default values defined for each column.

For more information about this clause, see INSERT (Transact-SQL).

<search condition>

Specifies the search conditions used to specify <merge_search_condition> or <clause_search_condition>. For more information about the arguments for this clause, see Search Condition (Transact-SQL).

Remarks

At least one of the three MATCHED clauses must be specified, but they can be specified in any order. A variable cannot be updated more than once in the same MATCHED clause.

Any insert, update, or delete actions specified on the target table by the MERGE statement are limited by any constraints defined on it, including any cascading referential integrity constraints. If IGNORE_DUP_KEY is set to ON for any unique indexes on the target table, MERGE ignores this setting.

The MERGE statement requires a semicolon (;) as a statement terminator. Error 10713 is raised when a MERGE statement is run without the terminator.

When used after MERGE, @@ROWCOUNT returns the total number of rows inserted, updated, and deleted to the client.

MERGE is a fully reserved keyword when the database compatibility level is set to 100. The MERGE statement is available under both 90 and 100 database compatibility levels; however the keyword is not fully reserved when the database compatibility level is set to 90.

Trigger Implementation

For every insert, update, or delete action specified in the MERGE statement, SQL Server fires any corresponding AFTER triggers defined on the target table, but does not guarantee on which action to fire triggers first or last. Triggers defined for the same action honor the order you specify. For more information about setting trigger firing order, see Specifying First and Last Triggers.

If the target table has an enabled INSTEAD OF trigger defined on it for an insert, update, or delete action performed by a MERGE statement, then it must have an enabled INSTEAD OF trigger for all of the actions specified in the MERGE statement.

If there are any INSTEAD OF UPDATE or INSTEAD OF DELETE triggers defined on target_table, the update or delete operations are not performed. Instead, the triggers fire and the **inserted** and **deleted** tables are populated accordingly.

If there are any INSTEAD OF INSERT triggers defined on target_table, the insert operation is not performed. Instead, the triggers fire and the **inserted** table is populated accordingly.

Permissions

Requires SELECT permission on the source table and INSERT, UPDATE, or DELETE permissions on the target table. For additional information, see the Permissions section in the SELECT, INSERT, UPDATE, and DELETE topics.

Examples

A. Using MERGE to perform INSERT and UPDATE operations on a table in a single statement

A common scenario is updating one or more columns in a table if a matching row exists, or inserting the data as a new row if a matching row does not exist. This is usually done by passing parameters to a stored procedure that contains the appropriate UPDATE and INSERT statements. With the MERGE statement, you can perform both tasks in a single statement. The following example shows a stored procedure that contains both an INSERT statement and an UPDATE statement. The procedure is then modified to perform the equivalent operations by using a single MERGE statement.

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE dbo.InsertUnitMeasure
    @UnitMeasureCode nchar(3),
    @Name nvarchar(25)
AS
BEGIN
    SET NOCOUNT ON;
-- Update the row if it exists.
    UPDATE Production.UnitMeasure
    SET Name = @Name
    WHERE UnitMeasureCode = @UnitMeasureCode
-- Insert the row if the UPDATE statement failed.
    IF (@@ROWCOUNT = 0 )
    BEGIN
        INSERT INTO Production.UnitMeasure (UnitMeasureCode, Name)
        VALUES (@UnitMeasureCode, @Name)
    END
END;
GO
-- Test the procedure and return the results.
EXEC InsertUnitMeasure @UnitMeasureCode = 'ABC', @Name = 'Test Value';
SELECT UnitMeasureCode, Name FROM Production.UnitMeasure
WHERE UnitMeasureCode = 'ABC';
GO

-- Rewrite the procedure to perform the same operations using the MERGE statement.
-- Create a temporary table to hold the updated or inserted values from the OUTPUT
clause.
CREATE TABLE #MyTempTable
    (ExistingCode nchar(3),
     ExistingName nvarchar(50),
     ExistingDate datetime,
     ActionTaken nvarchar(10),
     NewCode nchar(3),
     NewName nvarchar(50),
     NewDate datetime
    );
GO
ALTER PROCEDURE dbo.InsertUnitMeasure
    @UnitMeasureCode nchar(3),
    @Name nvarchar(25)
AS
BEGIN
    SET NOCOUNT ON;

    MERGE Production.UnitMeasure AS target
    USING (SELECT @UnitMeasureCode, @Name) AS source (UnitMeasureCode, Name)
    ON (target.UnitMeasureCode = source.UnitMeasureCode)
    WHEN MATCHED THEN
        UPDATE SET Name = source.Name
    WHEN NOT MATCHED THEN
        INSERT (UnitMeasureCode, Name)
        VALUES (source.UnitMeasureCode, source.Name)
        OUTPUT deleted.*, $action, inserted.* INTO #MyTempTable;
END;
GO
-- Test the procedure and return the results.
EXEC InsertUnitMeasure @UnitMeasureCode = 'ABC', @Name = 'New Test Value';
EXEC InsertUnitMeasure @UnitMeasureCode = 'XYZ', @Name = 'Test Value';
EXEC InsertUnitMeasure @UnitMeasureCode = 'ABC', @Name = 'Another Test Value';

SELECT * FROM #MyTempTable;
-- Cleanup
DELETE FROM Production.UnitMeasure WHERE UnitMeasureCode IN ('ABC','XYZ');
DROP TABLE #MyTempTable;
GO
```

B. Using MERGE to perform UPDATE and DELETE operations on a table in a single statement

The following example uses MERGE to update the `ProductInventory` table in the `AdventureWorks` sample database on a daily basis, based on orders that are processed in the `SalesOrderDetail` table. The `Quantity` column of the `ProductInventory` table is updated by subtracting the number of orders placed each day for each product in the `SalesOrderDetail` table. If the number of orders for a product drops the inventory level of a product to 0 or less, the row for that product is deleted from the `ProductInventory` table.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'Production.usp_UpdateInventory', N'P') IS NOT NULL DROP PROCEDURE
Production.usp_UpdateInventory;
GO
CREATE PROCEDURE Production.usp_UpdateInventory
    @OrderDate datetime
AS
MERGE Production.ProductInventory AS target
USING (SELECT ProductID, SUM(OrderQty) FROM Sales.SalesOrderDetail AS sod
    JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
    AND soh.OrderDate = @OrderDate
    GROUP BY ProductID) AS source (ProductID, OrderQty)
ON (target.ProductID = source.ProductID)
WHEN MATCHED AND target.Quantity - source.OrderQty <= 0
    THEN DELETE
WHEN MATCHED
    THEN UPDATE SET target.Quantity = target.Quantity - source.OrderQty,
                    target.ModifiedDate = GETDATE()
```

```
OUTPUT $action, Inserted.ProductID, Inserted.Quantity, Inserted.ModifiedDate,
Deleted.ProductID,
    Deleted.Quantity, Deleted.ModifiedDate;
GO

EXECUTE Production.usp_UpdateInventory '20030501'
```

C. Using MERGE to perform UPDATE and INSERT operations on a target table by using a derived source table

The following example uses MERGE to modify the `SalesReason` table by either updating or inserting rows. When the value of `NewName` in the source table matches a value in the `Name` column of the target table, (`SalesReason`), the `ReasonType` column is updated in the target table. When the value of `NewName` does not match, the source row is inserted into the target table. The source table is a derived table that uses the Transact-SQL table value constructor to specify multiple rows for the source table. For more information about using the table value constructor in a derived table, see Table Value Constructor (Transact-SQL). The example also shows how to store the results of the OUTPUT clause in a table variable and then summarize the results of the MERGE statment by performing a simple select operation that returns the count of inserted and updated rows.

```
USE AdventureWorks2012;
GO
-- Create a temporary table variable to hold the output actions.
DECLARE @SummaryOfChanges TABLE(Change VARCHAR(20));

MERGE INTO Sales.SalesReason AS Target
USING (VALUES ('Recommendation','Other'), ('Review', 'Marketing'), ('Internet',
'Promotion'))
       AS Source (NewName, NewReasonType)
ON Target.Name = Source.NewName
WHEN MATCHED THEN
    UPDATE SET ReasonType = Source.NewReasonType
WHEN NOT MATCHED BY TARGET THEN
    INSERT (Name, ReasonType) VALUES (NewName, NewReasonType)
OUTPUT $action INTO @SummaryOfChanges;

-- Query the results of the table variable.
SELECT Change, COUNT(*) AS CountPerChange
FROM @SummaryOfChanges
GROUP BY Change;
```

D. Inserting the results of the MERGE statement into another table

The following example captures data returned from the OUTPUT clause of a MERGE statement and inserts that data into another table. The MERGE statement updates the `Quantity` column of the `ProductInventory` table, based on orders that are processed in the `SalesOrderDetail` table. The example captures the rows that are updated and inserts them into another table that is used to track inventory changes.

```
USE AdventureWorks2012;
GO
CREATE TABLE Production.UpdatedInventory
    (ProductID INT NOT NULL, LocationID int, NewQty int, PreviousQty int,
     CONSTRAINT PK_Inventory PRIMARY KEY CLUSTERED (ProductID, LocationID));
GO
INSERT INTO Production.UpdatedInventory
SELECT ProductID, LocationID, NewQty, PreviousQty
FROM
(    MERGE Production.ProductInventory AS pi
     USING (SELECT ProductID, SUM(OrderQty)
            FROM Sales.SalesOrderDetail AS sod
            JOIN Sales.SalesOrderHeader AS soh
            ON sod.SalesOrderID = soh.SalesOrderID
            AND soh.OrderDate BETWEEN '20030701' AND '20030731'
            GROUP BY ProductID) AS src (ProductID, OrderQty)
     ON pi.ProductID = src.ProductID
    WHEN MATCHED AND pi.Quantity - src.OrderQty >= 0
        THEN UPDATE SET pi.Quantity = pi.Quantity - src.OrderQty
    WHEN MATCHED AND pi.Quantity - src.OrderQty <= 0
        THEN DELETE
    OUTPUT $action, Inserted.ProductID, Inserted.LocationID, Inserted.Quantity AS NewQty,
Deleted.Quantity AS PreviousQty)
 AS Changes (Action, ProductID, LocationID, NewQty, PreviousQty) WHERE Action = 'UPDATE';
GO
```

See Also

SELECT (Transact-SQL)

INSERT (Transact-SQL)

UPDATE (Transact-SQL)

DELETE (Transact-SQL)

OUTPUT Clause (Transact-SQL)

Using MERGE in Integration Services Packages

FROM (Transact-SQL)

Table Value Constructor (Transact-SQL)

OPTION Clause

Specifies that the indicated query hint should be used throughout the entire query. Each query hint can be specified only one time, although multiple query hints are permitted. Only one OPTION clause can be specified with the statement.

Transact-SQL Syntax Conventions

Syntax

```
[ OPTION (<query_hint> [ ,...n ] ) ]
```

See Also

Hints (Transact-SQL)

OUTPUT Clause

Returns information from, or expressions based on, each row affected by an INSERT, UPDATE, DELETE, or MERGE statement. These results can be returned to the processing application for use in such things as confirmation messages, archiving, and other such application requirements. The results can also be inserted into a table or table variable. Additionally, you can capture the results of an OUTPUT clause in a nested INSERT, UPDATE, DELETE, or MERGE statement, and insert those results into a target table or view.

Note

An UPDATE, INSERT, or DELETE statement that has an OUTPUT clause will return rows to the client even if the statement encounters errors and is rolled back. The result should not be used if any error occurs when you run the statement.

Used in:

DELETE

INSERT

UPDATE

MERGE

Transact-SQL Syntax Conventions

Syntax

```
<OUTPUT_CLAUSE> ::=
{
    [ OUTPUT <dml_select_list> INTO { @table_variable | output_table } [ (column_list) ] ]
    [ OUTPUT <dml_select_list> ]
}
<dml_select_list> ::=
{ <column_name> | scalar_expression } [ [AS] column_alias_identifier ]
    [ ,...n ]

<column_name> ::=
{ DELETED | INSERTED | from_table_name } . { * | column_name }
    | $action
```

Arguments

@table_variable

Specifies a **table** variable that the returned rows are inserted into instead of being returned to the caller. @table_variable must be declared before the INSERT, UPDATE, DELETE, or MERGE statement.

If column_list is not specified, the **table** variable must have the same number of columns as the OUTPUT result set. The exceptions are identity and computed columns, which must be skipped. If column_list is specified, any omitted columns must either allow null values or have default values assigned to them.

For more information about **table** variables, see table (Transact-SQL)).

output_table

Specifies a table that the returned rows are inserted into instead of being returned to the caller. output_table may be a temporary table.

If column_list is not specified, the table must have the same number of columns as the OUTPUT result set. The exceptions are identity and computed columns. These must be skipped. If column_list is specified, any omitted columns must either allow null values or have default values assigned to them.

output_table cannot:

- Have enabled triggers defined on it.
- Participate on either side of a FOREIGN KEY constraint.
- Have CHECK constraints or enabled rules.

column_list

Is an optional list of column names on the target table of the INTO clause. It is analogous to the column list allowed in the INSERT statement.

scalar_expression

Is any combination of symbols and operators that evaluates to a single value. Aggregate functions are not permitted in scalar_expression.

Any reference to columns in the table being modified must be qualified with the INSERTED or DELETED prefix.

column_alias_identifier

Is an alternative name used to reference the column name.

DELETED

Is a column prefix that specifies the value deleted by the update or delete operation. Columns prefixed with DELETED reflect the value before the UPDATE, DELETE, or MERGE statement is completed.

DELETED cannot be used with the OUTPUT clause in the INSERT statement.

INSERTED

Is a column prefix that specifies the value added by the insert or update operation. Columns prefixed with INSERTED reflect the value after the UPDATE, INSERT, or MERGE statement is completed but before triggers are executed.

INSERTED cannot be used with the OUTPUT clause in the DELETE statement.

from_table_name

Is a column prefix that specifies a table included in the FROM clause of a DELETE, UPDATE, or MERGE statement that is used to specify the rows to update or delete.

If the table being modified is also specified in the FROM clause, any reference to columns in that table must be qualified with the INSERTED or DELETED prefix.

*****

Specifies that all columns affected by the delete, insert, or update action will be returned in the order in which they exist in the table.

For example, `OUTPUT DELETED.*` in the following DELETE statement returns all columns deleted from the `ShoppingCartItem` table:

```
DELETE Sales.ShoppingCartItem

    OUTPUT DELETED.*;
```

column_name

Is an explicit column reference. Any reference to the table being modified must be correctly qualified by either the INSERTED or the DELETED prefix as appropriate, for example: INSERTED.column_name.

$action

Is available only for the MERGE statement. Specifies a column of type **nvarchar(10)** in the OUTPUT clause in a MERGE statement that returns one of three values for each row: 'INSERT', 'UPDATE', or 'DELETE', according to the action that was performed on that row.

Remarks

The OUTPUT <dml_select_list> clause and the OUTPUT <dml_select_list> INTO { @table_variable | output_table } clause can be defined in a single INSERT, UPDATE, DELETE, or MERGE statement.

Note

Unless specified otherwise, references to the OUTPUT clause refer to both the OUTPUT clause and the OUTPUT INTO clause.

The OUTPUT clause may be useful to retrieve the value of identity or computed columns after an INSERT or UPDATE operation.

When a computed column is included in the <dml_select_list>, the corresponding column in the output table or table variable is not a computed column. The values in the new column are the values that were computed at the time the statement was executed.

There is no guarantee that the order in which the changes are applied to the table and the order in which the rows are inserted into the output table or table variable will correspond.

If parameters or variables are modified as part of an UPDATE statement, the OUTPUT clause always returns the value of the parameter or variable as it was before the statement executed instead of the modified value.

You can use OUTPUT with an UPDATE or DELETE statement positioned on a cursor that uses WHERE CURRENT OF syntax.

The OUTPUT clause is not supported in the following statements:

- DML statements that reference local partitioned views, distributed partitioned views, or remote tables.
- INSERT statements that contain an EXECUTE statement.
- Full-text predicates are not allowed in the OUTPUT clause when the database compatibility level is set to 100.
- The OUTPUT INTO clause cannot be used to insert into a view, or rowset function.

- A user-defined function cannot be created if it contains an OUTPUT INTO clause that has a table as its target.

To prevent nondeterministic behavior, the OUTPUT clause cannot contain the following references:

- Subqueries or user-defined functions that perform user or system data access, or are assumed to perform such access. User-defined functions are assumed to perform data access if they are not schema-bound.
- A column from a view or inline table-valued function when that column is defined by one of the following methods:
  - Asubquery.
  - A user-defined function that performs user or system data access, or is assumed to perform such access.
  - A computed column that contains a user-defined function that performs user or system data access in its definition.

  When SQL Server detects such a column in the OUTPUT clause, error 4186 is raised. For more information, see MSSQLSERVER_4186.

Inserting Data Returned From an OUTPUT Clause Into a Table

When you are capturing the results of an OUTPUT clause in a nested INSERT, UPDATE, DELETE, or MERGE statement and inserting those results into a target table, keep the following information in mind:

- The whole operation is atomic. Either both the INSERT statement and the nested DML statement that contains the OUTPUT clause execute, or the whole statement fails.
- The following restrictions apply to the target of the outer INSERT statement:
  - The target cannot be a remote table, view, or common table expression.
  - The target cannot have a FOREIGN KEY constraint, or be referenced by a FOREIGN KEY constraint.
  - Triggers cannot be defined on the target.
  - The target cannot participate in merge replication or updatable subscriptions for transactional replication.
- The following restrictions apply to the nested DML statement:
  - The target cannot be a remote table or partitioned view.
  - The source itself cannot contain a <dml_table_source> clause.
- The OUTPUT INTO clause is not supported in INSERT statements that contain a <dml_table_source> clause.
- @@ROWCOUNT returns the rows inserted only by the outer INSERT statement.
- @@IDENTITY, SCOPE_IDENTITY, and IDENT_CURRENT return identity values generated only by the nested DML statement, and not those generated by the outer INSERT statement.

- Query notifications treat the statement as a single entity, and the type of any message that is created will be the type of the nested DML, even if the significant change is from the outer INSERT statement itself.
- In the <dml_table_source> clause, the SELECT and WHERE clauses cannot include subqueries, aggregate functions, ranking functions, full-text predicates, user-defined functions that perform data access, or the TEXTPTR function.

Triggers

Columns returned from OUTPUT reflect the data as it is after the INSERT, UPDATE, or DELETE statement has completed but before triggers are executed.

For INSTEAD OF triggers, the returned results are generated as if the INSERT, UPDATE, or DELETE had actually occurred, even if no modifications take place as the result of the trigger operation. If a statement that includes an OUTPUT clause is used inside the body of a trigger, table aliases must be used to reference the trigger inserted and deleted tables to avoid duplicating column references with the INSERTED and DELETED tables associated with OUTPUT.

If the OUTPUT clause is specified without also specifying the INTO keyword, the target of the DML operation cannot have any enabled trigger defined on it for the given DML action. For example, if the OUTPUT clause is defined in an UPDATE statement, the target table cannot have any enabled UPDATE triggers.

If the sp_configure option disallow results from triggers is set, an OUTPUT clause without an INTO clause causes the statement to fail when it is invoked from within a trigger.

Data Types

The OUTPUT clause supports the large object data types: **nvarchar(max)**, **varchar(max)**, **varbinary(max)**, **text**, **ntext**, **image**, and **xml**. When you use the .WRITE clause in the UPDATE statement to modify an **nvarchar(max)**, **varchar(max)**, or **varbinary(max)** column, the full before and after images of the values are returned if they are referenced. The TEXTPTR( ) function cannot appear as part of an expression on a **text**, **ntext**, or **image** column in the OUTPUT clause.

Queues

You can use OUTPUT in applications that use tables as queues, or to hold intermediate result sets. That is, the application is constantly adding or removing rows from the table. The following example uses the OUTPUT clause in a DELETE statement to return the deleted row to the calling application.

```
USE AdventureWorks2012;

GO

DELETE TOP(1) dbo.DatabaseLog WITH (READPAST)

OUTPUT deleted.*

WHERE DatabaseLogID = 7;
```

```
GO
```

This example removes a row from a table used as a queue and returns the deleted values to the processing application in a single action. Other semantics may also be implemented, such as using a table to implement a stack. However, SQL Server does not guarantee the order in which rows are processed and returned by DML statements using the OUTPUT clause. It is up to the application to include an appropriate WHERE clause that can guarantee the desired semantics, or understand that when multiple rows may qualify for the DML operation, there is no guaranteed order. The following example uses a subquery and assumes uniqueness is a characteristic of the `DatabaseLogID` column in order to implement the desired ordering semantics.

```
USE tempdb;
GO

CREATE TABLE dbo.table1
(
    id INT,
    employee VARCHAR(32)
)
go

INSERT INTO dbo.table1 VALUES
      (1, 'Fred')
     ,(2, 'Tom')
     ,(3, 'Sally')
     ,(4, 'Alice');
GO

DECLARE @MyTableVar TABLE
(
    id INT,
    employee VARCHAR(32)
);

PRINT 'table1, before delete'
SELECT * FROM dbo.table1;

DELETE FROM dbo.table1
OUTPUT DELETED.* INTO @MyTableVar
WHERE id = 4 OR id = 2;

PRINT 'table1, after delete'
SELECT * FROM dbo.table1;
```

```
PRINT '@MyTableVar, after delete'
SELECT * FROM @MyTableVar;

DROP TABLE dbo.table1;

--Results
--table1, before delete
--id          employee
------------- ------------------------------
--1           Fred
--2           Tom
--3           Sally
--4           Alice
--
--table1, after delete
--id          employee
------------- ------------------------------
--1           Fred
--3           Sally
--@MyTableVar, after delete
--id          employee
------------- ------------------------------
--2           Tom
--4           Alice
```

Note

Use the READPAST table hint in UPDATE and DELETE statements if your scenario allows for multiple applications to perform a destructive read from one table. This prevents locking issues that can come up if another application is already reading the first qualifying record in the table.

Permissions

SELECT permissions are required on any columns retrieved through <dml_select_list> or used in <scalar_expression>.

INSERT permissions are required on any tables specified in <output_table>.

Examples

A. Using OUTPUT INTO with a simple INSERT statement

The following example inserts a row into the `ScrapReason` table and uses the `OUTPUT` clause to return the results of the statement to the `@MyTableVartable` variable. Because the `ScrapReasonID` column is defined with an IDENTITY property, a value is not specified in the `INSERT` statement for that column. However, note that the value generated by the Database Engine for that column is returned in the OUTPUT clause in the column inserted.ScrapReasonID.

```
USE AdventureWorks2012;
GO
DECLARE @MyTableVar table( NewScrapReasonID smallint,
                           Name varchar(50),
                           ModifiedDate datetime);
INSERT Production.ScrapReason
    OUTPUT INSERTED.ScrapReasonID, INSERTED.Name, INSERTED.ModifiedDate
        INTO @MyTableVar
VALUES (N'Operator error', GETDATE());

--Display the result set of the table variable.
SELECT NewScrapReasonID, Name, ModifiedDate FROM @MyTableVar;
--Display the result set of the table.
SELECT ScrapReasonID, Name, ModifiedDate
FROM Production.ScrapReason;
GO
```

B. Using OUTPUT with a DELETE statement

The following example deletes all rows in the ShoppingCartItem table. The clause OUTPUT deleted.* specifies that the results of the DELETE statement, that is all columns in the deleted rows, be returned to the calling application. The SELECT statement that follows verifies the results of the delete operation on the ShoppingCartItem table.

```
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
```

C. Using OUTPUT INTO with an UPDATE statement

The following example updates the VacationHours column in the Employee table by 25 percent for the first 10 rows. The OUTPUT clause returns the VacationHours value that exists before applying the UPDATE statement in the column deleted.VacationHours, and the updated value in the column inserted.VacationHours to the @MyTableVartable variable.

Two SELECT statements follow that return the values in @MyTableVar and the results of the update operation in the Employee table.

```
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
```

D. Using OUTPUT INTO to return an expression

The following example builds on example C by defining an expression in the `OUTPUT` clause as the difference between the updated `VacationHours` value and the `VacationHours` value before the update was applied. The value of this expression is returned to the `@MyTableVartable` variable in the column `VacationHoursDifference`.

```
USE AdventureWorks2012;
GO
DECLARE @MyTableVar table(
    EmpID int NOT NULL,
    OldVacationHours int,
    NewVacationHours int,
    VacationHoursDifference int,
    ModifiedDate datetime);
UPDATE TOP (10) HumanResources.Employee
SET VacationHours = VacationHours * 1.25,
    ModifiedDate = GETDATE()
OUTPUT inserted.BusinessEntityID,
       deleted.VacationHours,
       inserted.VacationHours,
       inserted.VacationHours - deleted.VacationHours,
       inserted.ModifiedDate
INTO @MyTableVar;
--Display the result set of the table variable.
SELECT EmpID, OldVacationHours, NewVacationHours,
    VacationHoursDifference, ModifiedDate
FROM @MyTableVar;
GO
SELECT TOP (10) BusinessEntityID, VacationHours, ModifiedDate
FROM HumanResources.Employee;
GO
```

E. Using OUTPUT INTO with from_table_name in an UPDATE statement

The following example updates the `ScrapReasonID` column in the `WorkOrder` table for all work orders with a specified `ProductID` and `ScrapReasonID`. The `OUTPUT INTO` clause returns values from the table being updated (`WorkOrder`) and also from the `Product` table. The `Product` table is used in the `FROM` clause to specify the rows to update. Because the `WorkOrder` table has an `AFTER UPDATE` trigger defined on it, the `INTO` keyword is required.

```
USE AdventureWorks2012;
GO
DECLARE @MyTestVar table (
    OldScrapReasonID int NOT NULL,
    NewScrapReasonID int NOT NULL,
    WorkOrderID int NOT NULL,
    ProductID int NOT NULL,
    ProductName nvarchar(50)NOT NULL);
UPDATE Production.WorkOrder
SET ScrapReasonID = 4
OUTPUT deleted.ScrapReasonID,
       inserted.ScrapReasonID,
       inserted.WorkOrderID,
       inserted.ProductID,
       p.Name
    INTO @MyTestVar
FROM Production.WorkOrder AS wo
    INNER JOIN Production.Product AS p
    ON wo.ProductID = p.ProductID
    AND wo.ScrapReasonID= 16
    AND p.ProductID = 733;
SELECT OldScrapReasonID, NewScrapReasonID, WorkOrderID,
    ProductID, ProductName
FROM @MyTestVar;
GO
```

F. Using OUTPUT INTO 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 DELETE statement. The OUTPUT clause returns columns from the table being deleted (deleted.ProductID, deleted.ProductPhotoID) and columns from the Product table. This table is used in the FROM clause to specify the rows to delete.

```
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
```

G. Using OUTPUT INTO with a large object data type

The following example updates a partial value in DocumentSummary, an nvarchar(max) column in the Production.Document table, by using the .WRITE clause. The word components is replaced by the word features by specifying the replacement word, the beginning location (offset) of the word to be replaced in the existing data, and the number of characters to be replaced (length). The example uses the OUTPUT clause to return the before and after images of the DocumentSummary column to the @MyTableVartable variable. Note that the full before and after images of the DocumentSummary column are returned.

```
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
```

H. Using OUTPUT in an INSTEAD OF trigger

The following example uses the `OUTPUT` clause in a trigger to return the results of the trigger operation. First, a view is created on the `ScrapReason` table, and then an `INSTEAD OF INSERT` trigger is defined on the view that lets only the `Name` column of the base table to be modified by the user. Because the column `ScrapReasonID` is an `IDENTITY` column in the base table, the trigger ignores the user-supplied value. This allows the Database Engine to automatically generate the correct value. Also, the value supplied by the user for `ModifiedDate` is ignored and is set to the current date. The `OUTPUT` clause returns the values actually inserted into the `ScrapReason` table.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID('dbo.vw_ScrapReason','V') IS NOT NULL
    DROP VIEW dbo.vw_ScrapReason;
GO
CREATE VIEW dbo.vw_ScrapReason
AS (SELECT ScrapReasonID, Name, ModifiedDate
    FROM Production.ScrapReason);
GO
CREATE TRIGGER dbo.io_ScrapReason
    ON dbo.vw_ScrapReason
INSTEAD OF INSERT
AS
BEGIN
--ScrapReasonID is not specified in the list of columns to be inserted
--because it is an IDENTITY column.
    INSERT INTO Production.ScrapReason (Name, ModifiedDate)
        OUTPUT INSERTED.ScrapReasonID, INSERTED.Name,
               INSERTED.ModifiedDate
    SELECT Name, getdate()
    FROM inserted;
END
GO
INSERT vw_ScrapReason (ScrapReasonID, Name, ModifiedDate)
```

```
VALUES (99, N'My scrap reason','20030404');
GO
```

Here is the result set generated on April 12, 2004 ('2004-04-12'). Notice that the ScrapReasonIDActual and ModifiedDate columns reflect the values generated by the trigger operation instead of the values provided in the INSERT statement.

```
ScrapReasonID  Name             ModifiedDate
-------------  ---------------- -----------------------
17             My scrap reason  2004-04-12 16:23:33.050
```

I. Using OUTPUT INTO with identity and computed columns

The following example creates the EmployeeSales table and then inserts several rows into it using an INSERT statement with a SELECT statement to retrieve data from source tables. The EmployeeSales table contains an identity column (EmployeeID) and a computed column (ProjectedSales). Because these values are generated by the SQL Server Database Engine during the insert operation, neither of these columns can be defined in @MyTableVar.

```
USE AdventureWorks2012 ;
GO
IF OBJECT_ID ('dbo.EmployeeSales', 'U') IS NOT NULL
    DROP TABLE dbo.EmployeeSales;
GO
CREATE TABLE dbo.EmployeeSales
( EmployeeID   int IDENTITY (1,5)NOT NULL,
  LastName     nvarchar(20) NOT NULL,
  FirstName    nvarchar(20) NOT NULL,
  CurrentSales money NOT NULL,
  ProjectedSales AS CurrentSales * 1.10
);
GO
DECLARE @MyTableVar table(
  LastName     nvarchar(20) NOT NULL,
  FirstName    nvarchar(20) NOT NULL,
  CurrentSales money NOT NULL
  );

INSERT INTO dbo.EmployeeSales (LastName, FirstName, CurrentSales)
  OUTPUT INSERTED.LastName,
         INSERTED.FirstName,
         INSERTED.CurrentSales
  INTO @MyTableVar
    SELECT c.LastName, c.FirstName, sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
```

```
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.BusinessEntityID LIKE '2%'
    ORDER BY c.LastName, c.FirstName;

SELECT LastName, FirstName, CurrentSales
FROM @MyTableVar;
GO
SELECT EmployeeID, LastName, FirstName, CurrentSales, ProjectedSales
FROM dbo.EmployeeSales;
GO
```

J. Using OUTPUT and OUTPUT INTO in a single statement

The following example deletes rows in the `ProductProductPhoto` table based on search criteria defined in the `FROM` clause of `DELETE` statement. The `OUTPUT INTO` clause returns columns from the table being deleted (`deleted.ProductID`, `deleted.ProductPhotoID`) and columns from the `Product` table to the `@MyTableVartable` variable. The `Product` table is used in the `FROM` clause to specify the rows to delete. The `OUTPUT` clause returns the `deleted.ProductID`, `deleted.ProductPhotoID` columns and the date and time the row was deleted from the `ProductProductPhoto` table to the calling application.

```
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
OUTPUT DELETED.ProductID, DELETED.ProductPhotoID, GETDATE() AS DeletedDate
FROM Production.ProductProductPhoto AS ph
JOIN Production.Product as p
    ON ph.ProductID = p.ProductID
WHERE p.ProductID BETWEEN 800 and 810;

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

K. Inserting data returned from an OUTPUT clause

The following example captures data returned from the `OUTPUT` clause of a `MERGE` statement, and inserts that data into another table. The `MERGE` statement updates the `Quantity` column of the `ProductInventory` table daily, based on orders that are processed in the `SalesOrderDetail` table. It also deletes rows for products whose inventories drop to `0` or below. The example captures the rows that are deleted and inserts them into another table, `ZeroInventory`, which tracks products with no inventory.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID(N'Production.ZeroInventory', N'U') IS NOT NULL
    DROP TABLE Production.ZeroInventory;
GO
--Create ZeroInventory table.
CREATE TABLE Production.ZeroInventory (DeletedProductID int, RemovedOnDate DateTime);
GO

INSERT INTO Production.ZeroInventory (DeletedProductID, RemovedOnDate)
SELECT ProductID, GETDATE()
FROM
(   MERGE Production.ProductInventory AS pi
    USING (SELECT ProductID, SUM(OrderQty) FROM Sales.SalesOrderDetail AS sod
           JOIN Sales.SalesOrderHeader AS soh
           ON sod.SalesOrderID = soh.SalesOrderID
           AND soh.OrderDate = '20070401'
           GROUP BY ProductID) AS src (ProductID, OrderQty)
    ON (pi.ProductID = src.ProductID)
    WHEN MATCHED AND pi.Quantity - src.OrderQty <= 0
        THEN DELETE
    WHEN MATCHED
        THEN UPDATE SET pi.Quantity = pi.Quantity - src.OrderQty
    OUTPUT $action, deleted.ProductID) AS Changes (Action, ProductID)
WHERE Action = 'DELETE';
IF @@ROWCOUNT = 0
PRINT 'Warning: No rows were inserted';
GO
SELECT DeletedProductID, RemovedOnDate FROM Production.ZeroInventory;
```

See Also

DELETE (Transact-SQL)

INSERT (Transact-SQL)

UPDATE (Transact-SQL)

table (Transact-SQL)

CREATE TRIGGER (Transact-SQL)

sp_configure (Transact-SQL)

READTEXT

Reads **text**, **ntext**, or **image** values from a **text**, **ntext**, or **image** column, starting from a specified offset and reading the specified number of bytes.

Important

This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Use the SUBSTRING function instead.

Transact-SQL Syntax Conventions

Syntax

```
READTEXT { table.column text_ptr offset size } [ HOLDLOCK ]
```

Arguments

table.column

Is the name of a table and column from which to read. Table and column names must comply with the rules for identifiers. Specifying the table and column names is required; however, specifying the database name and owner names is optional.

text_ptr

Is a valid text pointer. text_ptr must be **binary(16)**.

offset

Is the number of bytes (when the **text** or **image** data types are used) or characters (when the **ntext** data type is used) to skip before it starts to read the **text**, **image**, or **ntext** data.

size

Is the number of bytes (when the **text** or **image** data types are used) or characters (when the **ntext** data type is used) of data to read. If size is 0, 4 KB bytes of data is read.

HOLDLOCK

Causes the text value to be locked for reads until the end of the transaction. Other users can read the value, but they cannot modify it.

Remarks

Use the TEXTPTR function to obtain a valid text_ptr value. TEXTPTR returns a pointer to the **text**, **ntext**, or **image** column in the specified row or to the **text**, **ntext**, or **image** column in the last row returned by the query if more than one row is returned. Because TEXTPTR returns a 16-byte binary string, we recommend declaring a local variable to hold the text pointer, and then use the variable with READTEXT. For more information about declaring a local variable, see DECLARE @local_variable.

In SQL Server, in-row text pointers may exist but may not be valid. For more information about the **text in row** option, see sp_tableoption. For more information about invalidating text pointers, see sp_invalidate_textptr.

The value of the @@TEXTSIZE function supersedes the size specified for READTEXT if it is less than the specified size for READTEXT. The @@TEXTSIZE function specifies the limit on the number of bytes of data to be returned set by the SET TEXTSIZE statement. For more information about how to set the session setting for TEXTSIZE, see SET TEXTSIZE.

Permissions

READTEXT permissions default to users that have SELECT permissions on the specified table. Permissions are transferable when SELECT permissions are transferred.

Examples

The following example reads the second through twenty-sixth characters of the `pr_info` column in the `pub_info` table.

Note

To run this example, you must install the **pubs** sample database.

```
USE pubs;
GO
DECLARE @ptrval varbinary(16);
SELECT @ptrval = TEXTPTR(pr_info)
   FROM pub_info pr INNER JOIN publishers p
      ON pr.pub_id = p.pub_id
      AND p.pub_name = 'New Moon Books'
READTEXT pub_info.pr_info @ptrval 1 25;
GO
```

See Also

@@TEXTSIZE

UPDATETEXT

WRITETEXT

Search Condition

Is a combination of one or more predicates that use the logical operators AND, OR, and NOT.

Transact-SQL Syntax Conventions

Syntax

```
<search_condition> ::=
    { [ NOT ] <predicate> | ( <search_condition> ) }
    [ { AND | OR } [ NOT ] { <predicate> | ( <search_condition> ) } ]
[ ,...n ]
<predicate> ::=
    { expression { = | <> | ! = | > | > = | ! > | < | < = | ! < } expression
    | string_expression [ NOT ] LIKE string_expression
  [ ESCAPE 'escape_character' ]
    | expression [ NOT ] BETWEEN expression AND expression
    | expression IS [ NOT ] NULL
    | CONTAINS
  ( { column | * } ,'<contains_search_condition>')
    | FREETEXT ( { column | * } ,'freetext_string')
    | expression [ NOT ] IN (subquery | expression [ ,...n ] )
    | expression { = | <> | ! = | > | > = | ! > | < | < = | ! < }
  { ALL | SOME | ANY} (subquery)
    | EXISTS (subquery)     }
```

Arguments

<search_condition>

Specifies the conditions for the rows returned in the result set for a SELECT statement, query expression, or subquery. For an UPDATE statement, specifies the rows to be updated. For a DELETE statement, specifies the rows to be deleted. There is no limit to the number of predicates that can be included in a Transact-SQL statement search condition.

NOT

Negates the Boolean expression specified by the predicate. For more information, see NOT.

AND

Combines two conditions and evaluates to TRUE when both of the conditions are TRUE. For more information, see AND.

OR

Combines two conditions and evaluates to TRUE when either condition is TRUE. For more information, see OR.

<predicate>

Is an expression that returns TRUE, FALSE, or UNKNOWN.

expression

Is a column name, a constant, a function, a variable, a scalar subquery, or any combination of column names, constants, and functions connected by an operator or operators, or a subquery. The expression can also contain the CASE expression.

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.

=

Is the operator used to test the equality between two expressions.

<>

Is the operator used to test the condition of two expressions not being equal to each other.

!=

Is the operator used to test the condition of two expressions not being equal to each other.

>

Is the operator used to test the condition of one expression being greater than the other.

>=

Is the operator used to test the condition of one expression being greater than or equal to the other expression.

!>

Is the operator used to test the condition of one expression not being greater than the other expression.

<

Is the operator used to test the condition of one expression being less than the other.

<=

Is the operator used to test the condition of one expression being less than or equal to the other expression.

!<

Is the operator used to test the condition of one expression not being less than the other expression.

string_expression

Is a string of characters and wildcard characters.

[NOT] LIKE

Indicates that the subsequent character string is to be used with pattern matching. For more information, see LIKE.

ESCAPE 'escape_ character'

Allows for a wildcard character to be searched for in a character string instead of functioning as a wildcard. escape_character is the character that is put in front of the wildcard character to indicate this special use.

[NOT] BETWEEN

Specifies an inclusive range of values. Use AND to separate the starting and ending values. For more information, see BETWEEN.

IS [NOT] NULL

Specifies a search for null values, or for values that are not null, depending on the keywords used. An expression with a bitwise or arithmetic operator evaluates to NULL if any one of the operands is NULL.

CONTAINS

Searches columns that contain character-based data for precise or less precise (*fuzzy*) matches to single words and phrases, the proximity of words within a certain distance of one another, and weighted matches. This option can only be used with SELECT statements. For more information, see CONTAINS.

FREETEXT

Provides a simple form of natural language query by searching columns that contain character-based data for values that match the meaning instead of the exact words in the predicate. This option can only be used with SELECT statements. For more information, see FREETEXT.

[NOT] IN

Specifies the search for an expression, based on whether the expression is included in or excluded from a list. The search expression can be a constant or a column name, and the list can be a set of constants or, more typically, a subquery. Enclose the list of values in parentheses. For more information, see IN.

subquery

Can be considered a restricted SELECT statement and is similar to <query_expresssion> in the SELECT statement. The ORDER BY clause and the INTO keyword are not allowed. For more information, see SELECT.

ALL

Used with a comparison operator and a subquery. Returns TRUE for <predicate> when all values retrieved for the subquery satisfy the comparison operation, or FALSE when not all values satisfy the comparison or when the subquery returns no rows to the outer statement. For more information, see ALL.

{ SOME | ANY }

Used with a comparison operator and a subquery. Returns TRUE for <predicate> when any value retrieved for the subquery satisfies the comparison operation, or FALSE when no values in the subquery satisfy the comparison or when the subquery returns no rows to the outer statement. Otherwise, the expression is UNKNOWN. For more information, see SOME | ANY.

EXISTS

Used with a subquery to test for the existence of rows returned by the subquery. For more information, see EXISTS.

Remarks

The order of precedence for the logical operators is NOT (highest), followed by AND, followed by OR. Parentheses can be used to override this precedence in a search condition. The order of evaluation of logical operators can vary depending on choices made by the query optimizer. For more information about how the logical operators operate on logic values, see AND, OR, and NOT.

Examples

A. Using WHERE with LIKE and ESCAPE syntax

The following example searches for the rows in which the `LargePhotoFileName` column has the characters `green_`, and uses the `ESCAPE` option because _ is a wildcard character. Without specifying the `ESCAPE` option, the query would search for any description values that contain the word `green` followed by any single character other than the _ character.

```
USE AdventureWorks2012 ;
GO
SELECT *
FROM Production.ProductPhoto
WHERE LargePhotoFileName LIKE '%greena_%' ESCAPE 'a' ;
```

B. Using WHERE and LIKE syntax with Unicode data

The following example uses the `WHERE` clause to retrieve the mailing address for any company that is outside the United States (`US`) and in a city whose name starts with `Pa`.

```
USE AdventureWorks2012 ;
GO
```

```
SELECT AddressLine1, AddressLine2, City, PostalCode, CountryRegionCode
FROM Person.Address AS a
JOIN Person.StateProvince AS s ON a.StateProvinceID = s.StateProvinceID
WHERE CountryRegionCode NOT IN ('US')
AND City LIKE N'Pa%' ;
```

See Also

Aggregate Functions

CASE

CONTAINSTABLE

Cursors

DELETE

Expressions

FREETEXTTABLE

FROM

Operators

UPDATE

SELECT

Retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables in SQL Server 2012. The full syntax of the SELECT statement is complex, but the main clauses can be summarized as:

[WITH<common_table_expression>]

SELECT select_list[INTOnew_table]

[FROMtable_source] [WHEREsearch_condition]

[GROUP BY group_by_expression]

[HAVINGsearch_condition]

[ORDER BY order_expression [ASC | DESC]]

The UNION, EXCEPT and INTERSECT operators can be used between queries to combine or compare their results into one result set.

Transact-SQL Syntax Conventions

Syntax

```
<SELECT statement> ::=
    [WITH <common_table_expression> [,...n]]
    <query_expression>
    [ ORDER BY { order_by_expression | column_position [ ASC | DESC ] }
  [ ,...n ] ]
    [ <FOR Clause>]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
<query_expression> ::=
    { <query_specification> | ( <query_expression> ) }
    [  { UNION [ ALL ] | EXCEPT | INTERSECT }
<query_specification> | ( <query_expression> ) [...n ] ]
<query_specification> ::=
SELECT [ ALL | DISTINCT ]
    [TOP (expression) [PERCENT] [ WITH TIES ] ]
    < select_list >
    [ INTO new_table ]
    [ FROM { <table_source> } [ ,...n ] ]
    [ WHERE <search_condition> ]
    [ <GROUP BY> ]
    [ HAVING < search_condition > ]
```

Remarks

Because of the complexity of the SELECT statement, detailed syntax elements and arguments are shown by clause:

WITH common_table_expression	HAVING
SELECT Clause	UNION
INTO Clause	EXCEPT and INTERSECT
FROM	ORDER BY
WHERE	FOR Clause
GROUP BY	OPTION Clause

The order of the clauses in the SELECT statement is significant. Any one of the optional clauses can be omitted, but when the optional clauses are used, they must appear in the appropriate order.

SELECT statements are permitted in user-defined functions only if the select lists of these statements contain expressions that assign values to variables that are local to the functions.

A four-part name constructed with the OPENDATASOURCE function as the server-name part can be used as a table source wherever a table name can appear in a SELECT statement.

Some syntax restrictions apply to SELECT statements that involve remote tables.

Logical Processing Order of the SELECT statement

The following steps show the logical processing order, or binding order, for a SELECT statement. This order determines when the objects defined in one step are made available to the clauses in subsequent steps. For example, if the query processor can bind to (access) the tables or views defined in the FROM clause, these objects and their columns are made available to all subsequent steps. Conversely, because the SELECT clause is step 8, any column aliases or derived columns defined in that clause cannot be referenced by preceding clauses. However, they can be referenced by subsequent clauses such as the ORDER BY clause. Note that the actual physical execution of the statement is determined by the query processor and the order may vary from this list.

1. FROM
2. ON
3. JOIN
4. WHERE
5. GROUP BY
6. WITH CUBE or WITH ROLLUP
7. HAVING
8. SELECT
9. DISTINCT
10. ORDER BY
11. TOP

Permissions

Selecting data requires **SELECT** permission on the table or view, which could be inherited from a higher scope such as **SELECT** permission on the schema or **CONTROL** permission on the table. Or requires membership in the **db_datareader** or **db_owner** fixed database roles, or the **sysadmin** fixed server role. Creating a new table using **SELECTINTO** also requires both the **CREATETABLE** permission, and the **ALTERSCHEMA** permission on the schema that owns the new table.

See Also

SELECT Examples (Transact-SQL)

SELECT Clause

Specifies the columns to be returned by the query.

Transact-SQL Syntax Conventions

Syntax

```
SELECT [ ALL | DISTINCT ]
[ TOP (expression) [ PERCENT ] [ WITH TIES ] ]
<select_list>
<select_list> ::=
    {
      *
      | { table_name | view_name | table_alias }.*
      | {
          [ { table_name | view_name | table_alias }. ]
               { column_name | $IDENTITY | $ROWGUID }
          | udt_column_name [ { . | :: } { { property_name | field_name }
            | method_name (argument [ ,...n] ) } ]
          | expression
          [ [ AS ] column_alias ]
         }
      | column_alias =expression
    } [ ,...n ]
```

Arguments

ALL

Specifies that duplicate rows can appear in the result set. ALL is the default.

DISTINCT

Specifies that only unique rows can appear in the result set. Null values are considered equal for the purposes of the DISTINCT keyword.

TOP (expression) [PERCENT] [WITH TIES]

Indicates that only a specified first set or percent of rows will be returned from the query result set. expression can be either a number or a percent of the rows.

For backward compatibility, using the TOP expression without parentheses in SELECT statements is supported, but we do not recommend it. For more information, see TOP.

<select_list>

The columns to be selected for the result set. The select list is a series of expressions separated by commas. The maximum number of expressions that can be specified in the select list is 4096.

*****

Specifies that all columns from all tables and views in the FROM clause should be returned. The columns are returned by table or view, as specified in the FROM clause, and in the order in which they exist in the table or view.

table_name | view_name | table_alias.*

Limits the scope of the * to the specified table or view.

column_name

Is the name of a column to return. Qualify column_name to prevent an ambiguous reference, such as occurs when two tables in the FROM clause have columns with duplicate names. For example, the SalesOrderHeader and SalesOrderDetail tables in the database both have a column named ModifiedDate. If the two tables are joined in a query, the modified date of the SalesOrderDetail entries can be specified in the select list as SalesOrderDetail.ModifiedDate.

expression

Is a constant, function, any combination of column names, constants, and functions connected by an operator or operators, or a subquery.

$IDENTITY

Returns the identity column. For more information, see IDENTITY (Property), ALTER TABLE (Transact-SQL), and CREATE TABLE (Transact-SQL).

If more than one table in the FROM clause has a column with the IDENTITY property, $IDENTITY must be qualified with the specific table name, such as T1.$IDENTITY.

$ROWGUID

Returns the row GUID column.

If there is more than one table in the FROM clause with the ROWGUIDCOL property,

$ROWGUID must be qualified with the specific table name, such as T1.$ROWGUID.

udt_column_name

Is the name of a common language runtime (CLR) user-defined type column to return.

Note

SQL Server Management Studio returns user-defined type values in binary representation. To return user-defined type values in string or XML format, use CAST or CONVERT.

{ . | :: }

Specifies a method, property, or field of a CLR user-defined type. Use .for an instance (nonstatic) method, property, or field. Use :: for a static method, property, or field. To invoke a method, property, or field of a CLR user-defined type, you must have EXECUTE permission on the type.

property_name

Is a public property of udt_column_name.

field_name

Is a public data member of udt_column_name.

method_name

Is a public method of udt_column_name that takes one or more arguments.method_name cannot be a mutator method.

The following example selects the values for the `Location` column, defined as type `point`, from the `Cities` table, by invoking a method of the type called `Distance`:

```
CREATE TABLE Cities (
     Name varchar(20),
     State varchar(20),
     Location point );
GO
DECLARE @p point (32, 23), @distance float;
GO
SELECT Location.Distance (@p)
FROM Cities;
```

column_ alias

Is an alternative name to replace the column name in the query result set. For example, an alias such as Quantity, or Quantity to Date, or Qty can be specified for a column named quantity.

Aliases are used also to specify names for the results of expressions, for example:

```
USE AdventureWorks2012;
GO
SELECT AVG(UnitPrice) AS [Average Price]
FROM Sales.SalesOrderDetail;
```

column_alias can be used in an ORDER BY clause. However, it cannot be used in a WHERE, GROUP BY, or HAVING clause. If the query expression is part of a DECLARE CURSOR statement, column_alias cannot be used in the FOR UPDATE clause.

Remarks

The length of data returned for **text** or **ntext** columns that are included in the select list is set to the smallest value of the following: the actual size of the **text** column, the default TEXTSIZE session setting, or the hard-coded application limit. To change the length of returned text for the session, use the SET statement. By default, the limit on the length of text data returned with a SELECT statement is 4,000 bytes.

The SQL Server Database Engine raises exception 511 and rolls back the current running statement if either of the following behavior occurs:

- The SELECT statement produces a result row or an intermediate work table row exceeding 8,060 bytes.
- The DELETE, INSERT, or UPDATE statement tries an action on a row exceeding 8,060 bytes.

An error occurs if no column name is specified to a column created by a SELECT INTO or CREATE VIEW statement.

See Also

SELECT Examples (Transact-SQL)

Expressions

SELECT (Transact-SQL)

SELECT Examples

This topic provides examples of using the SELECT statement.

A. Using SELECT to retrieve rows and columns

The following example shows three code examples. This first code example returns all rows (no WHERE clause is specified) and all columns (using the `*`) from the `Product` table in the database.

```
USE AdventureWorks2012;
GO
SELECT *
FROM Production.Product
ORDER BY Name ASC;
-- Alternate way.
USE AdventureWorks2012;
GO
SELECT p.*
FROM Production.Product AS p
ORDER BY Name ASC;
GO
```

This example returns all rows (no WHERE clause is specified), and only a subset of the columns (`Name`, `ProductNumber`, `ListPrice`) from the `Product` table in the database. Additionally, a column heading is added.

```
USE AdventureWorks2012;
GO
SELECT Name, ProductNumber, ListPrice AS Price
FROM Production.Product
ORDER BY Name ASC;
GO
```

This example returns only the rows for `Product` that have a product line of `R` and that have days to manufacture that is less than `4`.

```
USE AdventureWorks2012;
GO
SELECT Name, ProductNumber, ListPrice AS Price
FROM Production.Product
WHERE ProductLine = 'R'
AND DaysToManufacture < 4
ORDER BY Name ASC;
GO
```

B. Using SELECT with column headings and calculations

The following examples return all rows from the `Product` table. The first example returns total sales and the discounts for each product. In the second example, the total revenue is calculated for each product.

```
USE AdventureWorks2012;
GO
SELECT p.Name AS ProductName,
NonDiscountSales = (OrderQty * UnitPrice),
Discounts = ((OrderQty * UnitPrice) * UnitPriceDiscount)
FROM Production.Product AS p
INNER JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
ORDER BY ProductName DESC;
GO
```

This is the query that calculates the revenue for each product in each sales order.

```
USE AdventureWorks2012;
GO
SELECT 'Total income is', ((OrderQty * UnitPrice) * (1.0 - UnitPriceDiscount)), ' for ',
p.Name AS ProductName
FROM Production.Product AS p
INNER JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
ORDER BY ProductName ASC;
GO
```

C. Using DISTINCT with SELECT

The following example uses `DISTINCT` to prevent the retrieval of duplicate titles.

```
USE AdventureWorks2012;
GO
SELECT DISTINCT JobTitle
FROM HumanResources.Employee
ORDER BY JobTitle;
GO
```

D. Creating tables with SELECT INTO

The following first example creates a temporary table named `#Bicycles` in `tempdb`.

```
USE tempdb;
GO
IF OBJECT_ID (N'#Bicycles',N'U') IS NOT NULL
DROP TABLE #Bicycles;
```

```
GO
SELECT *
INTO #Bicycles
FROM AdventureWorks2012.Production.Product
WHERE ProductNumber LIKE 'BK%';
GO
```

This second example creates the permanent table `NewProducts`.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID('dbo.NewProducts', 'U') IS NOT NULL
    DROP TABLE dbo.NewProducts;
GO
ALTER DATABASE AdventureWorks2012 SET RECOVERY BULK_LOGGED;
GO

SELECT * INTO dbo.NewProducts
FROM Production.Product
WHERE ListPrice > $25
AND ListPrice < $100;
GO
ALTER DATABASE AdventureWorks2012 SET RECOVERY FULL;
GO
```

E. Using correlated subqueries

The following example shows queries that are semantically equivalent and illustrates the difference between using the `EXISTS` keyword and the `IN` keyword. Both are examples of a valid subquery that retrieves one instance of each product name for which the product model is a long sleeve logo jersey, and the `ProductModelID` numbers match between the `Product` and `ProductModel` tables.

```
USE AdventureWorks2012;
GO
SELECT DISTINCT Name
FROM Production.Product AS p
WHERE EXISTS
    (SELECT *
     FROM Production.ProductModel AS pm
     WHERE p.ProductModelID = pm.ProductModelID
           AND pm.Name LIKE 'Long-Sleeve Logo Jersey%');
GO

-- OR
```

```
USE AdventureWorks2012;
GO
SELECT DISTINCT Name
FROM Production.Product
WHERE ProductModelID IN
    (SELECT ProductModelID
     FROM Production.ProductModel
     WHERE Name LIKE 'Long-Sleeve Logo Jersey%');
GO
```

The following example uses `IN` in a correlated, or repeating, subquery. This is a query that depends on the outer query for its values. The query is executed repeatedly, one time for each row that may be selected by the outer query. This query retrieves one instance of the first and last name of each employee for which the bonus in the `SalesPerson` table is `5000.00` and for which the employee identification numbers match in the `Employee` and `SalesPerson` tables.

```
USE AdventureWorks2012;
GO
SELECT DISTINCT p.LastName, p.FirstName
FROM Person.Person AS p
JOIN HumanResources.Employee AS e
    ON e.BusinessEntityID = p.BusinessEntityID WHERE 5000.00 IN
    (SELECT Bonus
     FROM Sales.SalesPerson AS sp
     WHERE e.BusinessEntityID = sp.BusinessEntityID);
GO
```

The previous subquery in this statement cannot be evaluated independently of the outer query. It requires a value for `Employee.EmployeeID`, but this value changes as the SQL Server Database Engine examines different rows in `Employee`.

A correlated subquery can also be used in the `HAVING` clause of an outer query. This example finds the product models for which the maximum list price is more than twice the average for the model.

```
USE AdventureWorks2012;
GO
SELECT p1.ProductModelID
FROM Production.Product AS p1
GROUP BY p1.ProductModelID
HAVING MAX(p1.ListPrice) >= ALL
    (SELECT AVG(p2.ListPrice)
     FROM Production.Product AS p2
     WHERE p1.ProductModelID = p2.ProductModelID);
```

```
GO
```

This example uses two correlated subqueries to find the names of employees who have sold a particular product.

```
USE AdventureWorks2012;
GO
SELECT DISTINCT pp.LastName, pp.FirstName
FROM Person.Person pp JOIN HumanResources.Employee e
ON e.BusinessEntityID = pp.BusinessEntityID WHERE pp.BusinessEntityID IN
(SELECT SalesPersonID
FROM Sales.SalesOrderHeader
WHERE SalesOrderID IN
(SELECT SalesOrderID
FROM Sales.SalesOrderDetail
WHERE ProductID IN
(SELECT ProductID
FROM Production.Product p
WHERE ProductNumber = 'BK-M68B-42')));
GO
```

F. Using GROUP BY

The following example finds the total of each sales order in the database.

```
USE AdventureWorks2012;
GO
SELECT SalesOrderID, SUM(LineTotal) AS SubTotal
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
ORDER BY SalesOrderID;
GO
```

Because of the `GROUP BY` clause, only one row containing the sum of all sales is returned for each sales order.

G. Using GROUP BY with multiple groups

The following example finds the average price and the sum of year-to-date sales, grouped by product ID and special offer ID.

```
USE AdventureWorks2012;
GO
SELECT ProductID, SpecialOfferID, AVG(UnitPrice) AS [Average Price],
    SUM(LineTotal) AS SubTotal
FROM Sales.SalesOrderDetail
```

```
GROUP BY ProductID, SpecialOfferID
ORDER BY ProductID;
GO
```

H. Using GROUP BY and WHERE

The following example puts the results into groups after retrieving only the rows with list prices greater than `$1000`.

```
USE AdventureWorks2012;
GO
SELECT ProductModelID, AVG(ListPrice) AS [Average List Price]
FROM Production.Product
WHERE ListPrice > $1000
GROUP BY ProductModelID
ORDER BY ProductModelID;
GO
```

I. Using GROUP BY with an expression

The following example groups by an expression. You can group by an expression if the expression does not include aggregate functions.

```
USE AdventureWorks2012;
GO
SELECT AVG(OrderQty) AS [Average Quantity],
NonDiscountSales = (OrderQty * UnitPrice)
FROM Sales.SalesOrderDetail
GROUP BY (OrderQty * UnitPrice)
ORDER BY (OrderQty * UnitPrice) DESC;
GO
```

J. Using GROUP BY with ORDER BY

The following example finds the average price of each type of product and orders the results by average price.

```
USE AdventureWorks2012;
GO
SELECT ProductID, AVG(UnitPrice) AS [Average Price]
FROM Sales.SalesOrderDetail
WHERE OrderQty > 10
GROUP BY ProductID
ORDER BY AVG(UnitPrice);
GO
```

K. Using the HAVING clause

The first example that follows shows a `HAVING` clause with an aggregate function. It groups the rows in the `SalesOrderDetail` table by product ID and eliminates products whose average order quantities are five or less. The second example shows a `HAVING` clause without aggregate functions.

```
USE AdventureWorks2012;
GO
SELECT ProductID
FROM Sales.SalesOrderDetail
GROUP BY ProductID
HAVING AVG(OrderQty) > 5
ORDER BY ProductID;
GO
```

This query uses the `LIKE` clause in the `HAVING` clause.

```
USE AdventureWorks2012 ;
GO
SELECT SalesOrderID, CarrierTrackingNumber
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID, CarrierTrackingNumber
HAVING CarrierTrackingNumber LIKE '4BD%'
ORDER BY SalesOrderID ;
GO
```

L. Using HAVING and GROUP BY

The following example shows using `GROUP BY`, `HAVING`, `WHERE`, and `ORDER BY` clauses in one `SELECT` statement. It produces groups and summary values but does so after eliminating the products with prices over $25 and average order quantities under 5. It also organizes the results by `ProductID`.

```
USE AdventureWorks2012;
GO
SELECT ProductID
FROM Sales.SalesOrderDetail
WHERE UnitPrice < 25.00
GROUP BY ProductID
HAVING AVG(OrderQty) > 5
ORDER BY ProductID;
GO
```

M. Using HAVING with SUM and AVG

The following example groups the `SalesOrderDetail` table by product ID and includes only those groups of products that have orders totaling more than `$1000000.00` and whose average order quantities are less than `3`.

```
USE AdventureWorks2012;
GO
SELECT ProductID, AVG(OrderQty) AS AverageQuantity, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
GROUP BY ProductID
HAVING SUM(LineTotal) > $1000000.00
AND AVG(OrderQty) < 3;
GO
```

To see the products that have had total sales greater than `$2000000.00`, use this query:

```
USE AdventureWorks2012;
GO
SELECT ProductID, Total = SUM(LineTotal)
FROM Sales.SalesOrderDetail
GROUP BY ProductID
HAVING SUM(LineTotal) > $2000000.00;
GO
```

If you want to make sure there are at least one thousand five hundred items involved in the calculations for each product, use `HAVING COUNT(*) > 1500` to eliminate the products that return totals for fewer than `1500` items sold. The query looks like this:

```
USE AdventureWorks2012;
GO
SELECT ProductID, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
GROUP BY ProductID
HAVING COUNT(*) > 1500;
GO
```

N. Using the INDEX optimizer hint

The following example shows two ways to use the `INDEX` optimizer hint. The first example shows how to force the optimizer to use a nonclustered index to retrieve rows from a table, and the second example forces a table scan by using an index of 0.

```
USE AdventureWorks2012;
GO
SELECT pp.FirstName, pp.LastName, e.NationalIDNumber
FROM HumanResources.Employee AS e WITH (INDEX(AK_Employee_NationalIDNumber))
```

```
JOIN Person.Person AS pp on e.BusinessEntityID = pp.BusinessEntityID
WHERE LastName = 'Johnson';
GO

-- Force a table scan by using INDEX = 0.
USE AdventureWorks2012;
GO
SELECT pp.LastName, pp.FirstName, e.JobTitle
FROM HumanResources.Employee AS e WITH (INDEX = 0) JOIN Person.Person AS pp
ON e.BusinessEntityID = pp.BusinessEntityID
WHERE LastName = 'Johnson';
GO
```

M. Using OPTION and the GROUP hints

The following example shows how the `OPTION (GROUP)` clause is used with a `GROUP BY` clause.

```
USE AdventureWorks2012;
GO
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE UnitPrice < $5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (HASH GROUP, FAST 10);
GO
```

O. Using the UNION query hint

The following example uses the `MERGE UNION` query hint.

```
USE AdventureWorks2012;
GO
SELECT BusinessEntityID, JobTitle, HireDate, VacationHours, SickLeaveHours
FROM HumanResources.Employee AS e1
UNION
SELECT BusinessEntityID, JobTitle, HireDate, VacationHours, SickLeaveHours
FROM HumanResources.Employee AS e2
OPTION (MERGE UNION);
GO
```

P. Using a simple UNION

In the following example, the result set includes the contents of the `ProductModelID` and `Name` columns of both the `ProductModel` and `Gloves` tables.

```
USE AdventureWorks2012;
```

```
GO
IF OBJECT_ID ('dbo.Gloves', 'U') IS NOT NULL
DROP TABLE dbo.Gloves;
GO
-- Create Gloves table.
SELECT ProductModelID, Name
INTO dbo.Gloves
FROM Production.ProductModel
WHERE ProductModelID IN (3, 4);
GO

-- Here is the simple union.
USE AdventureWorks2012;
GO
SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE ProductModelID NOT IN (3, 4)
UNION
SELECT ProductModelID, Name
FROM dbo.Gloves
ORDER BY Name;
GO
```

Q. Using SELECT INTO with UNION

In the following example, the `INTO` clause in the second `SELECT` statement specifies that the table named `ProductResults` holds the final result set of the union of the designated columns of the `ProductModel` and `Gloves` tables. Note that the `Gloves` table is created in the first `SELECT` statement.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.ProductResults', 'U') IS NOT NULL
DROP TABLE dbo.ProductResults;
GO
IF OBJECT_ID ('dbo.Gloves', 'U') IS NOT NULL
DROP TABLE dbo.Gloves;
GO
-- Create Gloves table.
SELECT ProductModelID, Name
INTO dbo.Gloves
FROM Production.ProductModel
WHERE ProductModelID IN (3, 4);
GO
```

```
USE AdventureWorks2012;
GO
SELECT ProductModelID, Name
INTO dbo.ProductResults
FROM Production.ProductModel
WHERE ProductModelID NOT IN (3, 4)
UNION
SELECT ProductModelID, Name
FROM dbo.Gloves;
GO

SELECT ProductModelID, Name
FROM dbo.ProductResults;
```

R. Using UNION of two SELECT statements with ORDER BY

The order of certain parameters used with the UNION clause is important. The following example shows the incorrect and correct use of `UNION` in two `SELECT` statements in which a column is to be renamed in the output.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.Gloves', 'U') IS NOT NULL
DROP TABLE dbo.Gloves;
GO
-- Create Gloves table.
SELECT ProductModelID, Name
INTO dbo.Gloves
FROM Production.ProductModel
WHERE ProductModelID IN (3, 4);
GO

/* INCORRECT */
USE AdventureWorks2012;
GO
SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE ProductModelID NOT IN (3, 4)
ORDER BY Name
UNION
SELECT ProductModelID, Name
FROM dbo.Gloves;
GO
```

```
/* CORRECT */
USE AdventureWorks2012;
GO
SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE ProductModelID NOT IN (3, 4)
UNION
SELECT ProductModelID, Name
FROM dbo.Gloves
ORDER BY Name;
GO
```

S. Using UNION of three SELECT statements to show the effects of ALL and parentheses

The following examples use `UNION` to combine the results of three tables that all have the same 5 rows of data. The first example uses `UNION ALL` to show the duplicated records, and returns all 15 rows. The second example uses `UNION` without `ALL` to eliminate the duplicate rows from the combined results of the three `SELECT` statements, and returns 5 rows.

The third example uses `ALL` with the first `UNION` and parentheses enclose the second `UNION` that is not using `ALL`. The second `UNION` is processed first because it is in parentheses, and returns 5 rows because the `ALL` option is not used and the duplicates are removed. These 5 rows are combined with the results of the first `SELECT` by using the `UNION ALL` keywords. This does not remove the duplicates between the two sets of 5 rows. The final result has 10 rows.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('dbo.EmployeeOne', 'U') IS NOT NULL
DROP TABLE dbo.EmployeeOne;
GO
IF OBJECT_ID ('dbo.EmployeeTwo', 'U') IS NOT NULL
DROP TABLE dbo.EmployeeTwo;
GO
IF OBJECT_ID ('dbo.EmployeeThree', 'U') IS NOT NULL
DROP TABLE dbo.EmployeeThree;
GO

SELECT pp.LastName, pp.FirstName, e.JobTitle
INTO dbo.EmployeeOne
FROM Person.Person AS pp JOIN HumanResources.Employee AS e
ON e.BusinessEntityID = pp.BusinessEntityID
WHERE LastName = 'Johnson';
GO
SELECT pp.LastName, pp.FirstName, e.JobTitle
```

```
INTO dbo.EmployeeTwo
FROM Person.Person AS pp JOIN HumanResources.Employee AS e
ON e.BusinessEntityID = pp.BusinessEntityID
WHERE LastName = 'Johnson';
GO
SELECT pp.LastName, pp.FirstName, e.JobTitle
INTO dbo.EmployeeThree
FROM Person.Person AS pp JOIN HumanResources.Employee AS e
ON e.BusinessEntityID = pp.BusinessEntityID
WHERE LastName = 'Johnson';
GO
-- Union ALL
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeOne
UNION ALL
SELECT LastName, FirstName ,JobTitle
FROM dbo.EmployeeTwo
UNION ALL
SELECT LastName, FirstName,JobTitle
FROM dbo.EmployeeThree;
GO

SELECT LastName, FirstName,JobTitle
FROM dbo.EmployeeOne
UNION
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeTwo
UNION
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeThree;
GO

SELECT LastName, FirstName,JobTitle
FROM dbo.EmployeeOne
UNION ALL
(
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeTwo
UNION
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeThree
);
GO
```

See Also

CREATE TRIGGER

CREATE VIEW

DELETE

EXECUTE

Expressions

INSERT

LIKE

UNION

EXCEPT and INTERSECT (Transact-SQL)

UPDATE

WHERE

PathName (Transact-SQL)

INTO Clause (Transact-SQL)

FOR Clause

FOR clause is used to specify either the BROWSE or the XML option. BROWSE and XML are unrelated options.

Important

The XMLDATA directive to the FOR XML option is deprecated. Use XSD generation in the case of RAW and AUTO modes. There is no replacement for the XMLDATA directive in EXPLICIT mode. This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

Transact-SQL Syntax Conventions

Syntax

```
[ FOR { BROWSE | <XML> } ]
<XML> ::=
XML
{
   { RAW [ ( 'ElementName' ) ] | AUTO }
   [
<CommonDirectives>
      [ , { XMLDATA | XMLSCHEMA [ ( 'TargetNameSpaceURI' ) ] } ]
```

```
        [ , ELEMENTS [ XSINIL | ABSENT ]
      ]
    | EXPLICIT
      [
        <CommonDirectives>
        [ , XMLDATA ]
      ]
    | PATH [ ( 'ElementName' ) ]
      [
  <CommonDirectives>
        [ , ELEMENTS [ XSINIL | ABSENT ] ]
      ]
  }

  <CommonDirectives> ::=
  [ , BINARY BASE64 ]
  [ , TYPE ]
  [ , ROOT [ ( 'RootName' ) ] ]
```

Arguments

BROWSE

Specifies that updates be allowed while viewing the data in a DB-Library browse mode cursor. A table can be browsed in an application if the table includes a **timestamp** column, the table has a unique index, and the FOR BROWSE option is at the end of the SELECT statements sent to an instance of SQL Server.

> **Note**
> You cannot use the <lock_hint> HOLDLOCK in a SELECT statement that includes the FOR BROWSE option.

FOR BROWSE cannot appear in SELECT statements that are joined by the UNION operator.

> **Note**
> When the unique index key columns of a table are nullable, and the table is on the inner side of an outer join, the index is not supported by browse mode.

The browse mode lets you scan the rows in your SQL Server table and update the data in your table one row at a time. To access a SQL Server table in your application in the browse mode, you must use one of the following two options:

- The SELECT statement that you use to access the data from your SQL Server table must end with the keywords **FOR BROWSE**. When you turn on the **FOR BROWSE** option to use browse mode, temporary tables are created.
- You must run the following Transact-SQL statement to turn on the browse mode by using the **NO_BROWSETABLE** option:

  ```
  SET NO_BROWSETABLE ON
  ```

  When you turn on the **NO_BROWSETABLE** option, all the SELECT statements behave as if the **FOR BROWSE** option is appended to the statements. However, the **NO_BROWSETABLE** option does not create the temporary tables that the **FOR BROWSE** option generally uses to send the results to your application.

When you try to access the data from SQL Server tables in browse mode by using a SELECT query that involves an outer join statement, and when a unique index is defined on the table that is present on the inner side of an outer join statement, the browse mode does not support the unique index. The browse mode supports the unique index only when all the unique index key columns can accept null values. The browse mode does not support the unique index if the following conditions are true:

- You try to access the data from SQL Server tables in browse mode by using a SELECT query that involves an outer join statement.
- A unique index is defined on the table that is present on the inner side of an outer join statement.

To reproduce this behavior in the browse mode, follow these steps:

1. In SQL Server Management Studio, create a database, named SampleDB.
2. In the SampleDB database, create a tleft table and a tright table that both contain a single column that is named c1. Define a unique index on the c1 column in the tleft table, and set the column to accept null values. To do this, run the following Transact-SQL statements in an appropriate query window:

   ```
   CREATE TABLE tleft(c1 INT NULL UNIQUE) ;

   GO

   CREATE TABLE tright(c1 INT NULL) ;

   GO
   ```

3. Insert several values in the tleft table and the tright table. Make sure that you insert a null value in the tleft table. To do this, run the following Transact-SQL statements in the query window:

   ```
   INSERT INTO tleft VALUES(2) ;

   INSERT INTO tleft VALUES(NULL) ;

   INSERT INTO tright VALUES(1) ;

   INSERT INTO tright VALUES(3) ;
   ```

```
INSERT INTO tright VALUES(NULL) ;

GO
```

4. Turn on the **NO_BROWSETABLE** option. To do this, run the following Transact-SQL statements in the query window:

   ```
   SET NO_BROWSETABLE ON ;

   GO
   ```

5. Access the data in the tleft table and the tright table by using an outer join statement in the SELECT query. Make sure that the tleft table is on the inner side of the outer join statement. To do this, run the following Transact-SQL statements in the query window:

   ```
   SELECT tleft.c1

   FROM tleft

   RIGHT JOIN tright

   ON tleft.c1 = tright.c1

   WHERE tright.c1 <> 2 ;
   ```

   Notice the following output in the Results pane:

   ```
   c1

   ----

   NULL

   NULL
   ```

After you run the SELECT query to access the tables in the browse mode, the result set of the SELECT query contains two null values for the c1 column in the tleft table because of the definition of the right outer join statement. Therefore, in the result set, you cannot distinguish between the null values that came from the table and the null values that the right outer join statement introduced. You might receive incorrect results if you must ignore the null values from the result set.

Note

If the columns that are included in the unique index do not accept null values, all the null values in the result set were introduced by the right outer join statement.

XML

Specifies that the results of a query are to be returned as an XML document. One of the following XML modes must be specified: RAW, AUTO, EXPLICIT. For more information about XML data and SQL Server, see Constructing XML Using FOR XML.

RAW [('ElementName')]

Takes the query result and transforms each row in the result set into an XML element with a generic identifier <row /> as the element tag. You can optionally specify a name for the row element. The resulting XML output uses the specified ElementName as the row element generated for each row. For more information, see Using RAW Mode and Using RAW Mode.

AUTO

Returns query results in a simple, nested XML tree. Each table in the FROM clause, for which at least one column is listed in the SELECT clause, is represented as an XML element. The columns listed in the SELECT clause are mapped to the appropriate element attributes. For more information, see Using AUTO Mode.

EXPLICIT

Specifies that the shape of the resulting XML tree is defined explicitly. Using this mode, queries must be written in a particular way so that additional information about the desired nesting is specified explicitly. For more information, see Using EXPLICIT Mode.

XMLDATA

Returns inline XDR schema, but does not add the root element to the result. If XMLDATA is specified, XDR schema is appended to the document.

XMLSCHEMA [('TargetNameSpaceURI')]

Returns inline XSD schema. You can optionally specify a target namespace URI when you specify this directive, which returns the specified namespace in the schema. For more information, see Inline XSD Schema Generation.

ELEMENTS

Specifies that the columns are returned as subelements. Otherwise, they are mapped to XML attributes. This option is supported in RAW, AUTO and PATH modes only. For more information, see Using RAW Mode.

XSINIL

Specifies that an element with **xsi:nil** attribute set to **True** be created for NULL column values. This option can only be specified with ELEMENTS directive. For more information, see Generating Elements for NULL Values Using the XSINIL Parameter.

ABSENT

Indicates that for null column values, corresponding XML elements will not be added in the XML result. Specify this option only with ELEMENTS.

PATH [('ElementName')]

Generates a <row> element wrapper for each row in the result set. You can optionally specify an element name for the <row> element wrapper. If an empty string is provided, such as FOR XML PATH (**''**)), a wrapper element is not generated. Using PATH may provide a simpler alternative to queries written using the EXPLICIT directive. For more information, see Using PATH Mode.

BINARY BASE64

Specifies that the query returns the binary data in binary base64-encoded format. When you retrieve binary data by using RAW and EXPLICIT mode, this option must be specified. This is the default in AUTO mode.

TYPE

Specifies that the query returns results as **xml** type. For more information, see TYPE Directive in FOR XML Queries.

ROOT [('RootName')]

Specifies that a single top-level element be added to the resulting XML. You can optionally specify the root element name to generate. If the optional root name is not specified, the default <root> element is added.

Examples

The following example specifies `FOR XML AUTO` with the `TYPE` and `XMLSCHEMA` options. Because of the `TYPE` option, the result set is returned to the client as an **xml** type. The `XMLSCHEMA` option specifies that the inline XSD schema is included in the XML data returned, and the `ELEMENTS` option specifies that the XML result is element-centric.

```
USE AdventureWorks2012;
GO
SELECT p.BusinessEntityID, FirstName, LastName, PhoneNumber AS Phone
FROM Person.Person AS p
Join Person.PersonPhone AS pph ON p.BusinessEntityID  = pph.BusinessEntityID
WHERE LastName LIKE 'G%'
```

```
ORDER BY LastName, FirstName
FOR XML AUTO, TYPE, XMLSCHEMA, ELEMENTS XSINIL;
```

See Also

SELECT (Transact-SQL)

Constructing XML Using FOR XML

GROUP BY

Groups a selected set of rows into a set of summary rows by the values of one or more columns or expressions in SQL Server 2012. One row is returned for each group. Aggregate functions in the SELECT clause <select> list provide information about each group instead of individual rows.

The GROUP BY clause has an ISO-compliant syntax and a non-ISO-compliant syntax. Only one syntax style can be used in a single SELECT statement. Use the ISO compliant syntax for all new work. The non-ISO compliant syntax is provided for backward compatibility.

In this topic, a GROUP BY clause can be described as general or simple:

- A general GROUP BY clause includes GROUPING SETS, CUBE, ROLLUP, WITH CUBE, or WITH ROLLUP.
- A simple GROUP BY clause does not include GROUPING SETS, CUBE, ROLLUP, WITH CUBE, or WITH ROLLUP. GROUP BY (), grand total, is considered a simple GROUP BY.

Transact-SQL Syntax Conventions (Transact-SQL)

Syntax

ISO-Compliant Syntax

```
GROUP BY <group by spec>

<group by spec> ::=
<group by item> [ ,...n ]

<group by item> ::=
<simple group by item>
  | <rollup spec>
  | <cube spec>
  | <grouping sets spec>
  | <grand total>
```

```
<simple group by item> ::=
<column_expression>

<rollup spec> ::=
    ROLLUP (<composite element list>)

<cube spec> ::=
    CUBE (<composite element list>)

<composite element list> ::=
<composite element> [ ,...n ]

<composite element> ::=
<simple group by item>
    | (<simple group by item list>)

<simple group by item list> ::=
<simple group by item> [ ,...n ]

<grouping sets spec> ::=
    GROUPING SETS (<grouping set list>)

<grouping set list> ::=
<grouping set> [ ,...n ]

<grouping set> ::=
<grand total>
    | <grouping set item>
    | (<grouping set item list>)

<empty group> ::=
()

<grouping set item> ::=
<simple group by item>
    | <rollup spec>
```

| <cube spec>

<grouping set item list> ::=

<grouping set item> [,...**n**]

Non-ISO-Compliant Syntax

[GROUP BY [ALL] group_by_expression [,...**n**]

[WITH { CUBE | ROLLUP }]

]

Arguments

<column_expression>

Is the expression on which the grouping operation is performed.

ROLLUP ( )

Generates the simple GROUP BY aggregate rows, plus subtotal or super-aggregate rows, and also a grand total row.

The number of groupings that is returned equals the number of expressions in the <composite element list> plus one. For example, consider the following statement.

```
SELECT a, b, c, SUM ( <expression> )

FROM T

GROUP BY ROLLUP (a,b,c);
```

One row with a subtotal is generated for each unique combination of values of `(a, b, c)`, `(a, b)`, and `(a)`. A grand total row is also calculated.

Columns are rolled up from right to left. The column order affects the output groupings of ROLLUP and can affect the number of rows in the result set.

CUBE ( )

Generates simple GROUP BY aggregate rows, the ROLLUP super-aggregate rows, and cross-tabulation rows.

CUBE outputs a grouping for all permutations of expressions in the <composite element list>.

The number of groupings that is generated equals (2^n), where n = the number of expressions in the <composite element list>. For example, consider the following statement.

```
SELECT a, b, c, SUM (<expression>)

FROM T
```

```
GROUP BY CUBE (a,b,c);
```

One row is produced for each unique combination of values of `(a, b, c)`, `(a, b)`, `(a, c)`, `(b, c)`, `(a)`, `(b)` and `(c)` with a subtotal for each row and a grand total row.

Column order does not affect the output of CUBE.

GROUPING SETS ( )

Specifies multiple groupings of data in one query. Only the specified groups are aggregated instead of the full set of aggregations that are generated by CUBE or ROLLUP. The results are the equivalent of UNION ALL of the specified groups. GROUPING SETS can contain a single element or a list of elements. GROUPING SETS can specify groupings equivalent to those returned by ROLLUP or CUBE. The <grouping set item list> can contain ROLLUP or CUBE.

( )

The empty group generates a total.

Non-ISO Compliant Syntax

ALL

This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Includes all groups and result sets, even those that do not have any rows that meet the search condition specified in the WHERE clause. When ALL is specified, null values are returned for the summary columns of groups that do not meet the search condition. You cannot specify ALL with the CUBE or ROLLUP operators.

GROUP BY ALL is not supported in queries that access remote tables if there is also a WHERE clause in the query. GROUP BY ALL will fail on columns that have the FILESTREAM attribute.

group_by_expression

Is an expression on which grouping is performed. group_by_expression is also known as a grouping column. group_by expression can be a column or a non-aggregate expression that references a column returned by the FROM clause. A column alias that is defined in the SELECT list cannot be used to specify a grouping column.

Note

Columns of type **text**, **ntext**, and **image** cannot be used in group_by_expression.

For GROUP BY clauses that do not contain CUBE or ROLLUP, the number of group_by_expression items is limited by the GROUP BY column sizes, the aggregated columns, and the aggregate values involved in the query. This limit originates from the limit of 8,060 bytes on the intermediate worktable that is needed to hold intermediate query results. A maximum of 12 grouping expressions is permitted when CUBE or ROLLUP is specified.

xml data type methods cannot be specified directly in group_by_expression. Instead, refer to a user-defined function that uses **xml** data type methods inside it, or refer to a computed column that uses them.

WITH CUBE

This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Specifies that in addition to the usual rows provided by GROUP BY, summary rows are introduced into the result set. A GROUP BY summary row is returned for every possible combination of group and subgroup in the result set. Use the GROUPING function to determine whether null values in the result set are GROUP BY summary values.

The number of summary rows in the result set is determined by the number of columns included in the GROUP BY clause. Because CUBE returns every possible combination of group and subgroup, the number of rows is the same, regardless of the order in which the grouping columns are specified.

WITH ROLLUP

This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Specifies that in addition to the usual rows provided by GROUP BY, summary rows are introduced into the result set. Groups are summarized in a hierarchical order, from the lowest level in the group to the highest. The group hierarchy is determined by the order in which the grouping columns are specified. Changing the order of the grouping columns can affect the number of rows produced in the result set.

Important

Distinct aggregates, for example, AVG (DISTINCT column_name), COUNT (DISTINCT column_name), and SUM (DISTINCT column_name), are not supported when you use CUBE or ROLLUP. If these are used, the SQL Server Database Engine returns an error message and cancels the query.

Remarks

Expressions in the GROUP BY clause can contain columns of the tables, derived tables or views in the FROM clause. The columns are not required to appear in the SELECT clause <select> list.

Each table or view column in any nonaggregate expression in the <select> list must be included in the GROUP BY list:

- The following statements are allowed:

```
SELECT ColumnA, ColumnB FROM T GROUP BY ColumnA, ColumnB;

SELECT ColumnA + ColumnB FROM T GROUP BY ColumnA, ColumnB;

SELECT ColumnA + ColumnB FROM T GROUP BY ColumnA + ColumnB;

SELECT ColumnA + ColumnB + constant FROM T GROUP BY ColumnA, ColumnB;
```

- The following statements are not allowed:

```
SELECT ColumnA, ColumnB FROM T GROUP BY ColumnA + ColumnB;

SELECT ColumnA + constant + ColumnB FROM T GROUP BY ColumnA + ColumnB;
```

If aggregate functions are included in the SELECT clause <select list>, GROUP BY calculates a summary value for each group. These are known as vector aggregates.

Rows that do not meet the conditions in the WHERE clause are removed before any grouping operation is performed.

The HAVING clause is used with the GROUP BY clause to filter groups in the result set.

The GROUP BY clause does not order the result set. Use the ORDER BY clause to order the result set.

If a grouping column contains null values, all null values are considered equal, and they are put into a single group.

You cannot use GROUP BY with an alias to replace a column name in the AS clause unless the alias replaces a column name in a derived table in the FROM clause.

Duplicate grouping sets in a GROUPING SETS list are not eliminated. Duplicate grouping sets can be generated by specifying a column expression more than one time or by listing a column expression also generated by a CUBE or ROLLUP in the GROUPING SETS list.

Distinct aggregates, for example, AVG (DISTINCT column_name), COUNT (DISTINCT column_name), and SUM (DISTINCT column_name) are supported with ROLLUP, CUBE, and GROUPING SETS.

ROLLUP, CUBE, and GROUPING SETS cannot be specified in an indexed view.

GROUP BY or HAVING cannot be used directly on columns of **ntext**, **text**, or **image**. These columns can be used as arguments in functions that return a value of another data type, such as SUBSTRING() and CAST().

xml data type methods cannot be specified directly in a <column_expression>. Instead, refer to a user-defined function that uses **xml** data type methods inside it, or refer to a computed column that uses them.

GROUP BY Limitations for GROUPING SETS, ROLLUP, and CUBE

Syntax Limitations

GROUPING SETS are not allowed in the GROUP BY clause unless they are part of a GROUPING SETS list. For example, `GROUP BY C1, (C2,..., Cn)` is not allowed but `GROUP BY GROUPING SETS (C1, (C2, ..., Cn))` is allowed.

GROUPING SETS are not allowed inside GROUPING SETS. For example, `GROUP BY GROUPING SETS (C1, GROUPING SETS (C2, C3))` is not allowed.

The non-ISO ALL, WITH CUBE, and WITH ROLLUP keywords are not allowed in a GROUP BY clause with the ROLLUP, CUBE or GROUPING SETS keywords.

Size Limitations

For simple GROUP BY, there is no limit on the number of expressions.

For a GROUP BY clause that uses ROLLUP, CUBE, or GROUPING SETS, the maximum number of expressions is 32, and the maximum number of grouping sets that can be generated is 4096 (2^{12}). The following examples fail because the GROUP BY clause is too complex:

- The following examples generate 8192 (2^{13}) grouping sets.

  ```
  GROUP BY CUBE (a1, ..., a13)
  ```

  ```
  GROUP BY a1, ..., a13 WITH CUBE
  ```

- The following example generates 4097 (2^{12} + 1) grouping sets.

  ```
  GROUP BY GROUPING SETS( CUBE(a1, ..., a12), b )
  ```

- The following example also generates 4097 (2^{12} + 1) grouping sets. Both `CUBE ()` and the `()` grouping set produce a grand total row and duplicate grouping sets are not eliminated.

  ```
  GROUP BY GROUPING SETS( CUBE(a1, ..., a12), ())
  ```

Support for ISO and ANSI SQL-2006 GROUP BY Features

In SQL Server 2012, the GROUP BY clause cannot contain a subquery in an expression that is used for the group by list. Error 144 is returned.

SQL Server 2012 supports all GROUP BY features that are included in the SQL-2006 standard with the following syntax exceptions:

- Grouping sets are not allowed in the GROUP BY clause unless they are part of an explicit GROUPING SETS list. For example, `GROUP BY Column1, (Column2, ...ColumnN`) is allowed in the standard but not in SQL Server. `GROUP BY C1, GROUPING SETS ((Column2, ...ColumnN))` or `GROUP BY Column1, Column2, ... ColumnN` is allowed. These are semantically equivalent to the previous `GROUP BY` example. This is to avoid the possibility that `GROUP BY Column1, (Column2, ...ColumnN`) might be misinterpreted as `GROUP BY C1, GROUPING SETS ((Column2, ...ColumnN))`. This is not semantically equivalent.
- Grouping sets are not allowed inside grouping sets. For example, `GROUP BY GROUPING SETS (A1, A2,…An, GROUPING SETS (C1, C2, ...Cn))` is allowed in the SQL-2006 standard but not in SQL Server. SQL Server 2012 allows `GROUP BY GROUPING SETS( A1, A2,...An, C1, C2,`

...Cn) or GROUP BY GROUPING SETS((A1), (A2), ... (An), (C1), (C2), ... (Cn)). These examples are semantically equivalent to the first GROUP BY example and have a clearer syntax.

- GROUP BY [ALL/DISTINCT] is not allowed in a general GROUP BY clause or with the GROUPING SETS, ROLLUP, CUBE, WITH CUBE or WITH ROLLUP constructs. ALL is the default and is implicit.

Comparison of Supported GROUP BY Features

The following table describes the GROUP BY features that are supported based upon the version of SQL Server and the database compatibility level.

Feature	SQL Server 2005 Integration Services	SQL Server compatibility level 100 or higher	SQL Server 2008 or later with compatibility level 90
DISTINCT aggregates	Not supported for WITH CUBE or WITH ROLLUP.	Supported for WITH CUBE, WITH ROLLUP, GROUPING SETS, CUBE, or ROLLUP.	Same as compatibility level 100.

User-defined function with CUBE or ROLLUP name in the GROUP BY clause	User-defined function **dbo.cube(**arg1**,**...argN**)** or **dbo.rollup(**arg1**,...**argN) in the GROUP BY clause is allowed.
	For example:
	`SELECT SUM (x)`
	`FROM T`
	`GROUP BY dbo.cube(y);`
		User-defined function **dbo.cube (**arg1,...argN**)** or **dbo.rollup(**arg1**,**...argN**)** in the GROUP BY clause is not allowed.
		For example:
		`SELECT SUM (x)`
		`FROM T`
		`GROUP BY dbo.cube(y);`
		The following error message is returned: "Incorrect syntax near the keyword 'cube'|'rollup'."
		To avoid this problem, replace `dbo.cube` with `[dbo].[cube]` or `dbo.rollup` with `[dbo].[rollup]`.
		The following example is
			User-defined function **dbo.cube (**arg1**,**...argN) or **dbo.rollup(**arg1**,**...argN**)** in the GROUP BY clause is allowed
			For example:
			`SELECT SUM (x)`
			`FROM T`
			`GROUP BY dbo.cube(y);`

Feature	SQL Server 2005 Integration Services	SQL Server compatibility level 100 or higher	SQL Server 2008 or later with compatibility level 90

		allowed:
		`SELECT SUM (x)`
		`FROM T`
		`GROUP BY`
		`[dbo].[cube](y);`	

GROUPING SETS	Not supported	Supported	Supported
CUBE	Not supported	Supported	Not supported
ROLLUP	Not supported	Supported	Not supported
Grand total, such as GROUP BY ()	Not supported	Supported	Supported
GROUPING_ID function	Not supported	Supported	Supported
GROUPING function	Supported	Supported	Supported
WITH CUBE	Supported	Supported	Supported
WITH ROLLUP	Supported	Supported	Supported
WITH CUBE or WITH ROLLUP "duplicate" grouping removal	Supported	Supported	Supported

Examples

A. Using a simple GROUP BY clause

The following example retrieves the total for each `SalesOrderID` from the `SalesOrderDetail` table.

```
USE AdventureWorks2012;
GO
SELECT SalesOrderID, SUM(LineTotal) AS SubTotal
FROM Sales.SalesOrderDetail AS sod
```

```
GROUP BY SalesOrderID
ORDER BY SalesOrderID;
```

B. Using a GROUP BY clause with multiple tables

The following example retrieves the number of employees for each `City` from the `Address` table joined to the `EmployeeAddress` table.

```
USE AdventureWorks2012;
GO
SELECT a.City, COUNT(bea.AddressID) EmployeeCount
FROM Person.BusinessEntityAddress AS bea
    INNER JOIN Person.Address AS a
        ON bea.AddressID = a.AddressID
GROUP BY a.City
ORDER BY a.City;
```

C. Using a GROUP BY clause with an expression

The following example retrieves the total sales for each year by using the `DATEPART` function. The same expression must be present in both the `SELECT` list and `GROUP BY` clause.

```
USE AdventureWorks2012;
GO
SELECT DATEPART(yyyy,OrderDate) AS N'Year'
    ,SUM(TotalDue) AS N'Total Order Amount'
FROM Sales.SalesOrderHeader
GROUP BY DATEPART(yyyy,OrderDate)
ORDER BY DATEPART(yyyy,OrderDate);
```

D. Using a GROUP BY clause with a HAVING clause

The following example uses the `HAVING` clause to specify which of the groups generated in the `GROUP BY` clause should be included in the result set.

```
USE AdventureWorks2012;
GO
SELECT DATEPART(yyyy,OrderDate) AS N'Year'
    ,SUM(TotalDue) AS N'Total Order Amount'
FROM Sales.SalesOrderHeader
GROUP BY DATEPART(yyyy,OrderDate)
HAVING DATEPART(yyyy,OrderDate) >= N'2003'
ORDER BY DATEPART(yyyy,OrderDate);
```

See Also

GROUPING_ID (Transact-SQL)

GROUPING (Transact-SQL)

SELECT (Transact-SQL)

SELECT Clause (Transact-SQL)

HAVING

Specifies a search condition for a group or an aggregate. HAVING can be used only with the SELECT statement. HAVING is typically used in a GROUP BY clause. When GROUP BY is not used, HAVING behaves like a WHERE clause.

Transact-SQL Syntax Conventions

Syntax

[HAVING <**search condition**>]

Arguments

<search_condition>

Specifies the search condition for the group or the aggregate to meet.

The **text**, **image**, and **ntext** data types cannot be used in a HAVING clause.

Examples

The following example that uses a simple `HAVING` clause retrieves the total for each `SalesOrderID` from the `SalesOrderDetail` table that exceeds `$100000.00`.

```
USE AdventureWorks2012 ;
GO
SELECT SalesOrderID, SUM(LineTotal) AS SubTotal
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
HAVING SUM(LineTotal) > 100000.00
ORDER BY SalesOrderID ;
```

See Also

GROUP BY (Transact-SQL)

WHERE (Transact-SQL)

INTO Clause

SELECT…INTO creates a new table in the default filegroup and inserts the resulting rows from the query into it. To view the complete SELECT syntax, see SELECT (Transact-SQL).

Transact-SQL Syntax Conventions

Syntax

```
[ INTO new_table ]
```

Arguments

new_table

Specifies the name of a new table to be created, based on the columns in the select list and the rows chosen from the data source.

The format of new_table is determined by evaluating the expressions in the select list. The columns in new_table are created in the order specified by the select list. Each column in new_table has the same name, data type, nullability, and value as the corresponding expression in the select list. The IDENTITY property of a column is transferred except under the conditions defined in "Working with Identity Columns" in the Remarks section.

To create the table in another database on the same instance of SQL Server, specify new_table as a fully qualified name in the form *database.schema.table_name*.

You cannot create new_table on a remote server; however, you can populate new_table from a remote data source. To create new_table from a remote source table, specify the source table using a four-part name in the form *linked_server.catalog.schema.object* in the FROM clause of the SELECT statement. Alternatively, you can use the OPENQUERY function or the OPENDATASOURCE function in the FROM clause to specify the remote data source.

Data Types

The FILESTREAM attribute does not transfer to the new table. FILESTREAM BLOBs are copied and stored in the new table as **varbinary(max)** BLOBs. Without the FILESTREAM attribute, the **varbinary(max)** data type has a limitation of 2 GB. If a FILESTREAM BLOB exceeds this value, error 7119 is raised and the statement is stopped.

When an existing identity column is selected into a new table, the new column inherits the IDENTITY property, unless one of the following conditions is true:

- The SELECT statement contains a join, GROUP BY clause, or aggregate function.
- Multiple SELECT statements are joined by using UNION.

- The identity column is listed more than one time in the select list.
- The identity column is part of an expression.
- The identity column is from a remote data source.

If any one of these conditions is true, the column is created NOT NULL instead of inheriting the IDENTITY property. If an identity column is required in the new table but such a column is not available, or you want a seed or increment value that is different than the source identity column, define the column in the select list using the IDENTITY function. See "Creating an identity column using the IDENTITY function" in the Examples section below.

Limitations and Restrictions

You cannot specify a table variable or table-valued parameter as the new table.

You cannot use SELECT…INTO to create a partitioned table, even when the source table is partitioned. SELECT...INTO does not use the partition scheme of the source table; instead, the new table is created in the default filegroup. To insert rows into a partitioned table, you must first create the partitioned table and then use the INSERT INTO...SELECT FROM statement.

Indexes, constraints, and triggers defined in the source table are not transferred to the new table, nor can they be specified in the SELECT...INTO statement. If these objects are required, you can create them after executing the SELECT...INTO statement.

Specifying an ORDER BY clause does not guarantee the rows are inserted in the specified order.

When a sparse column is included in the select list, the sparse column property does not transfer to the column in the new table. If this property is required in the new table, alter the column definition after executing the SELECT...INTO statement to include this property.

When a computed column is included in the select list, the corresponding column in the new table is not a computed column. The values in the new column are the values that were computed at the time SELECT...INTO was executed.

Logging Behavior

The amount of logging for SELECT...INTO depends on the recovery model in effect for the database. Under the simple recovery model or bulk-logged recovery model, bulk operations are minimally logged. With minimal logging, using the SELECT… INTO statement can be more efficient than creating a table and then populating the table with an INSERT statement. For more information, see The Transaction Log (SQL Server).

Permissions

Requires CREATE TABLE permission in the destination database.

Examples

A. Creating a table by specifying columns from multiple sources

The following example creates the table `dbo.EmployeeAddresses` by selecting seven columns from various employee-related and address-related tables.

```
USE AdventureWorks2012;
GO
SELECT c.FirstName, c.LastName, e.JobTitle, a.AddressLine1, a.City,
    sp.Name AS [State/Province], a.PostalCode
INTO dbo.EmployeeAddresses
FROM Person.Person AS c
    JOIN HumanResources.Employee AS e
    ON e.BusinessEntityID = c.BusinessEntityID
    JOIN Person.BusinessEntityAddress AS bea
    ON e.BusinessEntityID = bea.BusinessEntityID
    JOIN Person.Address AS a
    ON bea.AddressID = a.AddressID
    JOIN Person.StateProvince as sp
    ON sp.StateProvinceID = a.StateProvinceID;
GO
```

B. Inserting rows using minimal logging

The following example creates the table `dbo.NewProducts` and inserts rows from the `Production.Product` table. The example assumes that the recovery model of the AdventureWorks2012 database is set to FULL. To ensure minimal logging is used, the recovery model of the AdventureWorks2012 database is set to BULK_LOGGED before rows are inserted and reset to FULL after the SELECT...INTO statement. This process ensures that the SELECT...INTO statement uses minimal space in the transaction log and performs efficiently.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID('dbo.NewProducts', 'U') IS NOT NULL
    DROP TABLE dbo.NewProducts;
GO
ALTER DATABASE AdventureWorks2012 SET RECOVERY BULK_LOGGED;
GO

SELECT * INTO dbo.NewProducts
FROM Production.Product
WHERE ListPrice > $25
AND ListPrice < $100;
GO
ALTER DATABASE AdventureWorks2012 SET RECOVERY FULL;
```

```
GO
```

C. Creating an identity column using the IDENTITY function

The following example uses the IDENTITY function to create an identity column in the new table `Person.USAddress`. This is required because the SELECT statement that defines the table contains a join, which causes the IDENTITY property to not transfer to the new table. Notice that the seed and increment values specified in the IDENTITY function are different from those of the `AddressID` column in the source table `Person.Address`.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ('Person.USAddress') IS NOT NULL
DROP TABLE Person.USAddress;
GO
-- Determine the IDENTITY status of the source column AddressID.
SELECT OBJECT_NAME(object_id) AS TableName, name AS column_name, is_identity, seed_value,
increment_value
FROM sys.identity_columns
WHERE name = 'AddressID';

-- Create a new table with columns from the existing table Person.Address. A new IDENTITY
-- column is created by using the IDENTITY function.
SELECT IDENTITY (int, 100, 5) AS AddressID,
       a.AddressLine1, a.City, b.Name AS State, a.PostalCode
INTO Person.USAddress
FROM Person.Address AS a
INNER JOIN Person.StateProvince AS b ON a.StateProvinceID = b.StateProvinceID
WHERE b.CountryRegionCode = N'US';

-- Verify the IDENTITY status of the AddressID columns in both tables.
SELECT OBJECT_NAME(object_id) AS TableName, name AS column_name, is_identity, seed_value,
increment_value
FROM sys.identity_columns
WHERE name = 'AddressID';
```

D. Creating a table by specifying columns from a remote data source

The following example demonstrates three methods of creating a new table on the local server from a remote data source. The example begins by creating a link to the remote data source. The linked server name, `MyLinkServer`, is then specified in the FROM clause of the first SELECT...INTO statement and in the OPENQUERY function of the second SELECT...INTO statement. The third SELECT...INTO statement uses the OPENDATASOURCE function, which specifies the remote data source directly instead of using the linked server name.

```
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
USE AdventureWorks2012;
GO
-- Specify the remote data source in the FROM clause using a four-part name
-- in the form linked_server.catalog.schema.object.
SELECT DepartmentID, Name, GroupName, ModifiedDate
INTO dbo.Departments
FROM MyLinkServer.AdventureWorks2012.HumanResources.Department
GO
-- Use the OPENQUERY function to access the remote data source.
SELECT DepartmentID, Name, GroupName, ModifiedDate
INTO dbo.DepartmentsUsingOpenQuery
FROM OPENQUERY(MyLinkServer, 'SELECT *
               FROM AdventureWorks2012.HumanResources.Department');
GO
-- Use the OPENDATASOURCE function to specify the remote data source.
-- Specify a valid server name for Data Source using the format server_name or
server_name\instance_name.
SELECT DepartmentID, Name, GroupName, ModifiedDate
INTO dbo.DepartmentsUsingOpenDataSource
FROM OPENDATASOURCE('SQLNCLI',
    'Data Source=server_name;Integrated Security=SSPI')
    .AdventureWorks2012.HumanResources.Department;
GO
```

See Also

SELECT

SELECT Examples (Transact-SQL)

INSERT (Transact-SQL)

IDENTITY (Function) (Transact-SQL)

ORDER BY Clause

Sorts data returned by a query in SQL Server 2012. Use this clause to:

- Order the result set of a query by the specified column list and, optionally, limit the rows returned to a specified range. The order in which rows are returned in a result set are not guaranteed unless an ORDER BY clause is specified.
- Determine the order in which ranking function values are applied to the result set.

Transact-SQL Syntax Conventions

Syntax

```
ORDER BY order_by_expression
    [ COLLATE collation_name ]
    [ ASC | DESC ]
    [ ,...n ]
[ <offset_fetch> ]



<offset_fetch> ::=
{
    OFFSET { integer_constant | offset_row_count_expression } { ROW | ROWS }
    [
      FETCH { FIRST | NEXT } {integer_constant | fetch_row_count_expression } { ROW | ROWS }
ONLY
    ]
}
```

Arguments

order_by_expression

Specifies a column or expression on which to sort the query result set. A sort column can be specified as a name or column alias, or a nonnegative integer representing the position of the column in the select list.

Multiple sort columns can be specified. Column names must be unique. The sequence of the sort columns in the ORDER BY clause defines the organization of the sorted result set. That is, the result set is sorted by the first column and then that ordered list is sorted by the second column, and so on.

The column names referenced in the ORDER BY clause must correspond to either a column in the select list or to a column defined in a table specified in the FROM clause without any ambiguities.

COLLATE collation_name

Specifies that the ORDER BY operation should be performed according to the collation specified in collation_name, and not according to the collation of the column as defined in the table or view. collation_name can be either a Windows collation name or a SQL collation name. For more information, see Collation and International Terminology. COLLATE is applicable only for columns of type**char**, **varchar**, **nchar**, and **nvarchar**.

ASC | DESC

Specifies that the values in the specified column should be sorted in ascending or descending order. ASC sorts from the lowest value to highest value. DESC sorts from highest value to lowest value. ASC is the default sort order. Null values are treated as the lowest possible values.

OFFSET { integer_constant | offset_row_count_expression } { ROW | ROWS }

Specifies the number of rows to skip before it starts to return rows from the query expression. The value can be an integer constant or expression that is greater than or equal to zero.

offset_row_count_expression can be a variable, parameter, or constant scalar subquery. When a subquery is used, it cannot reference any columns defined in the outer query scope. That is, it cannot be correlated with the outer query.

ROW and ROWS are synonyms and are provided for ANSI compatibility.

In query execution plans, the offset row count value is displayed in the **Offset** attribute of the TOP query operator.

FETCH { FIRST | NEXT } { integer_constant | fetch_row_count_expression } { ROW | ROWS } ONLY

Specifies the number of rows to return after the OFFSET clause has been processed. The value can be an integer constant or expression that is greater than or equal to one.

fetch_row_count_expression can be a variable, parameter, or constant scalar subquery. When a subquery is used, it cannot reference any columns defined in the outer query scope. That is, it cannot be correlated with the outer query.

FIRST and NEXT are synonyms and are provided for ANSI compatibility.

ROW and ROWS are synonyms and are provided for ANSI compatibility.

In query execution plans, the offset row count value is displayed in the **Rows** or **Top** attribute of the TOP query operator.

Best Practices

Avoid specifying integers in the ORDER BY clause as positional representations of the columns in the select list. For example, although a statement such as `SELECT ProductID, Name FROM Production.Production ORDER BY 2` is valid, the statement is not as easily understood by others compared with specifying the actual column name. In addition, changes to the select list, such as changing the column order or adding new columns, will require modifying the ORDER BY clause in order to avoid unexpected results.

In a SELECT TOP (*N*) statement, always use an ORDER BY clause. This is the only way to predictably indicate which rows are affected by TOP. For more information, see TOP (Transact-SQL).

Interoperability

When used with a SELECT…INTO statement to insert rows from another source, the ORDER BY clause does not guarantee the rows are inserted in the specified order.

Using OFFSET and FETCH in a view does not change the updateability property of the view.

Limitations and Restrictions

There is no limit to the number of columns in the ORDER BY clause; however, the total size of the columns specified in an ORDER BY clause cannot exceed 8,060 bytes.

Columns of type **ntext**, **text**, **image**, **geography**, **geometry**, and **xml** cannot be used in an ORDER BY clause.

An integer or constant cannot be specified when order_by_expression appears in a ranking function. For more information, see OVER Clause (Transact-SQL).

If a table name is aliased in the FROM clause, only the alias name can be used to qualify its columns in the ORDER BY clause.

Column names and aliases specified in the ORDER BY clause must be defined in the select list if the SELECT statement contains one of the following clauses or operators:

- UNION operator
- EXCEPT operator
- INTERSECT operator
- SELECT DISTINCT

Additionally, when the statement includes a UNION, EXCEPT, or INTERSECT operator, the column names or column aliases must be specified in the select list of the first (left-side) query.

In a query that uses UNION, EXCEPT, or INTERSECT operators, ORDER BY is allowed only at the end of the statement. This restriction applies only to when you specify UNION, EXCEPT and INTERSECT in a top-level query and not in a subquery. See the Examples section that follows.

The ORDER BY clause is not valid in views, inline functions, derived tables, and subqueries, unless either the TOP or OFFSET and FETCH clauses are also specified. When ORDER BY is used in these objects, the clause is used only to determine the rows returned by the TOP clause or OFFSET and FETCH clauses. The ORDER BY clause does not guarantee ordered results when these constructs are queried, unless ORDER BY is also specified in the query itself.

OFFSET and FETCH are not supported in indexed views or in a view that is defined by using the CHECK OPTION clause.

OFFSET and FETCH can be used in any query that allows TOP and ORDER BY with the following limitations:

- The OVER clause does not support OFFSET and FETCH.
- OFFSET and FETCH cannot be specified directly in INSERT, UPDATE, MERGE, and DELETE statements, but can be specified in a subquery defined in these statements. For example, in the INSERT INTO SELECT statement, OFFSET and FETCH can be specified in the SELECT statement.
- In a query that uses UNION, EXCEPT or INTERSECT operators, OFFSET and FETCH can only be specified in the final query that specifies the order of the query results.
- TOP cannot be combined with OFFSET and FETCH in the same query expression (in the same query scope).

Using OFFSET and FETCH to limit the rows returned

We recommend that you use the OFFSET and FETCH clauses instead of the TOP clause to implement a query paging solution and limit the number of rows sent to a client application.

Using OFFSET and FETCH as a paging solution requires running the query one time for each "page" of data returned to the client application. For example, to return the results of a query in 10-row increments, you must execute the query one time to return rows 1 to 10 and then run the query again to return rows 11 to 20 and so on. Each query is independent and not related to each other in any way. This means that, unlike using a cursor in which the query is executed once and state is maintained on the server, the client application is responsible for tracking state. To achieve stable results between query requests using OFFSET and FETCH, the following conditions must be met:

1. The underlying data that is used by the query must not change. That is, either the rows touched by the query are not updated or all requests for pages from the query are executed in a single transaction using either snapshot or serializable transaction isolation. For more information about these transaction isolation levels, see SET TRANSACTION ISOLATION LEVEL (Transact-SQL).
2. The ORDER BY clause contains a column or combination of columns that are guaranteed to be unique.

See the example "Running multiple queries in a single transaction" in the Examples section later in this topic.

If consistent execution plans are important in your paging solution, consider using the OPTIMIZE FOR query hint for the OFFSET and FETCH parameters. See "Specifying expressions for OFFSET and FETCH values" in the Examples section later in this topic. For more information about OPTIMZE FOR, see Query Hints (Transact-SQL).

Examples

Category	Featured syntax elements
Basic syntax	ORDER BY
Specifying ascending and descending order	DESC • ASC
Specifying a collation	COLLATE
Specifying a conditional order	CASE expression
Using ORDER BY in a ranking function	Ranking functions
Limiting the number of rows returned	OFFSET • FETCH
Using ORDER BY with UNION, EXCEPT, and INTERSECT	UNION

Basic syntax

Examples in this section demonstrate the basic functionality of the ORDER BY clause using the minimum required syntax.

A. Specifying a single column defined in the select list

The following example orders the result set by the numeric `ProductID` column. Because a specific sort order is not specified, the default (ascending order) is used.

```
USE AdventureWorks2012;
GO
SELECT ProductID, Name FROM Production.Product
WHERE Name LIKE 'Lock Washer%'
ORDER BY ProductID;
```

B. Specifying a column that is not defined in the select list

The following example orders the result set by a column that is not included in the select list, but is defined in the table specified in the FROM clause.

```
USE AdventureWorks2012;
GO
SELECT ProductID, Name, Color
FROM Production.Product
ORDER BY ListPrice;
```

C. Specifying an alias as the sort column

The following example specifies the column alias `SchemaName` as the sort order column.

```
USE AdventureWorks2012;
```

```
GO
SELECT name, SCHEMA_NAME(schema_id) AS SchemaName
FROM sys.objects
WHERE type = 'U'
ORDER BY SchemaName;
```

D. Specifying an expression as the sort column

The following example uses an expression as the sort column. The expression is defined by using the DATEPART function to sort the result set by the year in which employees were hired.

```
USE AdventureWorks2012;
Go
SELECT BusinessEntityID, JobTitle, HireDate
FROM HumanResources.Employee
ORDER BY DATEPART(year, HireDate);
```

Specifying ascending and descending sort order

A. Specifying a descending order

The following example orders the result set by the numeric column `ProductID` in descending order.

```
USE AdventureWorks2012;
GO
SELECT ProductID, Name FROM Production.Product
WHERE Name LIKE 'Lock Washer%'
ORDER BY ProductID DESC;
```

B. Specifying a ascending order

The following example orders the result set by the `Name` column in ascending order. Note that the characters are sorted alphabetically, not numerically. That is, 10 sorts before 2.

```
USE AdventureWorks2012;
GO
SELECT ProductID, Name FROM Production.Product
WHERE Name LIKE 'Lock Washer%'
ORDER BY Name ASC ;
```

C. Specifying both ascending and descending order

The following example orders the result set by two columns. The query result set is first sorted in ascending order by the `FirstName` column and then sorted in descending order by the `LastName` column.

```
USE AdventureWorks2012;
GO
SELECT LastName, FirstName FROM Person.Person
WHERE LastName LIKE 'R%'
```

```
ORDER BY FirstName ASC, LastName DESC ;
```

Specifying a collation

The following example shows how specifying a collation in the ORDER BY clause can change the order in which the query results are returned. A table is created that contains a column defined by using a case-insensitive, accent-insensitive collation. Values are inserted with a variety of case and accent differences. Because a collation is not specified in the ORDER BY clause, the first query uses the collation of the column when sorting the values. In the second query, a case-sensitive, accent-sensitive collation is specified in the ORDER BY clause, which changes the order in which the rows are returned.

```
USE tempdb;
GO
CREATE TABLE #t1 (name nvarchar(15) COLLATE Latin1_General_CI_AI)
GO
INSERT INTO #t1 VALUES(N'Sánchez'),(N'Sanchez'),(N'sánchez'),(N'sanchez');

-- This query uses the collation specified for the column 'name' for sorting.
SELECT name
FROM #t1
ORDER BY name;
-- This query uses the collation specified in the ORDER BY clause for sorting.
SELECT name
FROM #t1
ORDER BY name COLLATE Latin1_General_CS_AS;
```

Specifying a conditional order

The following examples uses the CASE expression in an ORDER BY clause to conditionally determine the sort order of the rows based on a given column value. In the first example, the value in the `SalariedFlag` column of the `HumanResources.Employee` table is evaluated. Employees that have the `SalariedFlag` set to 1 are returned in order by the `BusinessEntityID` in descending order. Employees that have the `SalariedFlag` set to 0 are returned in order by the `BusinessEntityID` in ascending order. In the second example, the result set is ordered by the column `TerritoryName` when the column `CountryRegionName` is equal to 'United States' and by `CountryRegionName` for all other rows.

```
SELECT BusinessEntityID, SalariedFlag
FROM HumanResources.Employee
ORDER BY CASE SalariedFlag WHEN 1 THEN BusinessEntityID END DESC
        ,CASE WHEN SalariedFlag = 0 THEN BusinessEntityID END;
GO

SELECT BusinessEntityID, LastName, TerritoryName, CountryRegionName
FROM Sales.vSalesPerson
```

```
WHERE TerritoryName IS NOT NULL
ORDER BY CASE CountryRegionName WHEN 'United States' THEN TerritoryName
         ELSE CountryRegionName END;
```

Using ORDER BY in a ranking function

The following example uses the ORDER BY clause in the ranking functions ROW_NUMBER, RANK, DENSE_RANK, and NTILE.

```
USE AdventureWorks2012;
GO
SELECT p.FirstName, p.LastName
    ,ROW_NUMBER() OVER (ORDER BY a.PostalCode) AS "Row Number"
    ,RANK() OVER (ORDER BY a.PostalCode) AS "Rank"
    ,DENSE_RANK() OVER (ORDER BY a.PostalCode) AS "Dense Rank"
    ,NTILE(4) OVER (ORDER BY a.PostalCode) AS "Quartile"
    ,s.SalesYTD, a.PostalCode
FROM Sales.SalesPerson AS s
    INNER JOIN Person.Person AS p
        ON s.BusinessEntityID = p.BusinessEntityID
    INNER JOIN Person.Address AS a
        ON a.AddressID = p.BusinessEntityID
WHERE TerritoryID IS NOT NULL AND SalesYTD <> 0;
```

Limiting the number of rows returned

The following examples use OFFSET and FETCH to limit the number of rows returned by a query.

A. Specifying integer constants for OFFSET and FETCH values

The following example specifies an integer constant as the value for the OFFSET and FETCH clauses. The first query returns all rows sorted by the column `DepartmentID`. Compare the results returned by this query with the results of the two queries that follow it. The next query uses the clause `OFFSET 5 ROWS` to skip the first 5 rows and return all remaining rows. The final query uses the clause `OFFSET 0 ROWS` to start with the first row and then uses `FETCH NEXT 10 ROWS ONLY` to limit the rows returned to 10 rows from the sorted result set.

```
USE AdventureWorks2012;
GO
SELECT p.FirstName, p.LastName
    ,ROW_NUMBER() OVER (ORDER BY a.PostalCode) AS "Row Number"
    ,RANK() OVER (ORDER BY a.PostalCode) AS "Rank"
    ,DENSE_RANK() OVER (ORDER BY a.PostalCode) AS "Dense Rank"
    ,NTILE(4) OVER (ORDER BY a.PostalCode) AS "Quartile"
    ,s.SalesYTD, a.PostalCode
FROM Sales.SalesPerson AS s
```

```
    INNER JOIN Person.Person AS p
        ON s.BusinessEntityID = p.BusinessEntityID
    INNER JOIN Person.Address AS a
        ON a.AddressID = p.BusinessEntityID
WHERE TerritoryID IS NOT NULL AND SalesYTD <> 0;
```

B. Specifying variables for OFFSET and FETCH values

The following example declares the variables `@StartingRowNumber` and `@FetchRows` and specifies these variables in the OFFSET and FETCH clauses.

```
USE AdventureWorks2012;
GO
-- Specifying variables for OFFSET and FETCH values
DECLARE @StartingRowNumber tinyint = 1
      , @FetchRows tinyint = 8;
SELECT DepartmentID, Name, GroupName
FROM HumanResources.Department
ORDER BY DepartmentID ASC
    OFFSET @StartingRowNumber ROWS
    FETCH NEXT @FetchRows ROWS ONLY;
```

C. Specifying expressions for OFFSET and FETCH values

The following example uses the expression `@StartingRowNumber - 1` to specify the OFFSET value and the expression `@EndingRowNumber - @StartingRowNumber + 1` to specify the FETCH value. In addition, the query hint, OPTIMIZE FOR, is specified. This hint can be used to provide 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. For more information, see Query Hints (Transact-SQL).

```
USE AdventureWorks2012;
GO

-- Specifying expressions for OFFSET and FETCH values
DECLARE @StartingRowNumber tinyint = 1
      , @EndingRowNumber tinyint = 8;
SELECT DepartmentID, Name, GroupName
FROM HumanResources.Department
ORDER BY DepartmentID ASC
    OFFSET @StartingRowNumber - 1 ROWS
    FETCH NEXT @EndingRowNumber - @StartingRowNumber + 1 ROWS ONLY
OPTION ( OPTIMIZE FOR (@StartingRowNumber = 1, @EndingRowNumber = 20) );
```

D. Specifying a constant scalar subquery for OFFSET and FETCH values

The following example uses a constant scalar subquery to define the value for the FETCH clause. The subquery returns a single value from the column `PageSize` in the table `dbo.AppSettings`.

```
-- Specifying a constant scalar subquery
USE AdventureWorks2012;
GO
CREATE TABLE dbo.AppSettings (AppSettingID int NOT NULL, PageSize int NOT NULL);
GO
INSERT INTO dbo.AppSettings VALUES(1, 10);
GO
DECLARE @StartingRowNumber tinyint = 1;
SELECT DepartmentID, Name, GroupName
FROM HumanResources.Department
ORDER BY DepartmentID ASC
    OFFSET @StartingRowNumber ROWS
    FETCH NEXT (SELECT PageSize FROM dbo.AppSettings WHERE AppSettingID = 1) ROWS ONLY;
```

E. Running multiple queries in a single transaction

The following example shows one method of implementing a paging solution that ensures stable results are returned in all requests from the query. The query is executed in a single transaction using the snapshot isolation level, and the column specified in the ORDER BY clause ensures column uniqueness.

```
USE AdventureWorks2012;
GO

-- Ensure the database can support the snapshot isolation level set for the query.
IF (SELECT snapshot_isolation_state FROM sys.databases WHERE name =
N'AdventureWorks2012') = 0
    ALTER DATABASE AdventureWorks2012 SET ALLOW_SNAPSHOT_ISOLATION ON;
GO

-- Set the transaction isolation level  to SNAPSHOT for this query.
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
GO

-- Beging the transaction
BEGIN TRANSACTION;
GO
-- Declare and set the variables for the OFFSET and FETCH values.
DECLARE @StartingRowNumber int = 1
      , @RowCountPerPage int = 3;

-- Create the condition to stop the transaction after all rows have been returned.
WHILE (SELECT COUNT(*) FROM HumanResources.Department) >= @StartingRowNumber
BEGIN
```

```
-- Run the query until the stop condition is met.
SELECT DepartmentID, Name, GroupName
FROM HumanResources.Department
ORDER BY DepartmentID ASC
    OFFSET @StartingRowNumber - 1 ROWS
    FETCH NEXT @RowCountPerPage ROWS ONLY;

-- Increment @StartingRowNumber value.
SET @StartingRowNumber = @StartingRowNumber + @RowCountPerPage;
CONTINUE
END;
GO
COMMIT TRANSACTION;
GO
```

Using ORDER BY with UNION, EXCEPT, and INTERSECT

When a query uses the UNION, EXCEPT, or INTERSECT operators, the ORDER BY clause must be specified at the end of the statement and the results of the combined queries are sorted. The following example returns all products that are red or yellow and sorts this combined list by the column `ListPrice`.

```
USE AdventureWorks2012;
GO
SELECT Name, Color, ListPrice
FROM Production.Product
WHERE Color = 'Red'
-- ORDER BY cannot be specified here.
UNION ALL
SELECT Name, Color, ListPrice
FROM Production.Product
WHERE Color = 'Yellow'
ORDER BY ListPrice ASC;
```

See Also

Expressions

SELECT

FROM

Ranking Functions (Transact-SQL)

TOP (Transact-SQL)

Query Hints (Transact-SQL)

EXCEPT and INTERSECT (Transact-SQL)

[UNION (Transact-SQL)]

[CASE (Transact-SQL)]

OVER Clause

Determines the partitioning and ordering of a rowset before the associated window function is applied. That is, the OVER clause defines a window or user-specified set of rows within a query result set. A window function then computes a value for each row in the window. You can use the OVER clause with functions to compute aggregated values such as moving averages, cumulative aggregates, running totals, or a top N per group results.

Applies to:

- Ranking functions
- Aggregate functions
- Analytic functions
- NEXT VALUE FOR function

Transact-SQL Syntax Conventions

Syntax

```
OVER (
       [ <PARTITION BY clause> ]
       [ <ORDER BY clause> ]
       [ <ROW or RANGE clause> ]
      )

<PARTITION BY clause> ::=
PARTITION BY value_expression , ... [ n ]

<ORDER BY clause> ::=
ORDER BY order_by_expression
    [ COLLATE collation_name ]
    [ ASC | DESC ]
    [ ,...n ]

<ROW or RANGE clause> ::=
{ ROWS | RANGE } <window frame extent>

<window frame extent> ::=
{   <window frame preceding>
  | <window frame between>
}

<window frame between> ::=
  BETWEEN <window frame bound> AND <window frame bound>

<window frame bound> ::=
{   <window frame preceding>
  | <window frame following>
}

<window frame preceding> ::=
{
    UNBOUNDED PRECEDING
  | <unsigned_value_specification> PRECEDING
  | CURRENT ROW
}

<window frame following> ::=
{
    UNBOUNDED FOLLOWING
  | <unsigned_value_specification> FOLLOWING
  | CURRENT ROW
}

<unsigned value specification> ::=
{  <unsigned integer literal> }
```

Arguments

PARTITION BY

Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.

value_expression

Specifies the column by which the rowset is partitioned. value_expression can only refer to columns made available by the FROM clause. value_expression cannot refer to expressions or aliases in the select list. value_expression can be a column expression, scalar subquery, scalar function, or user-defined variable.

<ORDER BY clause>

Defines the logical order of the rows within each partition of the result set. That is, it specifies the logical order in which the window functioncalculation is performed.

order_by_expression

Specifies a column or expression on which to sort. order_by_expression can only refer to columns made available by the FROM clause. An integer cannot be specified to represent a column name or alias.

COLLATE collation_name

Specifies that the ORDER BY operation should be performed according to the collation specified in collation_name. collation_name can be either a Windows collation name or a SQL collation name. For more information, see Collation and International Terminology. COLLATE is applicable only for columns of type **char**, **varchar**, **nchar**, and **nvarchar**.

ASC | DESC

Specifies that the values in the specified column should be sorted in ascending or descending order. ASC is the default sort order. Null values are treated as the lowest possible values.

ROWS | RANGE

Further limits the rows within the partition by specifying start and end points within the partition. This is done by specifying a range of rows with respect to the current row either by logical association or physical association. Physical association is achieved by using the ROWS clause.

The ROWS clause limits the rows within a partition by specifying a fixed number of rows preceding or following the current row. Alternatively, the RANGE clause logically limits the rows within a partition by specifying a range of values with respect to the value in the current row. Preceding and following rows are defined based on the ordering in the ORDER BY clause. The window frame “RANGE … CURRENT ROW …” includes all rows that have the same values in the ORDER BY expression as the current row. For example, ROWS BETWEEN 2 PRECEDING AND CURRENT ROW means that the window of rows that the function operates on is three rows in size, starting with 2 rows preceding until and including the current row.

Note

ROWS or RANGE requires that the ORDER BY clause be specified. If ORDER BY contains multiple order expressions, CURRENT ROW FOR RANGE considers all columns in the ORDER BY list when determining the current row.

UNBOUNDED PRECEDING

Specifies that the window starts at the first row of the partition. UNBOUNDED PRECEDING can only be specified as window starting point.

<unsigned value specification> PRECEDING

Specified with <unsigned value specification>to indicate the number of rows or values to precede the current row. This specification is not allowed for RANGE.

CURRENT ROW

Specifies that the window starts or ends at the current row when used with ROWS or the current value when used with RANGE. CURRENT ROW can be specified as both a starting and ending point.

BETWEEN <window frame bound > AND <window frame bound >

Used with either ROWS or RANGE to specify the lower (starting) and upper (ending) boundary points of the window. <window frame bound> defines the boundary starting point and <window frame bound> defines the boundary end point. The upper bound cannot be smaller than the lower bound.

UNBOUNDED FOLLOWING

Specifies that the window ends at the last row of the partition. UNBOUNDED FOLLOWING can only be specified as a window end point. For example RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING defines a window that starts with the current row and ends with the last row of the partition.

<unsigned value specification> FOLLOWING

Specified with <unsigned value specification> to indicate the number of rows or values to follow the current row. When <unsigned value specification> FOLLOWING is specified as the window starting point, the ending point must be <unsigned value specification>FOLLOWING. For example, ROWS BETWEEN 2 FOLLOWING AND 10

FOLLOWING defines a window that starts with the second row that follows the current row and ends with the tenth row that follows the current row. This specification is not allowed for RANGE.

unsigned integer literal

Is a positive integer literal (including 0) that specifies the number of rows or values to precede or follow the current row or value. This specification is valid only for ROWS.

General Remarks

More than one window function can be used in a single query with a single FROM clause. The OVER clause for each function can differ in partitioning and ordering.

If PARTITION BY is not specified, the function treats all rows of the query result set as a single group.

If ORDER BY is not specified entire partition is used for a window frame. This applies only to functions that do not require ORDER BY clause. If ROWS/RANGE is not specified but ORDER BY is specified, RANGE UNBOUNDED PRECEDING AND CURRENT ROW is used as default for window frame. This applies only to functions that have can accept optional ROWS/RANGE specification. For example, ranking functions cannot accept ROWS/RANGE, therefore this window frame is not applied even though ORDER BY is present and ROWS/RANGE is not.

If ROWS/RANGE is specified and <window frame preceding> is used for <window frame extent> (short syntax) then this specification is used for the window frame boundary starting point and CURRENT ROW is used for the boundary ending point. For example “ROWS 5 PRECEDING” is equal to “ROWS BETWEEN 5 PRECEDING AND CURRENT ROW”.

Limitations and Restrictions

The OVER clause cannot be used with the CHECKSUM aggregate function.

RANGE cannot be used with <unsigned value specification> PRECEDING or <unsigned value specification> FOLLOWING.

Depending on the ranking, aggregate, or analytic function used with the OVER clause, <ORDER BY clause> and/or the <ROWS and RANGE clause> may not be supported.

Examples

A. Using the OVER clause with the ROW_NUMBER function

The following example shows using the OVER clause with ROW_NUMBER function to display a row number for each row within a partition. The ORDER BY clause specified in the OVER clause orders the rows in each partition by the column `SalesYTD`. The ORDER BY clause in the SELECT statement determines the order in which the entire query result set is returned.

```
USE AdventureWorks2012;
GO
SELECT ROW_NUMBER() OVER(PARTITION BY PostalCode ORDER BY SalesYTD DESC) AS "Row Number",
    p.LastName, s.SalesYTD, a.PostalCode
FROM Sales.SalesPerson AS s
    INNER JOIN Person.Person AS p
        ON s.BusinessEntityID = p.BusinessEntityID
    INNER JOIN Person.Address AS a
        ON a.AddressID = p.BusinessEntityID
WHERE TerritoryID IS NOT NULL
    AND SalesYTD <> 0
ORDER BY PostalCode;
GO
```

Here is the result set.

```
Row Number      LastName                SalesYTD              PostalCode
--------------- ----------------------- --------------------- ----------
1               Mitchell                4251368.5497          98027
2               Blythe                  3763178.1787          98027
3               Carson                  3189418.3662          98027
4               Reiter                  2315185.611           98027
5               Vargas                  1453719.4653          98027
6               Ansman-Wolfe            1352577.1325          98027
1               Pak                     4116871.2277          98055
2               Varkey Chudukatil       3121616.3202          98055
3               Saraiva                 2604540.7172          98055
4               Ito                     2458535.6169          98055
5               Valdez                  1827066.7118          98055
6               Mensa-Annan             1576562.1966          98055
7               Campbell                1573012.9383          98055
8               Tsoflias                1421810.9242          98055
```

B. Using the OVER clause with aggregate functions

The following example uses the `OVER` clause with aggregate functions over all rows returned by the query. In this example, using the `OVER` clause is more efficient than using subqueries to derive the aggregate values.

```
USE AdventureWorks2012;
GO
SELECT SalesOrderID, ProductID, OrderQty
    ,SUM(OrderQty) OVER(PARTITION BY SalesOrderID) AS Total
    ,AVG(OrderQty) OVER(PARTITION BY SalesOrderID) AS "Avg"
    ,COUNT(OrderQty) OVER(PARTITION BY SalesOrderID) AS "Count"
    ,MIN(OrderQty) OVER(PARTITION BY SalesOrderID) AS "Min"
    ,MAX(OrderQty) OVER(PARTITION BY SalesOrderID) AS "Max"
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN(43659,43664);
GO
```

Here is the result set.

```
SalesOrderID ProductID   OrderQty Total       Avg         Count       Min    Max
------------ ----------- -------- ----------- ----------- ----------- ------ ------
43659        776         1        26          2           12          1      6
43659        777         3        26          2           12          1      6
43659        778         1        26          2           12          1      6
43659        771         1        26          2           12          1      6
43659        772         1        26          2           12          1      6
43659        773         2        26          2           12          1      6
43659        774         1        26          2           12          1      6
43659        714         3        26          2           12          1      6
43659        716         1        26          2           12          1      6
43659        709         6        26          2           12          1      6
43659        712         2        26          2           12          1      6
43659        711         4        26          2           12          1      6
43664        772         1        14          1           8           1      4
43664        775         4        14          1           8           1      4
43664        714         1        14          1           8           1      4
43664        716         1        14          1           8           1      4
43664        777         2        14          1           8           1      4
43664        771         3        14          1           8           1      4
43664        773         1        14          1           8           1      4
43664        778         1        14          1           8           1      4
```

The following example shows using the `OVER` clause with an aggregate function in a calculated value.

```
USE AdventureWorks2012;
GO
SELECT SalesOrderID, ProductID, OrderQty
    ,SUM(OrderQty) OVER(PARTITION BY SalesOrderID) AS Total
    ,CAST(1. * OrderQty / SUM(OrderQty) OVER(PARTITION BY SalesOrderID)
        *100 AS DECIMAL(5,2))AS "Percent by ProductID"
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN(43659,43664);
GO
```

Here is the result set. Notice that the aggregates are calculated by `SalesOrderID` and the `Percent by ProductID` is calculated for each line of each `SalesOrderID`.

```
SalesOrderID ProductID   OrderQty Total       Percent by ProductID
------------ ----------- -------- ----------- ---------------------------------------
43659        776         1        26          3.85
43659        777         3        26          11.54
43659        778         1        26          3.85
43659        771         1        26          3.85
43659        772         1        26          3.85
43659        773         2        26          7.69
43659        774         1        26          3.85
43659        714         3        26          11.54
43659        716         1        26          3.85
43659        709         6        26          23.08
43659        712         2        26          7.69
43659        711         4        26          15.38
43664        772         1        14          7.14
43664        775         4        14          28.57
43664        714         1        14          7.14
43664        716         1        14          7.14
43664        777         2        14          14.29
43664        771         3        14          21.4
43664        773         1        14          7.14
43664        778         1        14          7.14

 (20 row(s) affected)
```

C. Producing a moving average and cumulative total

The following example uses the AVG and SUM functions with the OVER clause to provide a moving average and cumulative total of yearly sales for each territory in the `Sales.SalesPerson` table. The data is partitioned by `TerritoryID` and logically ordered by `SalesYTD`. This means that the AVG function is computed for each territory based on the sales year. Notice that for `TerritoryID` 1, there are two rows for sales year 2005 representing the two sales people with sales that year. The average sales for these two rows is computed and then the third row representing sales for the year 2006 is included in the computation.

```
USE AdventureWorks2012;
GO
SELECT BusinessEntityID, TerritoryID
   ,DATEPART(yy,ModifiedDate) AS SalesYear
   ,CONVERT(varchar(20),SalesYTD,1) AS  SalesYTD
   ,CONVERT(varchar(20),AVG(SalesYTD) OVER (PARTITION BY TerritoryID
                                            ORDER BY DATEPART(yy,ModifiedDate)
                                           ),1) AS MovingAvg
   ,CONVERT(varchar(20),SUM(SalesYTD) OVER (PARTITION BY TerritoryID
                                            ORDER BY DATEPART(yy,ModifiedDate)
                                            ),1) AS CumulativeTotal
FROM Sales.SalesPerson
WHERE TerritoryID IS NULL OR TerritoryID < 5
ORDER BY TerritoryID,SalesYear;
```

Here is the result set.

```
BusinessEntityID TerritoryID SalesYear   SalesYTD             MovingAvg
CumulativeTotal
---------------- ----------- ----------- -------------------- -------------------- ------
--------------
274              NULL        2005        559,697.56           559,697.56
559,697.56
287              NULL        2006        519,905.93           539,801.75
1,079,603.50
285              NULL        2007        172,524.45           417,375.98
1,252,127.95
283              1           2005        1,573,012.94         1,462,795.04
2,925,590.07
280              1           2005        1,352,577.13         1,462,795.04
2,925,590.07
284              1           2006        1,576,562.20         1,500,717.42
4,502,152.27
275              2           2005        3,763,178.18         3,763,178.18
3,763,178.18
277              3           2005        3,189,418.37         3,189,418.37
3,189,418.37
276              4           2005        4,251,368.55         3,354,952.08
6,709,904.17
281              4           2005        2,458,535.62         3,354,952.08
6,709,904.17

(10 row(s) affected)
```

In this example, the OVER clause does not include PARTITION BY. This means that the function will be applied to all rows returned by the query. The ORDER BY clause specified in the OVER clause determines the logical order to which the AVG function is applied. The query returns a moving average of sales by year for all sales territories specified in the WHERE clause. The ORDER BY clause specified in the SELECT statement determines the order in which the rows of the query are displayed.

```
SELECT BusinessEntityID, TerritoryID
   ,DATEPART(yy,ModifiedDate) AS SalesYear
   ,CONVERT(varchar(20),SalesYTD,1) AS  SalesYTD
   ,CONVERT(varchar(20),AVG(SalesYTD) OVER (ORDER BY DATEPART(yy,ModifiedDate)
                                           ),1) AS MovingAvg
   ,CONVERT(varchar(20),SUM(SalesYTD) OVER (ORDER BY DATEPART(yy,ModifiedDate)
                                           ),1) AS CumulativeTotal
FROM Sales.SalesPerson
WHERE TerritoryID IS NULL OR TerritoryID < 5
ORDER BY SalesYear;
```

Here is the result set.

```
BusinessEntityID TerritoryID SalesYear   SalesYTD             MovingAvg
CumulativeTotal
---------------- ----------- ----------- -------------------- -------------------- ------
--------------
274              NULL        2005        559,697.56           2,449,684.05
17,147,788.35
275              2           2005        3,763,178.18         2,449,684.05
17,147,788.35
276              4           2005        4,251,368.55         2,449,684.05
17,147,788.35
277              3           2005        3,189,418.37         2,449,684.05
17,147,788.35
280              1           2005        1,352,577.13         2,449,684.05
17,147,788.35
281              4           2005        2,458,535.62         2,449,684.05
17,147,788.35
283              1           2005        1,573,012.94         2,449,684.05
17,147,788.35
284              1           2006        1,576,562.20         2,138,250.72
19,244,256.47
287              NULL        2006        519,905.93           2,138,250.72
19,244,256.47
285              NULL        2007        172,524.45           1,941,678.09
19,416,780.93
(10 row(s) affected)
```

D. Specifying the ROWS clause

The following example uses the ROWS clause to define a window over which the rows are computed as the current row and the *N* number of rows that follow (1 row in this example).

```
SELECT BusinessEntityID, TerritoryID
    ,CONVERT(varchar(20),SalesYTD,1) AS  SalesYTD
    ,DATEPART(yy,ModifiedDate) AS SalesYear
    ,CONVERT(varchar(20),SUM(SalesYTD) OVER (PARTITION BY TerritoryID
                                             ORDER BY DATEPART(yy,ModifiedDate)
                                             ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING
),1) AS CumulativeTotal
FROM Sales.SalesPerson
WHERE TerritoryID IS NULL OR TerritoryID < 5;
```

Here is the result set.

```
BusinessEntityID TerritoryID SalesYTD             SalesYear   CumulativeTotal
---------------- ----------- -------------------- ----------- --------------------
274              NULL        559,697.56           2005        1,079,603.50
287              NULL        519,905.93           2006        692,430.38
285              NULL        172,524.45           2007        172,524.45
283              1           1,573,012.94         2005        2,925,590.07
280              1           1,352,577.13         2005        2,929,139.33
284              1           1,576,562.20         2006        1,576,562.20
275              2           3,763,178.18         2005        3,763,178.18
277              3           3,189,418.37         2005        3,189,418.37
276              4           4,251,368.55         2005        6,709,904.17
281              4           2,458,535.62         2005        2,458,535.62
```

In the following example, the ROWS clause is specified with UNBOUNDED PRECEDING. The result is that the window starts at the first row of the partition.

```
SELECT BusinessEntityID, TerritoryID
    ,CONVERT(varchar(20),SalesYTD,1) AS  SalesYTD
    ,DATEPART(yy,ModifiedDate) AS SalesYear
    ,CONVERT(varchar(20),SUM(SalesYTD) OVER (PARTITION BY TerritoryID
                                             ORDER BY DATEPART(yy,ModifiedDate)
                                             ROWS UNBOUNDED PRECEDING),1) AS
CumulativeTotal
FROM Sales.SalesPerson
WHERE TerritoryID IS NULL OR TerritoryID < 5;
```

Here is the result set.

```
BusinessEntityID TerritoryID SalesYTD             SalesYear   CumulativeTotal
---------------- ----------- -------------------- ----------- --------------------
274              NULL        559,697.56           2005        559,697.56
287              NULL        519,905.93           2006        1,079,603.50
285              NULL        172,524.45           2007        1,252,127.95
283              1           1,573,012.94         2005        1,573,012.94
280              1           1,352,577.13         2005        2,925,590.07
284              1           1,576,562.20         2006        4,502,152.27
275              2           3,763,178.18         2005        3,763,178.18
277              3           3,189,418.37         2005        3,189,418.37
276              4           4,251,368.55         2005        4,251,368.55
281              4           2,458,535.62         2005        6,709,904.17
```

See Also

[Aggregate Functions (Transact-SQL)]

[Analytic Functions (Transact-SQL)]

Table Value Constructor

Specifies a set of row value expressions to be constructed into a table. The Transact-SQL table value constructor allows multiple rows of data to be specified in a single DML statement. The table value constructor can be specified in the VALUES clause of the INSERT statement, in the USING <source table> clause of the MERGE statement, and in the definition of a derived table in the FROM clause.

Transact-SQL Syntax Conventions

Syntax

```
VALUES ( <row value expression list>) [ ,...n ]

<row value expression list> ::=
    {<row value expression> } [ ,...n ]

<row value expression> ::=
    { DEFAULT | NULL | expression }
```

Arguments

VALUES

Introduces the row value expression lists. Each list must be enclosed in parentheses and separated by a comma.

The number of values specified in each list must be the same and the values must be in the same order as the columns in the table. A value for each column in the table must be specified or the column list must explicitly specify the columns for each incoming value.

DEFAULT

Forces the Database Engine to insert the default value defined for a column. If a default does not exist for the column and the column allows null values, NULL is inserted. DEFAULT is not valid for an identity column. When specified in a table value constructor, DEFAULT is allowed only in an INSERT statement.

expression

Is a constant, a variable, or an expression. The expression cannot contain an EXECUTE statement.

Limitations and Restrictions

The maximum number of rows that can be constructed using the table value constructor is 1000. To insert more than 1000 rows, create multiple INSERT statements, or bulk import the data by using the **bcp** utility or the BULK INSERT statement.

Only single scalar values are allowed as a row value expression. A subquery that involves multiple columns is not allowed as a row value expression. For example, the following code results in a syntax error because the third row value expression list contains a subquery with multiple columns.

```
USE AdventureWorks2012;
GO
CREATE TABLE dbo.MyProducts (Name varchar(50), ListPrice money);
GO
-- This statement fails because the third values list contains multiple columns in the
subquery.
INSERT INTO dbo.MyProducts (Name, ListPrice)
VALUES ('Helmet', 25.50),
       ('Wheel', 30.00),
       (SELECT Name, ListPrice FROM Production.Product WHERE ProductID = 720);
GO
```

However, the statement can be rewritten by specifying each column in the subquery separately. The following example successfully inserts three rows into the `MyProducts` table.

```
INSERT INTO dbo.MyProducts (Name, ListPrice)
VALUES ('Helmet', 25.50),
       ('Wheel', 30.00),
       ((SELECT Name FROM Production.Product WHERE ProductID = 720),
        (SELECT ListPrice FROM Production.Product WHERE ProductID = 720));
GO
```

Data Types

The values specified in a multi-row INSERT statement follow the data type conversion properties of the UNION ALL syntax. This results in the implicit conversion of unmatched types to the type of higher precedence. If the conversion is not a supported implicit conversion, an error is returned.

For example, the following statement inserts an integer value and a character value into a column of type **char**.

```
CREATE TABLE dbo.t (a int, b char);

GO

INSERT INTO dbo.t VALUES (1,'a'), (2, 1);

GO
```

When the INSERT statement is run, SQL Server tries to convert 'a' to an integer because the data type precedence indicates that an integer is of a higher type than a character. The conversion fails and an error is returned. You can avoid the error by explicitly converting values as appropriate. For example, the previous statement can be written as follows.

```
INSERT INTO dbo.t VALUES (1,'a'), (2, CONVERT(CHAR,1));
```

Examples

A. Inserting multiple rows of data

The following example creates the table `dbo.Departments` and then uses the table value constructor to insert five rows into the table. Because values for all columns are supplied and are listed in the same order as the columns in the table, the column names do not have to be specified in the column list.

```
USE AdventureWorks2012;

GO

INSERT INTO Production.UnitMeasure

VALUES (N'FT2', N'Square Feet ', '20080923'), (N'Y', N'Yards', '20080923'), (N'Y3',
N'Cubic Yards', '20080923');

GO
```

B. Inserting multiple rows with DEFAULT and NULL values

The following example demonstrates specifying DEFAULT and NULL when using the table value constructor to insert rows into a table.

```
USE AdventureWorks2012;

GO

CREATE TABLE Sales.MySalesReason(

    SalesReasonID int IDENTITY(1,1) NOT NULL,

    Name dbo.Name NULL ,

    ReasonType dbo.Name NOT NULL DEFAULT 'Not Applicable' );

GO

INSERT INTO Sales.MySalesReason

VALUES ('Recommendation','Other'), ('Advertisement', DEFAULT), (NULL, 'Promotion');


SELECT * FROM Sales.MySalesReason;
```

C. Specifying multiple values as a derived table in a FROM clause

The following example uses the table value constructor to specify multiple values in the FROM clause of a SELECT statement.

```
SELECT a, b FROM (VALUES (1, 2), (3, 4), (5, 6), (7, 8), (9, 10) ) AS MyTable(a, b);
GO
```

D. Specifying multiple values as a derived source table in a MERGE statement

The following example uses MERGE to modify the `SalesReason` table by either updating or inserting rows. When the value of `NewName` in the source table matches a value in the `Name` column of the target table, (`SalesReason`), the `ReasonType` column is updated in the target table. When the value of `NewName` does not match, the source row is inserted into the target table. The source table is a derived table that uses the Transact-SQL table value constructor to specify multiple rows for the source table.

```
USE AdventureWorks2012;
GO
-- Create a temporary table variable to hold the output actions.
DECLARE @SummaryOfChanges TABLE(Change VARCHAR(20));

MERGE INTO Sales.SalesReason AS Target
USING (VALUES ('Recommendation','Other'), ('Review', 'Marketing'), ('Internet', 
'Promotion'))
       AS Source (NewName, NewReasonType)
ON Target.Name = Source.NewName
WHEN MATCHED THEN
    UPDATE SET ReasonType = Source.NewReasonType
WHEN NOT MATCHED BY TARGET THEN
    INSERT (Name, ReasonType) VALUES (NewName, NewReasonType)
OUTPUT $action INTO @SummaryOfChanges;

-- Query the results of the table variable.
SELECT Change, COUNT(*) AS CountPerChange
FROM @SummaryOfChanges
GROUP BY Change;
```

See Also

[INSERT (Transact-SQL)]

[MERGE (Transact-SQL)]

[FROM (Transact-SQL)]

TOP

Limits the rows returned in a query result set to a specified number of rows or percentage of rows in SQL Server 2012. When TOP is used in conjunction with the ORDER BY clause, the result set is limited to the first *N* number of ordered rows; otherwise, it returns the first *N* number of random rows. Use this clause to specify the number of rows returned from a SELECT statement or affected by an INSERT, UPDATE, MERGE, or DELETE statement.

Transact-SQL Syntax Conventions

Syntax

```
[ 
    TOP (expression) [PERCENT]
    [ WITH TIES ]
]
```

Arguments

expression

Is the numeric expression that specifies the number of rows to be returned.expression is implicitly converted to a **float** value if PERCENT is specified; otherwise, it is converted to **bigint**.

PERCENT

Indicates that the query returns only the first expression percent of rows from the result set. Fractional values are rounded up to the next integer value.

WITH TIES

Specifies that the query result set includes any additional rows that match the values in the ORDER BY column or columns in the last row returned. This may cause more rows to be returned than the value specified in expression. For example, if expression is set to 5 but 2 additional rows match the values of the ORDER BY columns in row 5, the result set will contain 7 rows.

TOP...WITH TIES can be specified only in SELECT statements, and only if an ORDER BY clause is specified. The returned order of tying records is arbitrary. ORDER BY does not affect this rule.

Best Practices

In a SELECT statement, always use an ORDER BY clause with the TOP clause. This is the only way to predictably indicate which rows are affected by TOP.

Use OFFSET and FETCH in the ORDER BY clause instead of the TOP clause to implement a query paging solution. A paging solution (that is, sending chunks or "pages" of data to the client) is easier to implement using OFFSET and FETCH clauses. For more information, see ORDER BY Clause (Transact-SQL).

Use TOP (or OFFSET and FETCH) instead of SET ROWCOUNT to limit the number of rows returned. These methods are preferred over using SET ROWCOUNT for the following reasons:

- In SQL Server 2012, SET ROWCOUNT does not affect DELETE, INSERT, MERGE, and UPDATE statements.
- As a part of a SELECT statement, the query optimizer can consider the value of *expression* in the TOP or FETCH clauses during query optimization. Because SET ROWCOUNT is used outside a statement that executes a query, its value cannot be considered in a query plan.

Compatibility Support

For backward compatibility, the parentheses are optional in SELECT statements. We recommend that you always use parentheses for TOP in SELECT statements for consistency with its required use in INSERT, UPDATE, MERGE, and DELETE statements in which the parentheses are required.

Interoperability

The TOP expression does not affect statements that may be executed because of a trigger. The **inserted** and **deleted** tables in the triggers will return only the rows that were truly affected by the INSERT, UPDATE, MERGE, or DELETE statements. For example, if an INSERT TRIGGER is fired as the result of an INSERT statement that used a TOP clause,

SQL Server allows for updating rows through views. Because the TOP clause can be included in the view definition, certain rows may disappear from the view because of an update if the rows no longer meet the requirements of the TOP expression.

When specified in the MERGE statement, the TOP clause is applied *after* the entire source table and the entire target table are joined and the joined rows that do not qualify for an insert, update, or delete action are removed. The TOP clause further reduces the number of joined rows to the specified value and the insert, update, or delete actions are applied to the remaining joined rows in an unordered fashion. That is, there is no order in which the rows are distributed among the actions defined in the WHEN clauses. For example, if specifying TOP (10) affects 10 rows; of these rows, 7 may be updated and 3 inserted, or 1 may be deleted, 5 updated, and 4 inserted, and so on. Because the MERGE statement performs a full table scan of both the source and target tables, I/O performance can be affected when using the TOP clause to modify a large table by creating multiple batches. In this scenario, it is important to ensure that all successive batches target new rows.

Use caution when specifying the TOP clause in a query that contains a UNION, UNION ALL, EXCEPT, or INTERSECT operator. It is possible to write a query that returns unexpected results because the order in which the TOP and ORDER BY clauses are logically processed is not always intuitive when these operators are used in a select operation. For example, given the following table and data, assume that you want to return the least expensive red car and the least expensive blue car. That is, the red sedan and the blue van.

```
CREATE TABLE dbo.Cars(Model varchar(15), Price money, Color varchar(10));
INSERT dbo.Cars VALUES
    ('sedan', 10000, 'red'), ('convertible', 15000, 'blue'),
    ('coupe', 20000, 'red'), ('van', 8000, 'blue');
```

To achieve these results, you might write the following query.

```
SELECT TOP(1) Model, Color, Price
FROM dbo.Cars
WHERE Color = 'red'
UNION ALL
SELECT TOP(1) Model, Color, Price
FROM dbo.Cars
WHERE Color = 'blue'
ORDER BY Price ASC;
```

Here is the result set.

```
Model         Color      Price
------------- ---------- -------
sedan         red        10000.00
convertible   blue       15000.00
```

The unexpected results are returned because the TOP clause is logically executed before the ORDER BY clause, which sorts the results of the operator (UNION ALL in this case). Thus, the previous query returns any one red car and any one blue car and then orders the result of that union by the price. The following example shows the correct method of writing this query to achieve the desired result.

```
SELECT Model, Color, Price
FROM (SELECT TOP(1) Model, Color, Price
      FROM dbo.Cars
      WHERE Color = 'red'
      ORDER BY Price ASC) AS a
UNION ALL
SELECT Model, Color, Price
FROM (SELECT TOP(1) Model, Color, Price
      FROM dbo.Cars
      WHERE Color = 'blue'
      ORDER BY Price ASC) AS b;
```

By using TOP and ORDER BY in a subselect operation, you ensure that the results of the ORDER BY clause is used applied to the TOP clause and not to sorting the result of the UNION operation.

Here is the result set.

```
Model         Color      Price
------------- ---------- -------
sedan         red        10000.00
van           blue        8000.00
```

Limitations and Restrictions

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

TOP cannot be used in an UPDATE and DELETE statements on partitioned views.

TOP cannot be combined with OFFSET and FETCH in the same query expression (in the same query scope). For more information, see ORDER BY Clause (Transact-SQL).

Examples

Category	Featured syntax elements
Basic syntax	TOP • PERCENT
Including tie values	WITH TIES
Limiting the rows affected by DELETE, INSERT, or UPDATE	DELETE • INSERT • UPDATE

Basic syntax

Examples in this section demonstrate the basic functionality of the ORDER BY clause using the minimum required syntax.

A. Using TOP with a constant value

The following examples use a constant value to specify the number of employees that are returned in the query result set. In the first example, the first 10 random rows are returned because an ORDER BY clause is not used. In the second example, an ORDER BY clause is used to return the top 10 recently hired employees.

```
USE AdventureWorks2012;
```

```
GO
-- Select the first 10 random employees.
SELECT TOP(10)JobTitle, HireDate
FROM HumanResources.Employee;
GO
-- Select the first 10 employees hired most recently.
SELECT TOP(10)JobTitle, HireDate
FROM HumanResources.Employee
ORDER BY HireDate DESC;
```

B. Using TOP with a variable

The following example uses a variable to specify the number of employees that are returned in the query result set.

```
USE AdventureWorks2012;
GO
DECLARE @p AS int = 10;
SELECT TOP(@p)JobTitle, HireDate, VacationHours
FROM HumanResources.Employee
ORDER BY VacationHours DESC
GO
```

C. Specifying a percentage

The following example uses PERCENT to specify the number of employees that are returned in the query result set. There are 290 employees in the `HumanResources.Employee` table. Because 5 percent of 290 is a fractional value, the value is rounded up to the next whole number.

```
USE AdventureWorks2012;
GO
SELECT TOP(5)PERCENT JobTitle, HireDate
FROM HumanResources.Employee
ORDER BY HireDate DESC;
```

Including tie values

A. Using WITH TIES to include rows that match the values in the last row

The following example obtains the top `10` percent of all employees with the highest salary and returns them in descending order according to their salary. Specifying `WITH TIES` makes sure that any employees that have salaries equal to the lowest salary returned (the last row) are also included in the result set, even if doing this exceeds `10` percent of employees.

```
USE AdventureWorks2012;
GO
SELECT TOP(10)WITH TIES
pp.FirstName, pp.LastName, e.JobTitle, e.Gender, r.Rate
FROM Person.Person AS pp
```

```
    INNER JOIN HumanResources.Employee AS e
        ON pp.BusinessEntityID = e.BusinessEntityID
    INNER JOIN HumanResources.EmployeePayHistory AS r
        ON r.BusinessEntityID = e.BusinessEntityID
ORDER BY Rate DESC;
```

Limiting the rows affected by DELETE, INSERT, or UPDATE

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, 2002.

```
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.

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

B. Using TOP to limit the number of rows inserted

The following example creates the table `EmployeeSales` and inserts the name and year-to-date sales data for the top 5 random employees from the table `HumanResources.Employee`. The INSERT statement chooses any 5 rows returned by the `SELECT` statement. The OUTPUT clause displays the rows that are inserted into the `EmployeeSales` table. Notice that the ORDER BY clause in the SELECT statement is not used to determine the top 5 employees.

```
USE AdventureWorks2012 ;
GO
```

```
IF OBJECT_ID ('dbo.EmployeeSales', 'U') IS NOT NULL
    DROP TABLE dbo.EmployeeSales;
GO
CREATE TABLE dbo.EmployeeSales
( EmployeeID   nvarchar(11) NOT NULL,
  LastName     nvarchar(20) NOT NULL,
  FirstName    nvarchar(20) NOT NULL,
  YearlySales  money NOT NULL
 );
GO
INSERT TOP(5)INTO dbo.EmployeeSales
    OUTPUT inserted.EmployeeID, inserted.FirstName, inserted.LastName,
inserted.YearlySales
    SELECT sp.BusinessEntityID, c.LastName, c.FirstName, sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.SalesYTD > 250000.00
    ORDER BY sp.SalesYTD DESC;
```

If you have to use TOP to insert rows in a meaningful chronological order, you must use TOP together with ORDER BY in a subselect statement as shown in the following example. The OUTPUT clause displays the rows that are inserted into the `EmployeeSales` table. Notice that the top 5 employees are now inserted based on the results of the ORDER BY clause instead of random rows.

```
INSERT INTO dbo.EmployeeSales
    OUTPUT inserted.EmployeeID, inserted.FirstName, inserted.LastName,
inserted.YearlySales
    SELECT TOP (5) sp.BusinessEntityID, c.LastName, c.FirstName, sp.SalesYTD
    FROM Sales.SalesPerson AS sp
    INNER JOIN Person.Person AS c
        ON sp.BusinessEntityID = c.BusinessEntityID
    WHERE sp.SalesYTD > 250000.00
    ORDER BY sp.SalesYTD DESC;
```

B. Using TOP to limit the number of rows updated

The following example uses the TOP clause to

When a TOP (n) clause is used with UPDATE, the update operation will be performed on a random selection of 'n' number of rows. The following example assigns a random number of 10 customers from one salesperson to another.

```
USE AdventureWorks2012;
UPDATE TOP (10) Sales.Store
```

```
SET SalesPersonID = 276
WHERE SalesPersonID = 275;
GO
```

If you have to 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.

```
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
```

See Also

SELECT (Transact-SQL)
INSERT (Transact-SQL)
UPDATE (Transact-SQL)
DELETE (Transact-SQL)
ORDER BY Clause (Transact-SQL)
SET ROWCOUNT (Transact-SQL)
MERGE (Transact-SQL)

UPDATE

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

Syntax

```
[ 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 Updating Data in a Table.

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_orview_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 Hint (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.

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 Hint (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`.

```
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`.

```
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.

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.

```
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)

-- 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 •

Category	Featured syntax elements
	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.

```
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.

```
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'.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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`.

```
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 Modifying Data Through a View.

```
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.

```
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.

```
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.

```
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:

```
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`.

```
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 OPENQUERYrowset function. The linked server name created in the previous example is used in this example.

```
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 OPENDATASOURCErowset 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 Option.

```
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.

```
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.

```
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.

```
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).

```
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.

```
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.

```
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).

```
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`.

```
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.

```
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

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)

FILESTREAM (SQL Server)

UPDATETEXT

Updates an existing **text**, **ntext**, or **image** field. Use UPDATETEXT to change only a part of a **text**, **ntext**, or **image** column in place. Use WRITETEXT to update and replace a whole **text**, **ntext**, or **image** field.

Important
This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Use the large-value data types and the **.**WRITE clause of the UPDATE statement instead.

Transact-SQL Syntax Conventions

Syntax

```
UPDATETEXT [BULK] { table_name.dest_column_name dest_text_ptr }
  { NULL | insert_offset }
   { NULL | delete_length }
   [ WITH LOG ]
   [ inserted_data
  | { table_name.src_column_namesrc_text_ptr } ]
```

Arguments

BULK

Enables upload tools to upload a binary data stream. The stream must be provided by the tool at the TDS protocol level. When the data stream is not present the query processor ignores the BULK option.

Important
We recommend that the BULK option not be used in SQL Server-based applications. This option might be changed or removed in a future version of SQL Server.

table_name .dest_column_name

Is the name of the table and **text**, **ntext**, or **image** column to be updated. Table names and column names must comply with the rules for identifiers. Specifying the database name and owner names is optional.

dest_text_ptr

Is a text pointer value (returned by the TEXTPTR function) that points to the **text**, **ntext**, or **image** data to be updated. dest_text_ptr must be **binary(**16**)**.

insert_offset

Is the zero-based starting position for the update. For **text** or **image** columns, insert_offset is the number of bytes to skip from the start of the existing column before inserting new data. For **ntext** columns, insert_offsetis the number of characters (each **ntext** character uses 2 bytes). The existing **text**, **ntext**, or **image** data starting at this zero-based starting position is shifted to the right to make room for the new data. A value of 0 inserts the new data at the beginning of the existing data. A value of NULL appends the new data to the existing data value.

delete_length

Is the length of data to delete from the existing **text**, **ntext**, or **image** column, starting at the insert_offset position. The delete_lengthvalue is specified in bytes for **text** and

image columns and in characters for **ntext** columns. Each **ntext** character uses 2 bytes. A value of 0 deletes no data. A value of NULL deletes all data from the insert_offset position to the end of the existing **text** or **image** column.

WITH LOG

Logging is determined by the recovery model in effect for the database.

inserted_data

Is the data to be inserted into the existing **text**, **ntext**, or **image** column at the insert_offsetlocation. This is a single **char**, **nchar**, **varchar**, **nvarchar**, **binary**, **varbinary**, **text**, **ntext**, or **image** value. inserted_data can be a literal or a variable.

table_name.src_column_name

Is the name of the table and **text**, **ntext**, or **image** column used as the source of the inserted data. Table names and column names must comply with the rules for identifiers.

src_text_ptr

Is a text pointer value (returned by the TEXTPTR function) that points to a **text**, **ntext**, or **image** column used as the source of the inserted data.

Note

scr_text_ptrvalue must not be the same as dest_text_ptrvalue.

Remarks

Newly inserted data can be a single inserted_data constant, table name, column name, or text pointer.

Update action	UPDATETEXT parameters
To replace existing data	Specify a nonnullinsert_offset value, a nonzero delete_length value, and the new data to be inserted.
To delete existing data	Specify a nonnullinsert_offset value and a nonzero delete_length. Do not specify new data to be inserted.
To insert new data	Specify the insert_offset value, a delete_length

Update action	UPDATETEXT parameters
	of 0, and the new data to be inserted.

For best performance we recommend that **text**, **ntext** and **image** data be inserted or updated in chunks sizes that are multiples of 8,040 bytes.

In SQL Server, in-row text pointers to **text**, **ntext**, or **image** data may exist but may not be valid. For information about the text in row option, see sp_tableoption. For information about invalidating text pointers, see sp_invalidate_textptr.

To initialize **text** columns to NULL, use WRITETEXT; UPDATETEXT initializes **text** columns to an empty string.

Permissions

Requires UPDATE permission on the specified table.

Examples

The following example puts the text pointer into the local variable `@ptrval`, and then uses `UPDATETEXT` to update a spelling error.

Note

To run this example, you must install the pubs database.

```
USE pubs;
GO
ALTER DATABASE pubs SET RECOVERY SIMPLE;
GO
DECLARE @ptrval binary(16);
SELECT @ptrval = TEXTPTR(pr_info)
   FROM pub_info pr, publishers p
      WHERE p.pub_id = pr.pub_id
      AND p.pub_name = 'New Moon Books'
UPDATETEXT pub_info.pr_info @ptrval 88 1 'b';
GO
ALTER DATABASE pubs SET RECOVERY FULL;
GO
```

See Also

READTEXT

TEXTPTR

WRITETEXT

WHERE

Specifies the search condition for the rows returned by the query.

Transact-SQL Syntax Conventions

Syntax

```
[ WHERE <search_condition> ]
```

Arguments

<search_condition>

Defines the condition to be met for the rows to be returned. There is no limit to the number of predicates that can be included in a search condition. For more information about search conditions and predicates, see Search Condition.

Examples

The following examples show how to use some common search conditions in the `WHERE` clause.

A. Finding a row by using a simple equality

```
USE AdventureWorks2012
GO
SELECT ProductID, Name
FROM Production.Product
WHERE Name = 'Blade' ;
GO
```

B. Finding rows that contain a value as a part of a string

```
SELECT ProductID, Name, Color
FROM Production.Product
WHERE Name LIKE ('%Frame%');
GO
```

C. Finding rows by using a comparison operator

```
SELECT ProductID, Name
FROM Production.Product
WHERE ProductID <= 12 ;
GO
```

D. Finding rows that meet any of three conditions

```
SELECT ProductID, Name
FROM Production.Product
WHERE ProductID = 2
OR ProductID = 4
OR Name = 'Spokes' ;
GO
```

E. Finding rows that must meet several conditions

```
SELECT ProductID, Name, Color
FROM Production.Product
WHERE Name LIKE ('%Frame%')
AND Name LIKE ('HL%')
AND Color = 'Red' ;
GO
```

F. Finding rows that are in a list of values

```
SELECT ProductID, Name, Color
FROM Production.Product
WHERE Name IN ('Blade', 'Crown Race', 'Spokes');
GO
```

G. Finding rows that have a value between two values

```
SELECT ProductID, Name, Color
FROM Production.Product
WHERE ProductID BETWEEN 725 AND 734;
GO
```

See Also

DELETE

Predicate

Search Condition

SELECT

UPDATE

MERGE

WITH common_table_expression

Specifies a temporary named result set, known as a common table expression (CTE). This is derived from a simple query and defined within the execution scope of a single SELECT, INSERT, UPDATE, or DELETE statement. This clause can also be used in a CREATE VIEW statement as part of its defining SELECT statement. A common table expression can include references to itself. This is referred to as a recursive common table expression.

Transact-SQL Syntax Conventions

Syntax

```
[ WITH <common_table_expression> [ ,...n ] ]

<common_table_expression>::=
   expression_name [ (column_name [ ,...n ] ) ]
   AS
   (CTE_query_definition)
```

Arguments

expression_name

Is a valid identifier for the common table expression. expression_name must be different from the name of any other common table expression defined in the same WITH <common_table_expression> clause, but expression_name can be the same as the name of a base table or view. Any reference to expression_name in the query uses the common table expression and not the base object.

column_name

Specifies a column name in the common table expression. Duplicate names within a single CTE definition are not allowed. The number of column names specified must match the number of columns in the result set of the CTE_query_definition. The list of column names is optional only if distinct names for all resulting columns are supplied in the query definition.

CTE_query_definition

Specifies a SELECT statement whose result set populates the common table expression. The SELECT statement for CTE_query_definition must meet the same requirements as for creating a view, except a CTE cannot define another CTE. For more information, see the Remarks section and CREATE VIEW (Transact-SQL).

If more than one CTE_query_definition is defined, the query definitions must be joined by one of these set operators: UNION ALL, UNION, EXCEPT, or INTERSECT.

Remarks

Guidelines for Creating and Using Common Table Expressions

The following guidelines apply to nonrecursive common table expressions. For guidelines that apply to recursive common table expressions, see "Guidelines for Defining and Using Recursive Common Table Expressions" that follows.

- A CTE must be followed by a single SELECT, INSERT, UPDATE, or DELETE statement that references some or all the CTE columns. A CTE can also be specified in a CREATE VIEW statement as part of the defining SELECT statement of the view.
- Multiple CTE query definitions can be defined in a nonrecursive CTE. The definitions must be combined by one of these set operators: UNION ALL, UNION, INTERSECT, or EXCEPT.
- A CTE can reference itself and previously defined CTEs in the same WITH clause. Forward referencing is not allowed.
- Specifying more than one WITH clause in a CTE is not allowed. For example, if a CTE_query_definition contains a subquery, that subquery cannot contain a nested WITH clause that defines another CTE.
- The following clauses cannot be used in the CTE_query_definition:
  - ORDER BY (except when a TOP clause is specified)
  - INTO
  - OPTION clause with query hints
  - FOR XML
  - FOR BROWSE
- When a CTE is used in a statement that is part of a batch, the statement before it must be followed by a semicolon.
- A query referencing a CTE can be used to define a cursor.
- Tables on remote servers can be referenced in the CTE.
- When executing a CTE, any hints that reference a CTE may conflict with other hints that are discovered when the CTE accesses its underlying tables, in the same manner as hints that reference views in queries. When this occurs, the query returns an error.

Guidelines for Defining and Using Recursive Common Table Expressions

The following guidelines apply to defining a recursive common table expression:

- The recursive CTE definition must contain at least two CTE query definitions, an anchor member and a recursive member. Multiple anchor members and recursive members can be defined; however, all anchor member query definitions must be put before the first recursive member definition. All CTE query definitionsare anchor members unless they reference the CTE itself.
- Anchor members must be combined by one of these set operators: UNION ALL, UNION, INTERSECT, or EXCEPT. UNION ALL is the only set operator allowed between the last anchor member and first recursive member, and when combining multiple recursive members.
- The number of columns in the anchor and recursive members must be the same.
- The data type of a column in the recursive member must be the same as the data type of the corresponding column in the anchor member.
- The FROM clause of a recursive member must refer only one time to the CTE expression_name.
- The following items are not allowed in the CTE_query_definition of a recursive member:
  - SELECT DISTINCT
  - GROUP BY
  - PIVOT (When the database compatibility level is 110. See Breaking Changes to Database Engine Features in SQL Server "Denali".)
  - HAVING
  - Scalar aggregation
  - TOP
  - LEFT, RIGHT, OUTER JOIN (INNER JOIN is allowed)
  - Subqueries
  - A hint applied to a recursive reference to a CTE inside a CTE_query_definition.

The following guidelines apply to using a recursive common table expression:

- All columns returned by the recursive CTE are nullable regardless of the nullability of the columns returned by the participating SELECT statements.
- An incorrectly composed recursive CTE may cause an infinite loop. For example, if the recursive member query definition returns the same values for both the parent and child columns, an infinite loop is created. To prevent an infinite loop, you can limit the number of recursion levels allowed for a particular statement by using the MAXRECURSION hint and a value between 0 and 32,767 in the OPTION clause of the INSERT, UPDATE, DELETE, or SELECT statement. This lets you control the execution of the statement until you resolve the code problem that is creating the loop. The server-wide default is 100. When 0 is specified, no limit is applied. Only one MAXRECURSION value can be specified per statement. For more information, see Query Hint.
- A view that contains a recursive common table expression cannot be used to update data.

- Cursors may be defined on queries using CTEs. The CTE is the select_statement argument that defines the result set of the cursor. Only fast forward-only and static (snapshot) cursors are allowed for recursive CTEs. If another cursor type is specified in a recursive CTE, the cursor type is converted to static.
- Tables on remote servers may be referenced in the CTE. If the remote server is referenced in the recursive member of the CTE, a spool is created for each remote table so the tables can be repeatedly accessed locally. If it is a CTE query, Index Spool/Lazy Spools is displayed in the query plan and will have the additional WITH STACK predicate. This is one way to confirm proper recursion.
- Analytic and aggregate functions in the recursive part of the CTE are applied to the set for the current recursion level and not to the set for the CTE. Functions like ROW_NUMBER operate only on the subset of data passed to them by the current recursion level and not the entire set of data pased to the recursive part of the CTE. For more information, see J. Using analytical functions in a recursive CTE.

Examples

A. Creating a simple common table expression

The following example shows the total number of sales orders per year for each sales representative at Adventure Works Cycles.

```
USE AdventureWorks2012;
GO
-- Define the CTE expression name and column list.
WITH Sales_CTE (SalesPersonID, SalesOrderID, SalesYear)
AS
-- Define the CTE query.
(
    SELECT SalesPersonID, SalesOrderID, YEAR(OrderDate) AS SalesYear
    FROM Sales.SalesOrderHeader
    WHERE SalesPersonID IS NOT NULL
)
-- Define the outer query referencing the CTE name.
SELECT SalesPersonID, COUNT(SalesOrderID) AS TotalSales, SalesYear
FROM Sales_CTE
GROUP BY SalesYear, SalesPersonID
ORDER BY SalesPersonID, SalesYear;
GO
```

B. Using a common table expression to limit counts and report averages

The following example shows the average number of sales orders for all years for the sales representatives.

```
WITH Sales_CTE (SalesPersonID, NumberOfOrders)
AS
(
    SELECT SalesPersonID, COUNT(*)
    FROM Sales.SalesOrderHeader
    WHERE SalesPersonID IS NOT NULL
    GROUP BY SalesPersonID
)
SELECT AVG(NumberOfOrders) AS "Average Sales Per Person"
FROM Sales_CTE;
GO
```

C. Using a recursive common table expression to display multiple levels of recursion

The following example shows the hierarchical list of managers and the employees who report to them. The example begins by creating and populating the `dbo.MyEmployees` table.

```
-- Create an Employee table.
CREATE TABLE dbo.MyEmployees
(
    EmployeeID smallint NOT NULL,
    FirstName nvarchar(30)  NOT NULL,
    LastName  nvarchar(40) NOT NULL,
    Title nvarchar(50) NOT NULL,
    DeptID smallint NOT NULL,
    ManagerID int NULL,
 CONSTRAINT PK_EmployeeID PRIMARY KEY CLUSTERED (EmployeeID ASC)
);
-- Populate the table with values.
INSERT INTO dbo.MyEmployees VALUES
 (1, N'Ken', N'Sánchez', N'Chief Executive Officer',16,NULL)
,(273, N'Brian', N'Welcker', N'Vice President of Sales',3,1)
,(274, N'Stephen', N'Jiang', N'North American Sales Manager',3,273)
,(275, N'Michael', N'Blythe', N'Sales Representative',3,274)
,(276, N'Linda', N'Mitchell', N'Sales Representative',3,274)
,(285, N'Syed', N'Abbas', N'Pacific Sales Manager',3,273)
,(286, N'Lynn', N'Tsoflias', N'Sales Representative',3,285)
,(16,  N'David',N'Bradley', N'Marketing Manager', 4, 273)
,(23,  N'Mary', N'Gibson', N'Marketing Specialist', 4, 16);


USE AdventureWorks2012;
GO
WITH DirectReports(ManagerID, EmployeeID, Title, EmployeeLevel) AS
```

```
(
    SELECT ManagerID, EmployeeID, Title, 0 AS EmployeeLevel
    FROM dbo.MyEmployees
    WHERE ManagerID IS NULL
    UNION ALL
    SELECT e.ManagerID, e.EmployeeID, e.Title, EmployeeLevel + 1
    FROM dbo.MyEmployees AS e
        INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
SELECT ManagerID, EmployeeID, Title, EmployeeLevel
FROM DirectReports
ORDER BY ManagerID;
GO
```

D. Using a recursive common table expression to display two levels of recursion

The following example shows managers and the employees reporting to them. The number of levels returned is limited to two.

```
USE AdventureWorks2012;
GO
WITH DirectReports(ManagerID, EmployeeID, Title, EmployeeLevel) AS
(
    SELECT ManagerID, EmployeeID, Title, 0 AS EmployeeLevel
    FROM dbo.MyEmployees
    WHERE ManagerID IS NULL
    UNION ALL
    SELECT e.ManagerID, e.EmployeeID, e.Title, EmployeeLevel + 1
    FROM dbo.MyEmployees AS e
        INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
SELECT ManagerID, EmployeeID, Title, EmployeeLevel
FROM DirectReports
WHERE EmployeeLevel <= 2 ;
GO
```

E. Using a recursive common table expression to display a hierarchical list

The following example builds on Example C by adding the names of the manager and employees, and their respective titles. The hierarchy of managers and employees is additionally emphasized by indenting each level.

```
USE AdventureWorks2012;
```

```
GO
WITH DirectReports(Name, Title, EmployeeID, EmployeeLevel, Sort)
AS (SELECT CONVERT(varchar(255), e.FirstName + ' ' + e.LastName),
        e.Title,
        e.EmployeeID,
        1,
        CONVERT(varchar(255), e.FirstName + ' ' + e.LastName)
    FROM dbo.MyEmployees AS e
    WHERE e.ManagerID IS NULL
    UNION ALL
    SELECT CONVERT(varchar(255), REPLICATE ('|    ' , EmployeeLevel) +
        e.FirstName + ' ' + e.LastName),
        e.Title,
        e.EmployeeID,
        EmployeeLevel + 1,
        CONVERT (varchar(255), RTRIM(Sort) + '|    ' + FirstName + ' ' +
                 LastName)
    FROM dbo.MyEmployees AS e
    JOIN DirectReports AS d ON e.ManagerID = d.EmployeeID
    )
SELECT EmployeeID, Name, Title, EmployeeLevel
FROM DirectReports
ORDER BY Sort;
GO
```

F. Using MAXRECURSION to cancel a statement

`MAXRECURSION` can be used to prevent a poorly formed recursive CTE from entering into an infinite loop. The following example intentionally creates an infinite loop and uses the `MAXRECURSION` hint to limit the number of recursion levels to two.

```
USE AdventureWorks2012;
GO
--Creates an infinite loop
WITH cte (EmployeeID, ManagerID, Title) as
(
    SELECT EmployeeID, ManagerID, Title
    FROM dbo.MyEmployees
    WHERE ManagerID IS NOT NULL
  UNION ALL
    SELECT cte.EmployeeID, cte.ManagerID, cte.Title
    FROM cte
    JOIN  dbo.MyEmployees AS e
        ON cte.ManagerID = e.EmployeeID
)
```

```
--Uses MAXRECURSION to limit the recursive levels to 2
SELECT EmployeeID, ManagerID, Title
FROM cte
OPTION (MAXRECURSION 2);
GO
```

After the coding error is corrected, MAXRECURSION is no longer required. The following example shows the corrected code.

```
USE AdventureWorks2012;
GO
WITH cte (EmployeeID, ManagerID, Title)
AS
(
    SELECT EmployeeID, ManagerID, Title
    FROM dbo.MyEmployees
    WHERE ManagerID IS NOT NULL
  UNION ALL
    SELECT  e.EmployeeID, e.ManagerID, e.Title
    FROM dbo.MyEmployees AS e
    JOIN cte ON e.ManagerID = cte.EmployeeID
)
SELECT EmployeeID, ManagerID, Title
FROM cte;
GO
```

G. Using a common table expression to selectively step through a recursive relationship in a SELECT statement

The following example shows the hierarchy of product assemblies and components that are required to build the bicycle for `ProductAssemblyID = 800`.

```
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
)
SELECT AssemblyID, ComponentID, Name, PerAssemblyQty, EndDate,
        ComponentLevel
FROM Parts AS p
    INNER JOIN Production.Product AS pr
    ON p.ComponentID = pr.ProductID
ORDER BY ComponentLevel, AssemblyID, ComponentID;
GO
```

H. Using a recursive CTE in an UPDATE statement

The following example updates the `PerAssemblyQty` value for all parts that are used to build the product 'Road-550-W Yellow, 44' `(ProductAssemblyID800`). The common table expression returns a hierarchical list of parts that are used to build `ProductAssemblyID 800` and the components that are used to create those parts, and so on. Only the rows returned by the common table expression are modified.

```
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;
```

I. Using multiple anchor and recursive members

The following example uses multiple anchor and recursive members to return all the ancestors of a specified person. A table is created and values inserted to establish the family genealogy returned by the recursive CTE.

```
-- Genealogy table
IF OBJECT_ID('dbo.Person','U') IS NOT NULL DROP TABLE dbo.Person;
GO
CREATE TABLE dbo.Person(ID int, Name varchar(30), Mother int, Father int);
GO
INSERT dbo.Person
VALUES(1, 'Sue', NULL, NULL)
      ,(2, 'Ed', NULL, NULL)
      ,(3, 'Emma', 1, 2)
      ,(4, 'Jack', 1, 2)
      ,(5, 'Jane', NULL, NULL)
      ,(6, 'Bonnie', 5, 4)
      ,(7, 'Bill', 5, 4);
GO
-- Create the recursive CTE to find all of Bonnie's ancestors.
WITH Generation (ID) AS
(
-- First anchor member returns Bonnie's mother.
    SELECT Mother
    FROM dbo.Person
    WHERE Name = 'Bonnie'
UNION
-- Second anchor member returns Bonnie's father.
    SELECT Father
    FROM dbo.Person
    WHERE Name = 'Bonnie'
UNION ALL
-- First recursive member returns male ancestors of the previous generation.
    SELECT Person.Father
    FROM Generation, Person
    WHERE Generation.ID=Person.ID
UNION ALL
-- Second recursive member returns female ancestors of the previous generation.
    SELECT Person.Mother
    FROM Generation, dbo.Person
    WHERE Generation.ID=Person.ID
)
SELECT Person.ID, Person.Name, Person.Mother, Person.Father
FROM Generation, dbo.Person
```

```
WHERE Generation.ID = Person.ID;
GO
```

J. Using analytical functions in a recursive CTE

The following example shows a pitfall that can occur when using an analytical or aggregate function in the recursive part of a CTE.

```
DECLARE @t1 TABLE (itmID int, itmIDComp int);
INSERT @t1 VALUES (1,10), (2,10);

DECLARE @t2 TABLE (itmID int, itmIDComp int);
INSERT @t2 VALUES (3,10), (4,10);

WITH vw AS
 (
    SELECT itmIDComp, itmID
    FROM @t1

    UNION ALL

    SELECT itmIDComp, itmID
    FROM @t2
)
,r AS
 (
    SELECT t.itmID AS itmIDComp
           , NULL AS itmID
           ,CAST(0 AS bigint) AS N
           ,1 AS Lvl
    FROM (SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4) AS t (itmID)

UNION ALL

SELECT t.itmIDComp
    , t.itmID
    , ROW_NUMBER() OVER(PARTITION BY t.itmIDComp ORDER BY t.itmIDComp, t.itmID) AS N
    , Lvl + 1
FROM r
    JOIN vw AS t ON t.itmID = r.itmIDComp
)

SELECT Lvl, N FROM r;
```

The following results are the expected results for the query.

```
Lvl  N
1    0
1    0
1    0
1    0
2    4
2    3
2    2
2    1
```

The following results are the actual results for the query.

```
Lvl  N
1    0
1    0
1    0
1    0
2    1
2    1
2    1
2    1
```

`N` returns 1 for each pass of the recursive part of the CTE because only the subset of data for that recursion level is passed to `ROWNUMBER`. For each of the iterations of the recursive part of the query, only one row is passed to `ROWNUMBER`.

See Also

CREATE VIEW

DELETE

EXCEPT and INTERSECT (Transact-SQL)

INSERT

SELECT

UPDATE

WRITETEXT

Permits minimally logged, interactive updating of an existing **text**, **ntext**, or **image** column. WRITETEXT overwrites any existing data in the column it affects. WRITETEXT cannot be used on **text**, **ntext**, and **image** columns in views.

Important

This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Use the large-value data types and the **.**WRITE clause of the UPDATE statement instead.

Transact-SQL Syntax Conventions

Syntax

```
WRITETEXT [BULK]
  { table.column text_ptr }
  [ WITH LOG ] { data }
```

Arguments

BULK

Enables upload tools to upload a binary data stream. The stream must be provided by the tool at the TDS protocol level. When the data stream is not present the query processor ignores the BULK option.

Important

We recommend that the BULK option not be used in SQL Server-based applications. This option might be changed or removed in a future version of SQL Server.

table .column

Is the name of the table and **text**, **ntext**, or **image** column to update. Table and column names must comply with the rules for identifiers. Specifying the database name and owner names is optional.

text_ptr

Is a value that stores the pointer to the **text**, **ntext**, or **image**data.text_ptr must be **binary(16)**.To create a text pointer, execute an INSERT or UPDATE statement with data that is not null for the **text**, **ntext**, or **image** column.

WITH LOG

Ignored by SQL Server. Logging is determined by the recovery model in effect for the database.

data

Is the actual **text**, **ntext** or **image** data to store.data can be a literal or a parameter. The maximum length of text that can be inserted interactively with WRITETEXT is approximately 120 KB for **text**, **ntext**, and **image** data.

Remarks

Use WRITETEXT to replace **text**, **ntext**, and **image** data and UPDATETEXT to modify **text**, **ntext**, and **image** data. UPDATETEXT is more flexible because it changes only a part of a **text**, **ntext**, or **image** column instead of the whole column.

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

If the database recovery model is simple or bulk-logged, **text**, **ntext**, and **image** operations that use WRITETEXT are minimally logged operations when new data is inserted or appended.

Note

Minimal logging is not used when existing values are updated.

For WRITETEXT to work correctly, the column must already contain a valid text pointer.

If the table does not have in row text, SQL Server saves space by not initializing **text** columns when explicit or implicit null values are added in **text** columns with INSERT, and no text pointer can be obtained for such nulls. To initialize **text** columns to NULL, use the UPDATE statement. If the table has in row text, you do not have to initialize the text column for nulls and you can always get a text pointer.

The ODBC **SQLPutData** function is faster and uses less dynamic memory than WRITETEXT. This function can insert up to 2 gigabytes of **text**, **ntext**, or **image** data.

In SQL Server, in row text pointers to **text**, **ntext**, or **image** data may exist but may not be valid. For information about the text in row option, see sp_tableoption. For information about invalidating text pointers, see sp_invalidate_textptr.

Permissions

Requires UPDATE permission on the specified table. Permission is transferable when UPDATE permission is transferred.

Examples

The following example puts the text pointer into the local variable `@ptrval`, and then `WRITETEXT` places the new text string into the row pointed to by `@ptrval`.

Note

To run this example, you must install the pubs sample database.

```
USE pubs;
```

```
GO
ALTER DATABASE pubs SET RECOVERY SIMPLE;
GO
DECLARE @ptrval binary(16);
SELECT @ptrval = TEXTPTR(pr_info)
FROM pub_info pr, publishers p
WHERE p.pub_id = pr.pub_id
   AND p.pub_name = 'New Moon Books'
WRITETEXT pub_info.pr_info @ptrval 'New Moon Books (NMB) has just released another top
ten publication. With the latest publication this makes NMB the hottest new publisher of
the year!';
GO
ALTER DATABASE pubs SET RECOVERY SIMPLE;
GO
```

See Also

Data Types

DECLARE @local_variable

DELETE

SELECT

SET

UPDATETEXT

Transact-SQL Syntax Conventions

The following table lists and describes conventions that are used in the syntax diagrams in the Transact-SQL Reference.

Convention	Used for
UPPERCASE	Transact-SQL keywords.
italic	User-supplied parameters of Transact-SQL syntax.
bold	Database names, table names, column names, index names, stored procedures, utilities, data type names, and text that must be typed exactly as shown.

Convention	Used for
underline	Indicates the default value applied when the clause that contains the underlined value is omitted from the statement.
| (vertical bar)	Separates syntax items enclosed in brackets or braces. You can use only one of the items.
[ ] (brackets)	Optional syntax items. Do not type the brackets.
{ } (braces)	Required syntax items. Do not type the braces.
[,...n]	Indicates the preceding item can be repeated n number of times. The occurrences are separated by commas.
[...n]	Indicates the preceding item can be repeated n number of times. The occurrences are separated by blanks.
;	Transact-SQL statement terminator.Although the semicolon is not required for most statements in this version of SQL Server, it will be required in a future version.
<label> ::=	The name for a block of syntax. This convention is used to group and label sections of lengthy syntax or a unit of syntax that can be used in more than one location within a statement. Each location in which the block of syntax can be used is indicated with the label enclosed in chevrons: <label>. A set is a collection of expressions, for example <grouping set>; and a list is a collection of sets, for example <composite element list>.

Multipart Names

Unless specified otherwise, all Transact-SQL references to the name of a database object can be a four-part name in the following form:

```
server_name.[database_name].[schema_name].object_name
| database_name.[schema_name].object_name
| schema_name.object_name
```

| object_name

server_name

Specifies a linked server name or remote server name.

database_name

Specifies the name of a SQL Server database when the object resides in a local instance of SQL Server. When the object is in a linked server, database_name specifies an OLE DB catalog.

schema_name

Specifies the name of the schema that contains the object if the object is in a SQL Server database. When the object is in a linked server, schema_name specifies an OLE DB sch ema name.

object_name

Refers to the name of the object.

When referencing a specific object, you do not always have to specify the server, database, and schema for the SQL Server Database Engine to identify the object. However, if the object cannot be found, an error is returned.

Note
To avoid name resolution errors, we recommend specifying the schema name whenever you specify a schema-scoped object.

To omit intermediate nodes, use periods to indicate these positions. The following table shows the valid formats of object names.

Object reference format	Description
server**.**database**.**schema**.**object	Four-part name.
server**.**database**..**object	Schema name is omitted.
server**..**schema**.**object	Database name is omitted.
server**...**object	Database and schema name are omitted.
database**.**schema**.**object	Server name is omitted.
database**..**object	Server and schema name are omitted.
schema**.**object	Server and database name are omitted.

Object reference format	Description
object	Server, database, and schema name are omitted.

Code Example Conventions

Unless stated otherwise, the examples provided in the Transact-SQL Reference were tested by using SQL Server Management Studio and its default settings for the following options:

- ANSI_NULLS
- ANSI_NULL_DFLT_ON
- ANSI_PADDING
- ANSI_WARNINGS
- CONCAT_NULL_YIELDS_NULL
- QUOTED_IDENTIFIER

Most code examples in the Transact-SQL Reference have been tested on servers that are running a case-sensitive sort order. The test servers were typically running the ANSI/ISO 1252 code page.

Many code examples prefix Unicode character string constants with the letter **N**. Without the **N** prefix, the string is converted to the default code page of the database. This default code page may not recognize certain characters.

See Also

Transact-SQL Reference (Database Engine)