T-SQL INSERT

Microsoft
Developer Network

Table Of Contents

Chapter 1

INSERT (Transact-SQL)

Updated: September 1, 2016

THIS TOPIC APPLIES TO: ✓ SQL Server (starting with 2008) ✓ Azure SQL Database ✓ Azure SQL Data Warehouse ✓ Parallel Data Warehouse

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

Transact-SQL Syntax Conventions

Syntax

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

[ 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 ] ) ]
```

```
-- 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 ]
    ) ]
}

[; ] <column_definition> ::=
 column_name <data_type>
    [ COLLATE collation_name ]
    [ NULL | NOT NULL ]

<data type> ::=
[ type_schema_name . ] type_name
    [ ( precision [ , scale ] | max ]
-- Syntax for Azure SQL Data Warehouse and Parallel Data Warehouse

INSERT INTO [ database_name . [ schema_name ] . | schema_name . ]
table_name
    [ ( column_name [ ,...n ] ) ]
    {
      VALUES ( { NULL | expression } )
      | SELECT <select_criteria>
    }
    [ OPTION ( <query_option> [ ,...n ] ) ]
[;]
```

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

Applies to: SQL Server 2008 through SQL Server 2016.

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

Applies to: SQL Server 2008 through SQL Server 2016.

Is the name of the database.

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

table_or view_name
Is the name of the table or view 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

Applies to: SQL Server 2008 through SQL Server 2016.

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 Hints (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 Use 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

Applies to: SQL Server 2008 through SQL Server 2016.

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

BULK

Applies to: SQL Server 2008 through SQL Server 2016.

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

Applies to: SQL Server 2008 through SQL Server 2016.

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

Applies to: SQL Server 2008 through SQL Server 2016.

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

Applies to: SQL Server 2008 through SQL Server 2016.

Specifies that empty columns should retain a null value during the binary data stream upload operation. For more information, see Keep Nulls or Use Default Values During Bulk Import (SQL Server).

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*

Applies to: SQL Server 2008 through SQL Server 2016.

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.

INSERT queries that use SELECT with ORDER BY to populate rows guarantees how identity values are computed but not the order in which the rows are inserted.

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 Import Bulk Data by Using BULK INSERT or OPENROWSET(BULK...) (SQL Server).

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 • 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 in the AdventureWorks2012 database. 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.

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

B. Inserting multiple rows of data

The following example uses the table value constructor to insert three rows into the Production.UnitMeasure table in the AdventureWorks2012 database 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.

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

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 in the AdventureWorks2012 database is UnitMeasureCode, Name, ModifiedDate; however, the columns are not listed in that order in *column_list*.

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

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.

```
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.

```
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.

```
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;
```

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`) in the AdventureWorks2012 database 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.

```
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;
```

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

The following example creates the `NewEmployee` table in the AdventureWorks2012 database. 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.

```
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` in the AdventureWorks2012 database. 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.

```
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 Modify Data Through a View.

```
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 in the AdventureWorks2012 database.

```
-- 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*.

Applies to: SQL Server 2008 through SQL Server 2016.

```
USE master;
GO
-- Create a link to the remote data source.
-- Specify a valid server name for @datasrc as 'server_name' or
'server_nameinstance_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 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 OPENQUERY rowset function. The linked server name created in the previous example is used in this example.

Applies to: SQL Server 2008 through SQL Server 2016.

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

Applies to: SQL Server 2008 through SQL Server 2016.

```
-- 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_nameinstance_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
```

D. Inserting into an external table created using PolyBase

Export data from SQL Server to Hadoop or Azure Storage. First, create an external table that points to the destination file or directory. Then, use INSERT INTO to export data from a local SQL Server table to an external data source. The INSERT INTO statement creates the destination file or directory if it does not exist and the results of the SELECT statement are exported to the specified location in the specified file format. For more information, see Get started with PolyBase.

Applies to: SQL Server 2016.

```
-- Create an external table.
CREATE EXTERNAL TABLE [dbo].[FastCustomers2009] (
        [FirstName] char(25) NOT NULL,
        [LastName] char(25) NOT NULL,
        [YearlyIncome] float NULL,
        [MaritalStatus] char(1) NOT NULL
)
WITH (
        LOCATION='/old_data/2009/customerdata.tbl',
        DATA_SOURCE = HadoopHDP2,
        FILE_FORMAT = TextFileFormat,
        REJECT_TYPE = VALUE,
        REJECT_VALUE = 0
);

-- Export data: Move old data to Hadoop while keeping it query-able via
external table.
INSERT INTO dbo.FastCustomer2009
SELECT T.* FROM Insured_Customers T1 JOIN CarSensor_Data T2
ON (T1.CustomerKey = T2.CustomerKey)
WHERE T2.YearMeasured = 2009 and T2.Speed > 40;
```

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 a a 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.

```
-- 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 Import Bulk Data by Using BULK INSERT or OPENROWSET(BULK...) (SQL Server).

Applies to: SQL Server 2008 through SQL Server 2016.

```
-- 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:SQLFilesDepartmentData.txt',
    FORMATFILE = 'C:SQLFilesBulkloadFormatFile.xml',
    ROWS_PER_BATCH = 15000)AS b ;
```

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.

Applies to: SQL Server, SQL Database.

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

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.

```
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;
```

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`.

```
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;
```

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 in the AdventureWorks2012 database. 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.

```
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;
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

D. Using a simple INSERT statement

The following example inserts one row in the `Production.UnitMeasure` 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.

```
-- Uses AdventureWorks

INSERT INTO DimCurrency
VALUES (500, N'C1', N'Currency1');
```

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

The following example uses a column list to explicitly specify the values inserted into each column. The column order in the `DimCurrency` table is `CurrencyKey`, `CurrencyAlternateKey`, `CurrencyName`. However, the columns are not listed in that order in *column_list*.

```
-- Uses AdventureWorks

INSERT INTO DimCurrency (CurrencyName, CurrencyKey, CurrencyAlternateKey)
VALUES (N'Currency1', 500, N'C1');
```

F. Inserting data using the SELECT option

The following example shows how to insert multiple rows of data using an INSERT statement with a SELECT option. The first `INSERT` statement uses a `SELECT` statement directly to retrieve data from the source table, and then to store the result set in the `EmployeeTitles` table.

```
CREATE TABLE EmployeeTitles
( EmployeeKey   INT NOT NULL,
    LastName     varchar(40) NOT NULL,
    Title      varchar(50) NOT NULL
);
INSERT INTO EmployeeTitles
    SELECT EmployeeKey, LastName, Title
    FROM ssawPDW.dbo.DimEmployee
    WHERE EndDate IS NULL;
```

G. Specifying a label with the INSERT statement

The following example shows the use of a label with an INSERT statement.

```
-- Uses AdventureWorks

INSERT INTO DimCurrency
VALUES (500, N'C1', N'Currency1')
OPTION ( LABEL = N'label1' );
```

H. Using a label and a query hint with the INSERT statement

This query shows the basic syntax for using a label and a query join hint with the INSERT statement. After the query is submitted to the Control node, SQL Server, running on the Compute nodes, will apply the hash join strategy when it generates the SQL Server query plan. For more information on join hints and how to use the OPTION clause, see OPTION (SQL Server PDW).

```
-- Uses AdventureWorks

INSERT INTO DimCustomer (CustomerKey, CustomerAlternateKey, FirstName,
MiddleName, LastName )
SELECT ProspectiveBuyerKey, ProspectAlternateKey, FirstName, MiddleName,
LastName
FROM ProspectiveBuyer p JOIN DimGeography g ON p.PostalCode = g.PostalCode
WHERE g.CountryRegionCode = 'FR'
OPTION ( LABEL = 'Add French Prospects', HASH JOIN)
;
```

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)
Use the inserted and deleted Tables