Contents

- COALESCE
- NULLIF
- Operators
  - Unary - Positive
  - Unary - Negative
  - Set - EXCEPT and INTERSECT
  - Set - UNION
  - Arithmetic
    - + (Addition)
    - += (Addition Assignment)
    - - (Subtraction)
    - -= (Subtraction Assignment)
    - * (Multiplication)
    - *= (Multiplication Assignment)
    - / (Division)
    - /= (Division Assignment)
    - % Modulus
    - %= Modulus Assignment
  - = (Assignment)
  - Bitwise
    - & (Bitwise AND)
    - &= (Bitwise AND Assignment)
    - | (Bitwise OR)
    - |= (Bitwise OR Assignment)
    - ^ (Bitwise Exclusive OR)
    - ^= (Bitwise Exclusive OR Assignment)
    - ~ (Bitwise NOT)
  - Comparison
    - = (Equals)
    - > (Greater Than)
    - < (Less Than)
    - >= (Greater Than or Equal To)

Language Elements (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

SQL Server supports the following language elements.

In This Section

-- (Comment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Indicates user-provided text. Comments can be inserted on a separate line, nested at the end of a Transact-SQL command line, or within a Transact-SQL statement. The server does not evaluate the comment.

Transact-SQL Syntax Conventions

Syntax

```
-- text_of_comment
```

Arguments

text_of_comment
Is the character string that contains the text of the comment.

Remarks

Use two hyphens (--) for single-line or nested comments. Comments inserted with -- are terminated by the newline character. There is no maximum length for comments. The following table lists the keyboard shortcuts that you can use to comment or uncomment text.

ACTION	STANDARD
Make the selected text a comment	CTRL+K, CTRL+C
Uncomment the selected text	CTRL+K, CTRL+U

For more information about keyboard shortcuts, see SQL Server Management Studio Keyboard Shortcuts.

For multiline comments, see Slash Star (Block Comment) (Transact-SQL).

Examples

The following example uses the -- commenting characters.

```
-- Choose the AdventureWorks2012 database.
USE AdventureWorks2012;
GO
-- Choose all columns and all rows from the Address table.
SELECT *
FROM Person.Address
ORDER BY PostalCode ASC; -- We do not have to specify ASC because
-- that is the default.
GO
```

See Also

Control-of-Flow Language (Transact-SQL)

Slash Star (Block Comment) (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Indicates user-provided text. The text between the /* and */ is not evaluated by the server.

Transact-SQL Syntax Conventions

Syntax

```
/*
text_of_comment
*/
```

Arguments

text_of_comment
Is the text of the comment. This is one or more character strings.

Remarks

Comments can be inserted on a separate line or within a Transact-SQL statement. Multiple-line comments must be indicated by /* and */. A stylistic convention often used for multiple-line comments is to begin the first line with /*, subsequent lines with **, and end with */.

There is no maximum length for comments.

Nested comments are supported. If the /* character pattern occurs anywhere within an existing comment, it is treated as the start of a nested comment and, therefore, requires a closing */ comment mark. If the closing comment mark does not exist, an error is generated.

For example, the following code generates an error.

```
DECLARE @comment AS varchar(20);
GO
/*
SELECT @comment = '/*';
*/
SELECT @@VERSION;
GO
```

To work around this error, make the following change.

```
DECLARE @comment AS varchar(20);
GO
/*
SELECT @comment = '/*';
*/ */
SELECT @@VERSION;
GO
```

Examples

The following example uses comments to explain what the section of the code is supposed to do.

```
USE AdventureWorks2012;
GO
/*
This section of the code joins the Person table with the Address table,
by using the Employee and BusinessEntityAddress tables in the middle to
get a list of all the employees in the AdventureWorks2012 database
and their contact information.
*/
SELECT p.FirstName, p.LastName, a.AddressLine1, a.AddressLine2, a.City, a.PostalCode
FROM Person.Person AS p
JOIN HumanResources.Employee AS e ON p.BusinessEntityID = e.BusinessEntityID
JOIN Person.BusinessEntityAddress AS ea ON e.BusinessEntityID = ea.BusinessEntityID
JOIN Person.Address AS a ON ea.AddressID = a.AddressID;
GO
```

See Also

-- (Comment) (Transact-SQL)
Control-of-Flow Language (Transact-SQL)

CREATE DIAGNOSTICS SESSION (Transact-SQL)

7/4/2018 • 3 minutes to read • Edit Online

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

Diagnostics sessions allow you to save detailed, user-defined diagnostic information on system or query performance.

Diagnostics sessions are typically used to debug performance for a specific query, or to monitor the behavior of a specific appliance component during appliance operation.

> **NOTE**
> You should be familiar with XML in order to use diagnostics sessions.

Syntax

```
Creating a new diagnostics session:
CREATE DIAGNOSTICS SESSION diagnostics_name AS N'{<session_xml>}';

<session_xml>::
<Session>
   [ <MaxItemCount>max_item_count_num</MaxItemCount> ]
   <Filter>
      { \<Event Name="event_name"/>
         [ <Where>\<filter_property_name Name="value" ComparisonType="comp_type"/></Where> ] [ ,...n ]
      } [ ,...n ]
   </Filter> ]
   <Capture>
      \<Property Name="property_name"/> [ ,...n ]
   </Capture>
<Session>

Retrieving results for a diagnostics session:
SELECT * FROM master.sysdiag.diagnostics_name ;

Removing results for a diagnostics session:
DROP DIAGNOSTICS SESSION diagnostics_name ;
```

Arguments

diagnostics_name
The name of the diagnostics session. Diagnostics session names can include characters a-z, A-Z, and 0-9 only. Also, diagnostics session names must start with a character. *diagnostics_name* is limited to 127 characters.

max_item_count_num
The number of events to be persisted in a view. For example, if 100 is specified, the 100 most recent events matching the filter criteria will be persisted to the diagnostics session. If fewer than 100 matching events are found, the diagnostics session will contain less than 100 events. *max_item_count_num* must be at least 100 and less than or equal to 100,000.

event_name
Defines the actual events to be collected in the diagnostics session. *event_name* is one of the events listed in

sys.pdw_diag_events where `sys.pdw_diag_events.is_enabled='True'`.

filter_property_name
The name of the property on which to restrict results. For example, if you want to limit based on session id, *filter_property_name* should be *SessionId*. See *property_name* below for a list of potential values for *filter_property_name*.

value
A value to evaluate against *filter_property_name*. The value type must match the property type. For example, if the property type is decimal, the type of *value* must be decimal.

comp_type
The comparison type. Potential values are:Equals, EqualsOrGreaterThan, EqualsOrLessThan, GreaterThan, LessThan, NotEquals, Contains, RegEx

property_name
A property related to the event. Property names can be part of the capture tag, or used as part of filtering criteria.

PROPERTY NAME	DESCRIPTION
UserName	A user (login) name.
SessionId	A session ID.
QueryId	A query ID.
CommandType	A command type.
CommandText	Text within a command processed.
OperationType	The operation type for the event.
Duration	The duration of the event.
SPID	The Service Process ID.

Remarks

Each user is allowed a maximum of 10 concurrent diagnostics sessions. See sys.pdw_diag_sessions for a list of your current sessions, and drop any unneeded sessions using `DROP DIAGNOSTICS SESSION`.

Diagnostics sessions will continue to collect metadata until dropped.

Permissions

Requires the **ALTER SERVER STATE** permission.

Locking

Takes a shared lock on the Diagnostic Sessions table.

Examples

A. Creating a diagnostics session

This example creates a diagnostics session to record metrics of the database engine performance. The example

creates a diagnostics session that listens for Engine Query running/end events and a blocking DMS event. What is returned is the command text, machine name, request id (query id) and the session that the event was created on.

```
CREATE DIAGNOSTICS SESSION MYDIAGSESSION AS N'
<Session>
   <MaxItemCount>100</MaxItemCount>
   <Filter>
      <Event Name="EngineInstrumentation:EngineQueryRunningEvent" />
      <Event Name="DmsCoreInstrumentation:DmsBlockingQueueEnqueueBeginEvent" />
      <Where>
         <SessionId Value="381" ComparisonType="NotEquals" />
      </Where>
   </Filter>
   <Capture>
      <Property Name="Query.CommandText" />
      <Property Name="MachineName" />
      <Property Name="Query.QueryId" />
      <Property Name="Alias" />
      <Property Name="Duration" />
      <Property Name="Session.SessionId" />
   </Capture>
</Session>';
```

After creation of the diagnostics session, run a query.

```
SELECT COUNT(EmployeeKey) FROM AdventureWorksPDW2012..FactSalesQuota;
```

Then view the diagnostics session results by selecting from the sysdiag schema.

```
SELECT * FROM master.sysdiag.MYDIAGSESSION;
```

Notice that the sysdiag schema contains a view that is named your diagnostics session name.

To see only the activity for your connection, add the `Session.SPID` property and add `WHERE [Session.SPID] = @@spid;` to the query.

When you are finished with the diagnostics session, drop it using the **DROP DIAGNOSTICS** command.

```
DROP DIAGNOSTICS SESSION MYDIAGSESSION;
```

B. Alternative diagnostic session

A second example with slightly different properties.

```
-- Determine the session_id of your current session
SELECT TOP 1 session_id();
-- Replace \<*session_number*> in the code below with the numbers in your session_id
CREATE DIAGNOSTICS SESSION PdwOptimizationDiagnostics AS N'
<Session>
   <MaxItemCount>100</MaxItemCount>
   <Filter>
      <Event Name="EngineInstrumentation:MemoGenerationBeginEvent" />
      <Event Name="EngineInstrumentation:MemoGenerationEndEvent" />
      <Event Name="DSQLInstrumentation:OptimizationBeginEvent" />
      <Event Name="DSQLInstrumentation:OptimizationEndEvent" />
      <Event Name="DSQLInstrumentation:BuildRelOpContextTreeBeginEvent" />
      <Event Name="DSQLInstrumentation:PostPlanGenModifiersEndEvent" />
      <Where>
         <SessionId Value="\<*session_number*>" ComparisonType="Equals" />
      </Where>
   </Filter>
   <Capture>
      <Property Name="Session.SessionId" />
      <Property Name="Query.QueryId" />
      <Property Name="Query.CommandText" />
      <Property Name="Name" />
      <Property Name="DateTimePublished" />
      <Property Name="DateTimePublished.Ticks" />
  </Capture>
</Session>';
```

Run a query, such as:

```
USE ssawPDW;
GO
SELECT * FROM dbo.FactFinance;
```

The following query returns the authorization timing:

```
SELECT *
FROM master.sysdiag.PdwOptimizationDiagnostics
ORDER BY DateTimePublished;
```

When you are finished with the diagnostics session, drop it using the **DROP DIAGNOSTICS** command.

```
DROP DIAGNOSTICS SESSION PdwOptimizationDiagnostics;
```

NULL and UNKNOWN (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

NULL indicates that the value is unknown. A null value is different from an empty or zero value. No two null values are equal. Comparisons between two null values, or between a null value and any other value, return unknown because the value of each NULL is unknown.

Null values generally indicate data that is unknown, not applicable, or to be added later. For example, a customer's middle initial may not be known at the time the customer places an order.

Note the following about null values:

- To test for null values in a query, use IS NULL or IS NOT NULL in the WHERE clause.
- Null values can be inserted into a column by explicitly stating NULL in an INSERT or UPDATE statement or by leaving a column out of an INSERT statement.
- Null values cannot be used as information that is required to distinguish one row in a table from another row in a table, such as primary keys, or for information used to distribute rows, such as distribution keys.

  When null values are present in data, logical and comparison operators can potentially return a third result of UNKNOWN instead of just TRUE or FALSE. This need for three-valued logic is a source of many application errors. Logical operators in a boolean expression that includes UNKNOWNs will return UNKNOWN unless the result of the operator does not depend on the UNKNOWN expression. These tables provide examples of this behavior.

  The following table shows the results of applying an AND operator to two Boolean expressions where one expression returns UNKNOWN.

EXPRESSION 1	EXPRESSION 2	RESULT
TRUE	UNKNOWN	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN
FALSE	UNKNOWN	FALSE

The following table shows the results of applying an OR operator to two Boolean expressions where one expression returns UNKNOWN.

EXPRESSION 1	EXPRESSION 2	RESULT
TRUE	UNKNOWN	TRUE
UNKNOWN	UNKNOWN	UNKNOWN
FALSE	UNKNOWN	UNKNOWN

See Also

AND (Transact-SQL)
OR (Transact-SQL)
NOT (Transact-SQL)
IS NULL (Transact-SQL)

USE (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Changes the database context to the specified database or database snapshot in SQL Server.

Transact-SQL Syntax Conventions

Syntax

```
USE { database_name }
[;]
```

Arguments

database_name
Is the name of the database or database snapshot to which the user context is switched. Database and database snapshot names must comply with the rules for identifiers.

In Azure SQL Database, the database parameter can only refer to the current database. If a database other than the current database is provided, the `USE` statement does not switch between databases, and error code 40508 is returned. To change databases, you must directly connect to the database. The USE statement is marked as not applicable to SQL Database at the top of this page, because even though you can have the `USE` statement in a batch, it doesn't do anything.

Remarks

When a SQL Server login connects to SQL Server, the login is automatically connected to its default database and acquires the security context of a database user. If no database user has been created for the SQL Server login, the login connects as guest. If the database user does not have CONNECT permission on the database, the USE statement will fail. If no default database has been assigned to the login, its default database will be set to master.

USE is executed at both compile and execution time and takes effect immediately. Therefore, statements that appear in a batch after the USE statement are executed in the specified database.

Permissions

Requires CONNECT permission on the target database.

Examples

The following example changes the database context to the `AdventureWorks2012` database.

```
USE AdventureWorks2012;
GO
```

See Also

CREATE LOGIN (Transact-SQL)
CREATE USER (Transact-SQL)
Principals (Database Engine)
CREATE DATABASE (SQL Server Transact-SQL)
DROP DATABASE (Transact-SQL)
EXECUTE (Transact-SQL)

Backslash (Line Continuation) (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

`\` breaks a long string constant, character or binary, into two or more lines for readability.

Transact-SQL Syntax Conventions

Syntax

```
<first section of string> \
<continued section of string>
```

Arguments

<first section of string>
Is the start of a string.

<continued section of string>
Is the continuation of a string.

Remarks

This command returns the first and continued sections of the string as one string, without the backslash.

Examples

A. Splitting a character string

The following example uses a backslash and a carriage return to split a character string into two lines.

```
SELECT 'abc\
def' AS [ColumnResult];
```

Here is the result set.

```
ColumnResult
------------
abcdef
```

B. Splitting a binary string

The following example uses a backslash and a carriage return to split a binary string into two lines.

```
SELECT 0xabc\
def AS [ColumnResult];
```

Here is the result set.

```
ColumnResult
------------
0xABCDEF
```

See Also

Data Types (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
(Division) (Transact-SQL)
(Division Assignment) (Transact-SQL)
Compound Operators (Transact-SQL)

SQL Server Utilities Statements - GO

6/20/2018 • 2 minutes to read • Edit Online

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

SQL Server provides commands that are not Transact-SQL statements, but are recognized by the **sqlcmd** and **osql** utilities and SQL Server Management Studio Code Editor. These commands can be used to facilitate the readability and execution of batches and scripts.

GO signals the end of a batch of Transact-SQL statements to the SQL Server utilities.

Transact-SQL Syntax Conventions

Syntax

```
GO [count]
```

Arguments

count
Is a positive integer. The batch preceding GO will execute the specified number of times.

Remarks

GO is not a Transact-SQL statement; it is a command recognized by the **sqlcmd** and **osql** utilities and SQL Server Management Studio Code editor.

SQL Server utilities interpret GO as a signal that they should send the current batch of Transact-SQL statements to an instance of SQL Server. The current batch of statements is composed of all statements entered since the last GO, or since the start of the ad hoc session or script if this is the first GO.

A Transact-SQL statement cannot occupy the same line as a GO command. However, the line can contain comments.

Users must follow the rules for batches. For example, any execution of a stored procedure after the first statement in a batch must include the EXECUTE keyword. The scope of local (user-defined) variables is limited to a batch, and cannot be referenced after a GO command.

```
USE AdventureWorks2012;
GO
DECLARE @MyMsg VARCHAR(50)
SELECT @MyMsg = 'Hello, World.'
GO -- @MyMsg is not valid after this GO ends the batch.

-- Yields an error because @MyMsg not declared in this batch.
PRINT @MyMsg
GO

SELECT @@VERSION;
-- Yields an error: Must be EXEC sp_who if not first statement in
-- batch.
sp_who
GO
```

SQL Server applications can send multiple Transact-SQL statements to an instance of SQL Server for execution as a batch. The statements in the batch are then compiled into a single execution plan. Programmers executing ad hoc statements in the SQL Server utilities, or building scripts of Transact-SQL statements to run through the SQL Server utilities, use GO to signal the end of a batch.

Applications based on the ODBC or OLE DB APIs receive a syntax error if they try to execute a GO command. The SQL Server utilities never send a GO command to the server.

Do not use a semicolon as a statement terminator after GO.

Permissions

GO is a utility command that requires no permissions. It can be executed by any user.

```
-- Yields an error because ; is not permitted after GO
SELECT @@VERSION;
GO;
```

Examples

The following example creates two batches. The first batch contains only a `USE AdventureWorks2012` statement to set the database context. The remaining statements use a local variable. Therefore, all local variable declarations must be grouped in a single batch. This is done by not having a `GO` command until after the last statement that references the variable.

```
USE AdventureWorks2012;
GO
DECLARE @NmbrPeople int
SELECT @NmbrPeople = COUNT(*)
FROM Person.Person;
PRINT 'The number of people as of ' +
      CAST(GETDATE() AS char(20)) + ' is ' +
      CAST(@NmbrPeople AS char (10));
GO
```

The following example executes the statements in the batch twice.

```
SELECT DB_NAME();
SELECT USER_NAME();
GO 2
```

Control-of-Flow

6/20/2018 • 2 minutes to read • Edit Online

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The Transact-SQL control-of-flow language keywords are:

BEGIN...END	RETURN
BREAK	THROW
CONTINUE	TRY...CATCH
GOTO *label*	WAITFOR
IF...ELSE	WHILE

See Also

CASE (Transact-SQL)
Slash Star (Block Comment) (Transact-SQL)
-- (Comment) (Transact-SQL)
DECLARE @local_variable (Transact-SQL)
EXECUTE (Transact-SQL)
PRINT (Transact-SQL)
RAISERROR (Transact-SQL)

BEGIN...END (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Encloses a series of Transact-SQL statements so that a group of Transact-SQL statements can be executed. BEGIN and END are control-of-flow language keywords.

Transact-SQL Syntax Conventions

Syntax

```
BEGIN
    { sql_statement | statement_block }
END
```

Arguments

{ *sql_statement* | *statement_block* }
Is any valid Transact-SQL statement or statement grouping as defined by using a statement block.

Remarks

BEGIN...END blocks can be nested.

Although all Transact-SQL statements are valid within a BEGIN...END block, certain Transact-SQL statements should not be grouped together within the same batch, or statement block.

Examples

In the following example, `BEGIN` and `END` define a series of Transact-SQL statements that execute together. If the `BEGIN...END` block were not included, both `ROLLBACK TRANSACTION` statements would execute and both `PRINT` messages would be returned.

```
USE AdventureWorks2012;
GO
BEGIN TRANSACTION;
GO
IF @@TRANCOUNT = 0
BEGIN
    SELECT FirstName, MiddleName
    FROM Person.Person WHERE LastName = 'Adams';
    ROLLBACK TRANSACTION;
    PRINT N'Rolling back the transaction two times would cause an error.';
END;
ROLLBACK TRANSACTION;
PRINT N'Rolled back the transaction.';
GO
/*
Rolled back the transaction.
*/
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

In the following example, `BEGIN` and `END` define a series of SQL statements that run together. If the `BEGIN...END` block are not included, the following example will be in a continuous loop.

```
-- Uses AdventureWorks

DECLARE @Iteration Integer = 0
WHILE @Iteration <10
BEGIN
    SELECT FirstName, MiddleName
    FROM dbo.DimCustomer WHERE LastName = 'Adams';
SET @Iteration += 1
END;
```

See Also

ALTER TRIGGER (Transact-SQL)
Control-of-Flow Language (Transact-SQL)
CREATE TRIGGER (Transact-SQL)
END (BEGIN...END) (Transact-SQL)

BREAK (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Exits the innermost loop in a WHILE statement or an IF...ELSE statement inside a WHILE loop. Any statements appearing after the END keyword, marking the end of the loop, are executed. BREAK is frequently, but not always, started by an IF test.

Examples

```
-- Uses AdventureWorks

WHILE ((SELECT AVG(ListPrice) FROM dbo.DimProduct) < $300)
BEGIN
    UPDATE DimProduct
        SET ListPrice = ListPrice * 2;
     IF ((SELECT MAX(ListPrice) FROM dbo.DimProduct) > $500)
         BREAK;
END
```

See Also

Control-of-Flow Language (Transact-SQL)
WHILE (Transact-SQL)
IF...ELSE (Transact-SQL)

CONTINUE (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Restarts a WHILE loop. Any statements after the CONTINUE keyword are ignored. CONTINUE is frequently, but not always, opened by an IF test. For more information, see WHILE (Transact-SQL) and Control-of-Flow Language (Transact-SQL).

ELSE (IF...ELSE) (Transact-SQL)

8/27/2018 • 3 minutes to read • Edit Online

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

Imposes conditions on the execution of a Transact-SQL statement. The Transact-SQL statement (*sql_statement*) following the *Boolean_expression*is executed if the *Boolean_expression* evaluates to TRUE. The optional ELSE keyword is an alternate Transact-SQL statement that is executed when *Boolean_expression* evaluates to FALSE or NULL.

Transact-SQL Syntax Conventions

Syntax

```
IF Boolean_expression
     { sql_statement | statement_block }
[ ELSE
     { sql_statement | statement_block } ]
```

Arguments

Boolean_expression
Is an expression that returns TRUE or FALSE. If the *Boolean_expression* contains a SELECT statement, the SELECT statement must be enclosed in parentheses.

{ *sql_statement* | *statement_block* }
Is any valid Transact-SQL statement or statement grouping as defined with a statement block. To define a statement block (batch), use the control-of-flow language keywords BEGIN and END. Although all Transact-SQL statements are valid within a BEGIN...END block, certain Transact-SQL statements should not be grouped together within the same batch (statement block).

Result Types

Boolean

Examples

A. Using a simple Boolean expression

The following example has a simple Boolean expression (`1=1`) that is true and, therefore, prints the first statement.

```
IF 1 = 1 PRINT 'Boolean_expression is true.'
ELSE PRINT 'Boolean_expression is false.' ;
```

The following example has a simple Boolean expression (`1=2`) that is false, and therefore prints the second statement.

```
IF 1 = 2 PRINT 'Boolean_expression is true.'
ELSE PRINT 'Boolean_expression is false.' ;
GO
```

B. Using a query as part of a Boolean expression

The following example executes a query as part of the Boolean expression. Because there are 10 bikes in the `Product` table that meet the `WHERE` clause, the first print statement will execute. Change `> 5` to `> 15` to see how the second part of the statement could execute.

```
USE AdventureWorks2012;
GO
IF
(SELECT COUNT(*) FROM Production.Product WHERE Name LIKE 'Touring-3000%' ) > 5
PRINT 'There are more than 5 Touring-3000 bicycles.'
ELSE PRINT 'There are 5 or less Touring-3000 bicycles.' ;
GO
```

C. Using a statement block

The following example executes a query as part of the Boolean expression and then executes slightly different statement blocks based on the result of the Boolean expression. Each statement block starts with `BEGIN` and completes with `END`.

```
USE AdventureWorks2012;
GO
DECLARE @AvgWeight decimal(8,2), @BikeCount int
IF
(SELECT COUNT(*) FROM Production.Product WHERE Name LIKE 'Touring-3000%' ) > 5
BEGIN
   SET @BikeCount =
        (SELECT COUNT(*)
         FROM Production.Product
         WHERE Name LIKE 'Touring-3000%');
   SET @AvgWeight =
        (SELECT AVG(Weight)
         FROM Production.Product
         WHERE Name LIKE 'Touring-3000%');
   PRINT 'There are ' + CAST(@BikeCount AS varchar(3)) + ' Touring-3000 bikes.'
   PRINT 'The average weight of the top 5 Touring-3000 bikes is ' + CAST(@AvgWeight AS varchar(8)) + '.';
END
ELSE
BEGIN
SET @AvgWeight =
        (SELECT AVG(Weight)
         FROM Production.Product
         WHERE Name LIKE 'Touring-3000%' );
   PRINT 'Average weight of the Touring-3000 bikes is ' + CAST(@AvgWeight AS varchar(8)) + '.' ;
END ;
GO
```

D. Using nested IF...ELSE statements

The following example shows how an IF ... ELSE statement can be nested inside another. Set the `@Number` variable to `5`, `50`, and `500` to test each statement.

```
DECLARE @Number int;
SET @Number = 50;
IF @Number > 100
   PRINT 'The number is large.';
ELSE
   BEGIN
      IF @Number < 10
      PRINT 'The number is small.';
   ELSE
      PRINT 'The number is medium.';
   END ;
GO
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

E: Using a query as part of a Boolean expression

The following example uses `IF...ELSE` to determine which of two responses to show the user, based on the weight of an item in the `DimProduct` table.

```
-- Uses AdventureWorks

DECLARE @maxWeight float, @productKey integer
SET @maxWeight = 100.00
SET @productKey = 424
IF @maxWeight <= (SELECT Weight from DimProduct WHERE ProductKey=@productKey)
    (SELECT @productKey, EnglishDescription, Weight, 'This product is too heavy to ship and is only available
for pickup.' FROM DimProduct WHERE ProductKey=@productKey)
ELSE
    (SELECT @productKey, EnglishDescription, Weight, 'This product is available for shipping or pickup.' FROM
DimProduct WHERE ProductKey=@productKey)
```

See Also

ALTER TRIGGER (Transact-SQL)
Control-of-Flow Language (Transact-SQL)
CREATE TRIGGER (Transact-SQL)
IF...ELSE (Transact-SQL)

END (BEGIN...END) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Encloses a series of Transact-SQL statements that will execute as a group. BEGIN...END blocks can be nested.

Transact-SQL Syntax Conventions

Syntax

```
BEGIN
     { sql_statement | statement_block }
END
```

Arguments

{ *sql_statement*| *statement_block*}
Is any valid Transact-SQL statement or statement grouping as defined with a statement block. To define a statement block (batch), use the control-of-flow language keywords BEGIN and END. Although all Transact-SQL statements are valid within a BEGIN...END block, certain Transact-SQL statements should not be grouped together within the same batch (statement block).

Result Types

Boolean

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

In the following example, `BEGIN` and `END` define a series of SQL statements that run together. If the `BEGIN...END` block are not included, the following example will be in a continuous loop.

```
-- Uses AdventureWorks

DECLARE @Iteration Integer = 0
WHILE @Iteration <10
BEGIN
    SELECT FirstName, MiddleName
    FROM dbo.DimCustomer WHERE LastName = 'Adams';
SET @Iteration += 1
END;
```

See Also

ALTER TRIGGER (Transact-SQL)
BEGIN...END (Transact-SQL)
Control-of-Flow Language (Transact-SQL)
CREATE TRIGGER (Transact-SQL)
ELSE (IF...ELSE) (Transact-SQL)
IF...ELSE (Transact-SQL)

GOTO (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Alters the flow of execution to a label. The Transact-SQL statement or statements that follow GOTO are skipped and processing continues at the label. GOTO statements and labels can be used anywhere within a procedure, batch, or statement block. GOTO statements can be nested.

Transact-SQL Syntax Conventions

Syntax

```
Define the label:
label:
Alter the execution:
GOTO label
```

Arguments

label

Is the point after which processing starts if a GOTO is targeted to that label. Labels must follow the rules for identifiers. A label can be used as a commenting method whether GOTO is used.

Remarks

GOTO can exist within conditional control-of-flow statements, statement blocks, or procedures, but it cannot go to a label outside the batch. GOTO branching can go to a label defined before or after GOTO.

Permissions

GOTO permissions default to any valid user.

Examples

The following example shows how to use `GOTO` as a branch mechanism.

```
DECLARE @Counter int;
SET @Counter = 1;
WHILE @Counter < 10
BEGIN
    SELECT @Counter
    SET @Counter = @Counter + 1
    IF @Counter = 4 GOTO Branch_One --Jumps to the first branch.
    IF @Counter = 5 GOTO Branch_Two  --This will never execute.
END
Branch_One:
    SELECT 'Jumping To Branch One.'
    GOTO Branch_Three; --This will prevent Branch_Two from executing.
Branch_Two:
    SELECT 'Jumping To Branch Two.'
Branch_Three:
    SELECT 'Jumping To Branch Three.';
```

See Also

Control-of-Flow Language (Transact-SQL)
BEGIN...END (Transact-SQL)
BREAK (Transact-SQL)
CONTINUE (Transact-SQL)
IF...ELSE (Transact-SQL)
WAITFOR (Transact-SQL)
WHILE (Transact-SQL)

IF...ELSE (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Imposes conditions on the execution of a Transact-SQL statement. The Transact-SQL statement that follows an IF keyword and its condition is executed if the condition is satisfied: the Boolean expression returns TRUE. The optional ELSE keyword introduces another Transact-SQL statement that is executed when the IF condition is not satisfied: the Boolean expression returns FALSE.

 Transact-SQL Syntax Conventions

Syntax

```
IF Boolean_expression
     { sql_statement | statement_block }
[ ELSE
     { sql_statement | statement_block } ]
```

Arguments

Boolean_expression
Is an expression that returns TRUE or FALSE. If the Boolean expression contains a SELECT statement, the SELECT statement must be enclosed in parentheses.

{ *sql_statement*| *statement_block* }
Is any Transact-SQL statement or statement grouping as defined by using a statement block. Unless a statement block is used, the IF or ELSE condition can affect the performance of only one Transact-SQL statement.

To define a statement block, use the control-of-flow keywords BEGIN and END.

Remarks

An IF...ELSE construct can be used in batches, in stored procedures, and in ad hoc queries. When this construct is used in a stored procedure, it is frequently used to test for the existence of some parameter.

IF tests can be nested after another IF or following an ELSE. The limit to the number of nested levels depends on available memory.

Example

```
IF DATENAME(weekday, GETDATE()) IN (N'Saturday', N'Sunday')
       SELECT 'Weekend';
ELSE
       SELECT 'Weekday';
```

For more examples, see ELSE (IF...ELSE) (Transact-SQL).

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

The following example uses `IF…ELSE` to determine which of two responses to show the user, based on the weight of an item in the `DimProduct` table.

```
-- Uses AdventureWorksDW

DECLARE @maxWeight float, @productKey integer
SET @maxWeight = 100.00
SET @productKey = 424
IF @maxWeight <= (SELECT Weight from DimProduct
                  WHERE ProductKey = @productKey)
    (SELECT @productKey AS ProductKey, EnglishDescription, Weight,
    'This product is too heavy to ship and is only available for pickup.'
        AS ShippingStatus
    FROM DimProduct WHERE ProductKey = @productKey);
ELSE
    (SELECT @productKey AS ProductKey, EnglishDescription, Weight,
    'This product is available for shipping or pickup.'
        AS ShippingStatus
    FROM DimProduct WHERE ProductKey = @productKey);
```

See Also

BEGIN...END (Transact-SQL)
END (BEGIN...END) (Transact-SQL)
SELECT (Transact-SQL)
WHILE (Transact-SQL)
CASE (Transact-SQL)
Control-of-Flow Language (Transact-SQL) ELSE (IF...ELSE) (Transact-SQL)

RETURN (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Exits unconditionally from a query or procedure. RETURN is immediate and complete and can be used at any point to exit from a procedure, batch, or statement block. Statements that follow RETURN are not executed.

Transact-SQL Syntax Conventions

Syntax

```
RETURN [ integer_expression ]
```

Arguments

integer_expression
Is the integer value that is returned. Stored procedures can return an integer value to a calling procedure or an application.

Return Types

Optionally returns **int**.

> **NOTE**
> Unless documented otherwise, all system stored procedures return a value of 0. This indicates success and a nonzero value indicates failure.

Remarks

When used with a stored procedure, RETURN cannot return a null value. If a procedure tries to return a null value (for example, using RETURN @status when @status is NULL), a warning message is generated and a value of 0 is returned.

The return status value can be included in subsequent Transact-SQL statements in the batch or procedure that executed the current procedure, but it must be entered in the following form:
`EXECUTE @return_status = <procedure_name>`.

Examples

A. Returning from a procedure

The following example shows if no user name is specified as a parameter when `findjobs` is executed, `RETURN` causes the procedure to exit after a message has been sent to the user's screen. If a user name is specified, the names of all objects created by this user in the current database are retrieved from the appropriate system tables.

```
CREATE PROCEDURE findjobs @nm sysname = NULL
AS
IF @nm IS NULL
    BEGIN
        PRINT 'You must give a user name'
        RETURN
    END
ELSE
    BEGIN
        SELECT o.name, o.id, o.uid
        FROM sysobjects o INNER JOIN master..syslogins l
            ON o.uid = l.sid
        WHERE l.name = @nm
    END;
```

B. Returning status codes

The following example checks the state for the ID of a specified contact. If the state is Washington (`WA`), a status of `1` is returned. Otherwise, `2` is returned for any other condition (a value other than `WA` for `StateProvince` or `ContactID` that did not match a row).

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE checkstate @param varchar(11)
AS
IF (SELECT StateProvince FROM Person.vAdditionalContactInfo WHERE ContactID = @param) = 'WA'
    RETURN 1
ELSE
    RETURN 2;
GO
```

The following examples show the return status from executing `checkstate` . The first shows a contact in Washington; the second, contact not in Washington; and the third, a contact that is not valid. The `@return_status` local variable must be declared before it can be used.

```
DECLARE @return_status int;
EXEC @return_status = checkstate '2';
SELECT 'Return Status' = @return_status;
GO
```

Here is the result set.

```
Return Status

-------------

1
```

Execute the query again, specifying a different contact number.

```
DECLARE @return_status int;
EXEC @return_status = checkstate '6';
SELECT 'Return Status' = @return_status;
GO
```

Here is the result set.

```
Return Status
-------------

2
```

Execute the query again, specifying another contact number.

```
DECLARE @return_status int
EXEC @return_status = checkstate '12345678901';
SELECT 'Return Status' = @return_status;
GO
```

Here is the result set.

```
Return Status
-------------

2
```

See Also

ALTER PROCEDURE (Transact-SQL)
CREATE PROCEDURE (Transact-SQL)
DECLARE @local_variable (Transact-SQL)
EXECUTE (Transact-SQL)
SET @local_variable (Transact-SQL)
THROW (Transact-SQL)

THROW (Transact-SQL)

8/27/2018 • 3 minutes to read • Edit Online

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Raises an exception and transfers execution to a CATCH block of a TRY...CATCH construct in SQL Server 2017.

Transact-SQL Syntax Conventions

Syntax

```
THROW [ { error_number | @local_variable },
        { message | @local_variable },
        { state | @local_variable } ]
[ ; ]
```

Arguments

error_number
Is a constant or variable that represents the exception. *error_number* is **int** and must be greater than or equal to 50000 and less than or equal to 2147483647.

message
Is an string or variable that describes the exception. *message* is **nvarchar(2048)**.

state
Is a constant or variable between 0 and 255 that indicates the state to associate with the message. *state* is **tinyint**.

Remarks

The statement before the THROW statement must be followed by the semicolon (;) statement terminator.

If a TRY...CATCH construct is not available, the statement batch is terminated. The line number and procedure where the exception is raised are set. The severity is set to 16.

If the THROW statement is specified without parameters, it must appear inside a CATCH block. This causes the caught exception to be raised. Any error that occurs in a THROW statement causes the statement batch to be terminated.

% is a reserved character in the message text of a THROW statement and must be escaped. Double the % character to return % as part of the message text, for example 'The increase exceeded 15%% of the original value.'

Differences Between RAISERROR and THROW

The following table lists differences between the RAISERROR and THROW statements.

RAISERROR STATEMENT	THROW STATEMENT
If a *msg_id* is passed to RAISERROR, the ID must be defined in sys.messages.	The *error_number* parameter does not have to be defined in sys.messages.

RAISERROR STATEMENT	THROW STATEMENT
The *msg_str* parameter can contain **printf** formatting styles.	The *message* parameter does not accept **printf** style formatting.
The *severity* parameter specifies the severity of the exception.	There is no *severity* parameter. The exception severity is always set to 16.

Examples

A. Using THROW to raise an exception

The following example shows how to use the `THROW` statement to raise an exception.

```
THROW 51000, 'The record does not exist.', 1;
```

Here is the result set.

```
Msg 51000, Level 16, State 1, Line 1

The record does not exist.
```

B. Using THROW to raise an exception again

The following example shows how use the `THROW` statement to raise the last thrown exception again.

```
USE tempdb;
GO
CREATE TABLE dbo.TestRethrow
(    ID INT PRIMARY KEY
);
BEGIN TRY
    INSERT dbo.TestRethrow(ID) VALUES(1);
--  Force error 2627, Violation of PRIMARY KEY constraint to be raised.
    INSERT dbo.TestRethrow(ID) VALUES(1);
END TRY
BEGIN CATCH

    PRINT 'In catch block.';
    THROW;
END CATCH;
```

Here is the result set.

```
PRINT 'In catch block.';
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__TestReth__3214EC272E3BD7D3'. Cannot insert duplicate key in object
'dbo.TestRethrow'.
The statement has been terminated.
```

C. Using FORMATMESSAGE with THROW

The following example shows how to use the `FORMATMESSAGE` function with `THROW` to throw a customized error message. The example first creates a user-defined error message by using `sp_addmessage`. Because the THROW statement does not allow for substitution parameters in the *message* parameter in the way that RAISERROR does, the FORMATMESSAGE function is used to pass the three parameter values expected by error message 60000.

```
EXEC sys.sp_addmessage
     @msgnum   = 60000
,@severity = 16
,@msgtext  = N'This is a test message with one numeric parameter (%d), one string parameter (%s), and another
string parameter (%s).'
    ,@lang = 'us_english';
GO

DECLARE @msg NVARCHAR(2048) = FORMATMESSAGE(60000, 500, N'First string', N'second string');

THROW 60000, @msg, 1;
```

Here is the result set.

```
Msg 60000, Level 16, State 1, Line 2
This is a test message with one numeric parameter (500), one string parameter (First string), and another
string parameter (second string).
```

See Also

FORMATMESSAGE (Transact-SQL)
Database Engine Error Severities
ERROR_LINE (Transact-SQL)
ERROR_MESSAGE (Transact-SQL)
ERROR_NUMBER (Transact-SQL)
ERROR_PROCEDURE (Transact-SQL)
ERROR_SEVERITY (Transact-SQL)
ERROR_STATE (Transact-SQL)
RAISERROR (Transact-SQL)
@@ERROR (Transact-SQL)
GOTO (Transact-SQL)
BEGIN...END (Transact-SQL)
XACT_STATE (Transact-SQL)
SET XACT_ABORT (Transact-SQL)

TRY...CATCH (Transact-SQL)

8/27/2018 • 9 minutes to read • Edit Online

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

Implements error handling for Transact-SQL that is similar to the exception handling in the Microsoft Visual C# and Microsoft Visual C++ languages. A group of Transact-SQL statements can be enclosed in a TRY block. If an error occurs in the TRY block, control is passed to another group of statements that is enclosed in a CATCH block.

Transact-SQL Syntax Conventions

Syntax

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
    [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

Arguments

sql_statement
Is any Transact-SQL statement.

statement_block
Any group of Transact-SQL statements in a batch or enclosed in a BEGIN...END block.

Remarks

A TRY...CATCH construct catches all execution errors that have a severity higher than 10 that do not close the database connection.

A TRY block must be immediately followed by an associated CATCH block. Including any other statements between the END TRY and BEGIN CATCH statements generates a syntax error.

A TRY...CATCH construct cannot span multiple batches. A TRY...CATCH construct cannot span multiple blocks of Transact-SQL statements. For example, a TRY...CATCH construct cannot span two BEGIN...END blocks of Transact-SQL statements and cannot span an IF...ELSE construct.

If there are no errors in the code that is enclosed in a TRY block, when the last statement in the TRY block has finished running, control passes to the statement immediately after the associated END CATCH statement. If there is an error in the code that is enclosed in a TRY block, control passes to the first statement in the associated CATCH block. If the END CATCH statement is the last statement in a stored procedure or trigger, control is passed back to the statement that called the stored procedure or fired the trigger.

When the code in the CATCH block finishes, control passes to the statement immediately after the END CATCH statement. Errors trapped by a CATCH block are not returned to the calling application. If any part of the error information must be returned to the application, the code in the CATCH block must do so by using mechanisms such as SELECT result sets or the RAISERROR and PRINT statements.

TRY...CATCH constructs can be nested. Either a TRY block or a CATCH block can contain nested TRY...CATCH constructs. For example, a CATCH block can contain an embedded TRY...CATCH construct to handle errors encountered by the CATCH code.

Errors encountered in a CATCH block are treated like errors generated anywhere else. If the CATCH block contains a nested TRY...CATCH construct, any error in the nested TRY block will pass control to the nested CATCH block. If there is no nested TRY...CATCH construct, the error is passed back to the caller.

TRY...CATCH constructs catch unhandled errors from stored procedures or triggers executed by the code in the TRY block. Alternatively, the stored procedures or triggers can contain their own TRY...CATCH constructs to handle errors generated by their code. For example, when a TRY block executes a stored procedure and an error occurs in the stored procedure, the error can be handled in the following ways:

- If the stored procedure does not contain its own TRY...CATCH construct, the error returns control to the CATCH block associated with the TRY block that contains the EXECUTE statement.
- If the stored procedure contains a TRY...CATCH construct, the error transfers control to the CATCH block in the stored procedure. When the CATCH block code finishes, control is passed back to the statement immediately after the EXECUTE statement that called the stored procedure.

  GOTO statements cannot be used to enter a TRY or CATCH block. GOTO statements can be used to jump to a label inside the same TRY or CATCH block or to leave a TRY or CATCH block.

  The TRY...CATCH construct cannot be used in a user-defined function.

Retrieving Error Information

In the scope of a CATCH block, the following system functions can be used to obtain information about the error that caused the CATCH block to be executed:

- ERROR_NUMBER() returns the number of the error.
- ERROR_SEVERITY() returns the severity.
- ERROR_STATE() returns the error state number.
- ERROR_PROCEDURE() returns the name of the stored procedure or trigger where the error occurred.
- ERROR_LINE() returns the line number inside the routine that caused the error.
- ERROR_MESSAGE() returns the complete text of the error message. The text includes the values supplied for any substitutable parameters, such as lengths, object names, or times.

  These functions return NULL if they are called outside the scope of the CATCH block. Error information can be retrieved by using these functions from anywhere within the scope of the CATCH block. For example, the following script shows a stored procedure that contains error-handling functions. In the `CATCH` block of a `TRY...CATCH` construct, the stored procedure is called and information about the error is returned.

```
-- Verify that the stored procedure does not already exist.
IF OBJECT_ID ( 'usp_GetErrorInfo', 'P' ) IS NOT NULL
    DROP PROCEDURE usp_GetErrorInfo;
GO

-- Create procedure to retrieve error information.
CREATE PROCEDURE usp_GetErrorInfo
AS
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;
GO

BEGIN TRY
    -- Generate divide-by-zero error.
    SELECT 1/0;
END TRY
BEGIN CATCH
    -- Execute error retrieval routine.
    EXECUTE usp_GetErrorInfo;
END CATCH;
```

The ERROR_* functions also work in a `CATCH` block inside a natively compiled stored procedure.

Errors Unaffected by a TRY...CATCH Construct

TRY...CATCH constructs do not trap the following conditions:

- Warnings or informational messages that have a severity of 10 or lower.
- Errors that have a severity of 20 or higher that stop the SQL Server Database Engine task processing for the session. If an error occurs that has severity of 20 or higher and the database connection is not disrupted, TRY...CATCH will handle the error.
- Attentions, such as client-interrupt requests or broken client connections.
- When the session is ended by a system administrator by using the KILL statement.

  The following types of errors are not handled by a CATCH block when they occur at the same level of execution as the TRY...CATCH construct:

- Compile errors, such as syntax errors, that prevent a batch from running.
- Errors that occur during statement-level recompilation, such as object name resolution errors that occur after compilation because of deferred name resolution.

  These errors are returned to the level that ran the batch, stored procedure, or trigger.

  If an error occurs during compilation or statement-level recompilation at a lower execution level (for example, when executing sp_executesql or a user-defined stored procedure) inside the TRY block, the error occurs at a lower level than the TRY...CATCH construct and will be handled by the associated CATCH block.

  The following example shows how an object name resolution error generated by a `SELECT` statement is not caught by the `TRY...CATCH` construct, but is caught by the `CATCH` block when the same `SELECT` statement is executed inside a stored procedure.

```
BEGIN TRY
    -- Table does not exist; object name resolution
    -- error not caught.
    SELECT * FROM NonexistentTable;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber
       ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

The error is not caught and control passes out of the `TRY...CATCH` construct to the next higher level.

Running the `SELECT` statement inside a stored procedure will cause the error to occur at a level lower than the `TRY` block. The error will be handled by the `TRY...CATCH` construct.

```
-- Verify that the stored procedure does not exist.
IF OBJECT_ID ( N'usp_ExampleProc', N'P' ) IS NOT NULL
    DROP PROCEDURE usp_ExampleProc;
GO

-- Create a stored procedure that will cause an
-- object resolution error.
CREATE PROCEDURE usp_ExampleProc
AS
    SELECT * FROM NonexistentTable;
GO

BEGIN TRY
    EXECUTE usp_ExampleProc;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber
        ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

Uncommittable Transactions and XACT_STATE

If an error generated in a TRY block causes the state of the current transaction to be invalidated, the transaction is classified as an uncommittable transaction. An error that ordinarily ends a transaction outside a TRY block causes a transaction to enter an uncommittable state when the error occurs inside a TRY block. An uncommittable transaction can only perform read operations or a ROLLBACK TRANSACTION. The transaction cannot execute any Transact-SQL statements that would generate a write operation or a COMMIT TRANSACTION. The XACT_STATE function returns a value of -1 if a transaction has been classified as an uncommittable transaction. When a batch finishes, the Database Engine rolls back any active uncommittable transactions. If no error message was sent when the transaction entered an uncommittable state, when the batch finishes, an error message will be sent to the client application. This indicates that an uncommittable transaction was detected and rolled back.

For more information about uncommittable transactions and the XACT_STATE function, see XACT_STATE (Transact-SQL).

Examples

A. Using TRY...CATCH

The following example shows a `SELECT` statement that will generate a divide-by-zero error. The error causes execution to jump to the associated `CATCH` block.

```
BEGIN TRY
    -- Generate a divide-by-zero error.
    SELECT 1/0;
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;
END CATCH;
GO
```

B. Using TRY...CATCH in a transaction

The following example shows how a `TRY...CATCH` block works inside a transaction. The statement inside the `TRY` block generates a constraint violation error.

```
BEGIN TRANSACTION;

BEGIN TRY
    -- Generate a constraint violation error.
    DELETE FROM Production.Product
    WHERE ProductID = 980;
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
```

C. Using TRY...CATCH with XACT_STATE

The following example shows how to use the `TRY...CATCH` construct to handle errors that occur inside a transaction. The `XACT_STATE` function determines whether the transaction should be committed or rolled back. In this example, `SET XACT_ABORT` is `ON`. This makes the transaction uncommittable when the constraint violation error occurs.

```
-- Check to see whether this stored procedure exists.
IF OBJECT_ID (N'usp_GetErrorInfo', N'P') IS NOT NULL
    DROP PROCEDURE usp_GetErrorInfo;
GO

-- Create procedure to retrieve error information.
CREATE PROCEDURE usp_GetErrorInfo
AS
    SELECT
         ERROR_NUMBER() AS ErrorNumber
        ,ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_STATE() AS ErrorState
        ,ERROR_LINE () AS ErrorLine
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_MESSAGE() AS ErrorMessage;
GO

-- SET XACT_ABORT ON will cause the transaction to be uncommittable
-- when the constraint violation occurs.
SET XACT_ABORT ON;

BEGIN TRY
    BEGIN TRANSACTION;
        -- A FOREIGN KEY constraint exists on this table. This
        -- statement will generate a constraint violation error.
        DELETE FROM Production.Product
            WHERE ProductID = 980;

    -- If the DELETE statement succeeds, commit the transaction.
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    -- Execute error retrieval routine.
    EXECUTE usp_GetErrorInfo;

    -- Test XACT_STATE:
        -- If 1, the transaction is committable.
        -- If -1, the transaction is uncommittable and should
        --     be rolled back.
        -- XACT_STATE = 0 means that there is no transaction and
        --     a commit or rollback operation would generate an error.

    -- Test whether the transaction is uncommittable.
    IF (XACT_STATE()) = -1
    BEGIN
        PRINT
            N'The transaction is in an uncommittable state.' +
            'Rolling back transaction.'
        ROLLBACK TRANSACTION;
    END;

    -- Test whether the transaction is committable.
    IF (XACT_STATE()) = 1
    BEGIN
        PRINT
            N'The transaction is committable.' +
            'Committing transaction.'
        COMMIT TRANSACTION;
    END;
END CATCH;
GO
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

D. Using TRY...CATCH

The following example shows a `SELECT` statement that will generate a divide-by-zero error. The error causes execution to jump to the associated `CATCH` block.

```
BEGIN TRY
    -- Generate a divide-by-zero error.
    SELECT 1/0;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber
        ,ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_STATE() AS ErrorState
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

See Also

THROW (Transact-SQL)
Database Engine Error Severities
ERROR_LINE (Transact-SQL)
ERROR_MESSAGE (Transact-SQL)
ERROR_NUMBER (Transact-SQL)
ERROR_PROCEDURE (Transact-SQL)
ERROR_SEVERITY (Transact-SQL)
ERROR_STATE (Transact-SQL)
RAISERROR (Transact-SQL)
@@ERROR (Transact-SQL)
GOTO (Transact-SQL)
BEGIN...END (Transact-SQL)
XACT_STATE (Transact-SQL)
SET XACT_ABORT (Transact-SQL)

WAITFOR (Transact-SQL)

6/20/2018 • 4 minutes to read • Edit Online

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

Blocks the execution of a batch, stored procedure, or transaction until a specified time or time interval is reached, or a specified statement modifies or returns at least one row.

Transact-SQL Syntax Conventions

Syntax

```
WAITFOR
{
    DELAY 'time_to_pass'
  | TIME 'time_to_execute'
  | [ ( receive_statement ) | ( get_conversation_group_statement ) ]
    [ , TIMEOUT timeout ]
}
```

Arguments

DELAY
Is the specified period of time that must pass, up to a maximum of 24 hours, before execution of a batch, stored procedure, or transaction proceeds.

'*time_to_pass*'
Is the period of time to wait. *time_to_pass* can be specified in one of the acceptable formats for **datetime** data, or it can be specified as a local variable. Dates cannot be specified; therefore, the date part of the **datetime** value is not allowed. This is formatted as hh:mm[[:ss].mss].

TIME
Is the specified time when the batch, stored procedure, or transaction runs.

'*time_to_execute*'
Is the time at which the WAITFOR statement finishes. *time_to_execute* can be specified in one of the acceptable formats for **datetime** data, or it can be specified as a local variable. Dates cannot be specified; therefore, the date part of the **datetime** value is not allowed. This is formatted as hh:mm[[:ss].mss] and can optionally include the date of 1900-01-01.

receive_statement
Is a valid RECEIVE statement.

> **IMPORTANT**
> WAITFOR with a *receive_statement* is applicable only to Service Broker messages. For more information, see RECEIVE (Transact-SQL).

get_conversation_group_statement
Is a valid GET CONVERSATION GROUP statement.

IMPORTANT

WAITFOR with a *get_conversation_group_statement* is applicable only to Service Broker messages. For more information, see GET CONVERSATION GROUP (Transact-SQL).

TIMEOUT *timeout*

Specifies the period of time, in milliseconds, to wait for a message to arrive on the queue.

IMPORTANT

Specifying WAITFOR with TIMEOUT is applicable only to Service Broker messages. For more information, see RECEIVE (Transact-SQL) and GET CONVERSATION GROUP (Transact-SQL).

Remarks

While executing the WAITFOR statement, the transaction is running and no other requests can run under the same transaction.

The actual time delay may vary from the time specified in *time_to_pass*, *time_to_execute*, or *timeout* and depends on the activity level of the server. The time counter starts when the thread associated with the WAITFOR statement is scheduled. If the server is busy, the thread may not be immediately scheduled; therefore, the time delay may be longer than the specified time.

WAITFOR does not change the semantics of a query. If a query cannot return any rows, WAITFOR will wait forever or until TIMEOUT is reached, if specified.

Cursors cannot be opened on WAITFOR statements.

Views cannot be defined on WAITFOR statements.

When the query exceeds the query wait option, the WAITFOR statement argument can complete without running. For more information about the configuration option, see Configure the query wait Server Configuration Option. To see the active and waiting processes, use sp_who.

Each WAITFOR statement has a thread associated with it. If many WAITFOR statements are specified on the same server, many threads can be tied up waiting for these statements to run. SQL Server monitors the number of threads associated with WAITFOR statements, and randomly selects some of these threads to exit if the server starts to experience thread starvation.

You can create a deadlock by running a query with WAITFOR within a transaction that also holds locks preventing changes to the rowset that the WAITFOR statement is trying to access. SQL Server identifies these scenarios and returns an empty result set if the chance of such a deadlock exists.

Caution

Including WAITFOR will slow the completion of the SQL Server process and can result in a timeout message in the application. If necessary, adjust the timeout setting for the connection at the application level.

Examples

A. Using WAITFOR TIME

The following example executes the stored procedure `sp_update_job` in the msdb database at 10:20 P.M. (`22:20`).

```
EXECUTE sp_add_job @job_name = 'TestJob';
BEGIN
    WAITFOR TIME '22:20';
    EXECUTE sp_update_job @job_name = 'TestJob',
        @new_name = 'UpdatedJob';
END;
GO
```

B. Using WAITFOR DELAY

The following example executes the stored procedure after a two-hour delay.

```
BEGIN
    WAITFOR DELAY '02:00';
    EXECUTE sp_helpdb;
END;
GO
```

C. Using WAITFOR DELAY with a local variable

The following example shows how a local variable can be used with the `WAITFOR DELAY` option. A stored procedure is created to wait for a variable period of time and then returns information to the user as to the number of hours, minutes, and seconds that have elapsed.

```
IF OBJECT_ID('dbo.TimeDelay_hh_mm_ss','P') IS NOT NULL
    DROP PROCEDURE dbo.TimeDelay_hh_mm_ss;
GO
CREATE PROCEDURE dbo.TimeDelay_hh_mm_ss
    (
    @DelayLength char(8)= '00:00:00'
    )
AS
DECLARE @ReturnInfo varchar(255)
IF ISDATE('2000-01-01 ' + @DelayLength + '.000') = 0
    BEGIN
        SELECT @ReturnInfo = 'Invalid time ' + @DelayLength
        + ',hh:mm:ss, submitted.';
        -- This PRINT statement is for testing, not use in production.
        PRINT @ReturnInfo
        RETURN(1)
    END
BEGIN
    WAITFOR DELAY @DelayLength
    SELECT @ReturnInfo = 'A total time of ' + @DelayLength + ',
        hh:mm:ss, has elapsed! Your time is up.'
    -- This PRINT statement is for testing, not use in production.
    PRINT @ReturnInfo;
END;
GO
/* This statement executes the dbo.TimeDelay_hh_mm_ss procedure. */
EXEC TimeDelay_hh_mm_ss '00:00:10';
GO
```

Here is the result set.

```
A total time of 00:00:10, in hh:mm:ss, has elapsed. Your time is up.
```

See Also

Control-of-Flow Language (Transact-SQL)
datetime (Transact-SQL)

WHILE (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Sets a condition for the repeated execution of an SQL statement or statement block. The statements are executed repeatedly as long as the specified condition is true. The execution of statements in the WHILE loop can be controlled from inside the loop with the BREAK and CONTINUE keywords.

Transact-SQL Syntax Conventions

Syntax

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

WHILE Boolean_expression
     { sql_statement | statement_block | BREAK | CONTINUE }
```

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

WHILE Boolean_expression
     { sql_statement | statement_block | BREAK }
```

Arguments

Boolean_expression
Is an expression that returns **TRUE** or **FALSE**. If the Boolean expression contains a SELECT statement, the SELECT statement must be enclosed in parentheses.

{*sql_statement* | *statement_block*}
Is any Transact-SQL statement or statement grouping as defined with a statement block. To define a statement block, use the control-of-flow keywords BEGIN and END.

BREAK
Causes an exit from the innermost WHILE loop. Any statements that appear after the END keyword, marking the end of the loop, are executed.

CONTINUE
Causes the WHILE loop to restart, ignoring any statements after the CONTINUE keyword.

Remarks

If two or more WHILE loops are nested, the inner BREAK exits to the next outermost loop. All the statements after the end of the inner loop run first, and then the next outermost loop restarts.

Examples

A. Using BREAK and CONTINUE with nested IF...ELSE and WHILE

In the following example, if the average list price of a product is less than `$300`, the `WHILE` loop doubles the prices and then selects the maximum price. If the maximum price is less than or equal to `$500`, the `WHILE` loop

restarts and doubles the prices again. This loop continues doubling the prices until the maximum price is greater than `$500`, and then exits the `WHILE` loop and prints a message.

```
USE AdventureWorks2012;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
   UPDATE Production.Product
      SET ListPrice = ListPrice * 2
   SELECT MAX(ListPrice) FROM Production.Product
   IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
      BREAK
   ELSE
      CONTINUE
END
PRINT 'Too much for the market to bear';
```

B. Using WHILE in a cursor

The following example uses `@@FETCH_STATUS` to control cursor activities in a `WHILE` loop.

```
DECLARE Employee_Cursor CURSOR FOR
SELECT EmployeeID, Title
FROM AdventureWorks2012.HumanResources.Employee
WHERE JobTitle = 'Marketing Specialist';
OPEN Employee_Cursor;
FETCH NEXT FROM Employee_Cursor;
WHILE @@FETCH_STATUS = 0
   BEGIN
      FETCH NEXT FROM Employee_Cursor;
   END;
CLOSE Employee_Cursor;
DEALLOCATE Employee_Cursor;
GO
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

C: Simple While Loop

In the following example, if the average list price of a product is less than `$300`, the `WHILE` loop doubles the prices and then selects the maximum price. If the maximum price is less than or equal to `$500`, the `WHILE` loop restarts and doubles the prices again. This loop continues doubling the prices until the maximum price is greater than `$500`, and then exits the `WHILE` loop.

```
-- Uses AdventureWorks

WHILE ( SELECT AVG(ListPrice) FROM dbo.DimProduct) < $300
BEGIN
    UPDATE dbo.DimProduct
        SET ListPrice = ListPrice * 2;
    SELECT MAX ( ListPrice) FROM dbo.DimProduct
    IF ( SELECT MAX (ListPrice) FROM dbo.DimProduct) > $500
        BREAK;
END
```

See Also

ALTER TRIGGER (Transact-SQL)
Control-of-Flow Language (Transact-SQL)

CREATE TRIGGER (Transact-SQL)
Cursors (Transact-SQL)
SELECT (Transact-SQL)

Cursors (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Microsoft SQL Server statements produce a complete result set, but there are times when the results are best processed one row at a time. Opening a cursor on a result set allows processing the result set one row at a time. You can assign a cursor to a variable or parameter with a **cursor** data type.

Cursor operations are supported on these statements:

CLOSE

CREATE PROCEDURE

DEALLOCATE

DECLARE CURSOR

DECLARE @local_variable

DELETE

FETCH

OPEN

UPDATE

SET

These system functions and system stored procedures also support cursors:

@@CURSOR_ROWS

CURSOR_STATUS

@@FETCH_STATUS

sp_cursor_list

sp_describe_cursor

sp_describe_cursor_columns

sp_describe_cursor_tables

See Also

Cursors

CLOSE (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Closes an open cursor by releasing the current result set and freeing any cursor locks held on the rows on which the cursor is positioned. CLOSE leaves the data structures available for reopening, but fetches and positioned updates are not allowed until the cursor is reopened. CLOSE must be issued on an open cursor; CLOSE is not allowed on cursors that have only been declared or are already closed.

Transact-SQL Syntax Conventions

Syntax

```
CLOSE { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Arguments

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

cursor_name
Is the name of an open cursor. If both a global and a local cursor exist with *cursor_name* as their name, *cursor_name* refers to the global cursor when GLOBAL is specified; otherwise, *cursor_name* refers to the local cursor.

cursor_variable_name
Is the name of a cursor variable associated with an open cursor.

Examples

The following example shows the correct placement of the `CLOSE` statement in a cursor-based process.

```
DECLARE Employee_Cursor CURSOR FOR
SELECT EmployeeID, Title FROM AdventureWorks2012.HumanResources.Employee;
OPEN Employee_Cursor;
FETCH NEXT FROM Employee_Cursor;
WHILE @@FETCH_STATUS = 0
   BEGIN
      FETCH NEXT FROM Employee_Cursor;
   END;
CLOSE Employee_Cursor;
DEALLOCATE Employee_Cursor;
GO
```

See Also

Cursors
Cursors (Transact-SQL)
DEALLOCATE (Transact-SQL)

FETCH (Transact-SQL)
OPEN (Transact-SQL)

DEALLOCATE (Transact-SQL)

6/20/2018 • 3 minutes to read • Edit Online

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

Removes a cursor reference. When the last cursor reference is deallocated, the data structures comprising the cursor are released by Microsoft SQL Server.

Transact-SQL Syntax Conventions

Syntax

```
DEALLOCATE { { [ GLOBAL ] cursor_name } | @cursor_variable_name }
```

Arguments

cursor_name
Is the name of an already declared cursor. If both a global and a local cursor exist with *cursor_name* as their name, *cursor_name* refers to the global cursor if GLOBAL is specified and to the local cursor if GLOBAL is not specified.

@cursor_variable_name
Is the name of a **cursor** variable. *@cursor_variable_name* must be of type **cursor**.

Remarks

Statements that operate on cursors use either a cursor name or a cursor variable to refer to the cursor. DEALLOCATE removes the association between a cursor and the cursor name or cursor variable. If a name or variable is the last one referencing the cursor, the cursor is deallocated and any resources used by the cursor are freed. Scroll locks used to protect the isolation of fetches are freed at DEALLOCATE. Transaction locks used to protect updates, including positioned updates made through the cursor, are held until the end of the transaction.

The DECLARE CURSOR statement allocates and associates a cursor with a cursor name.

```
DECLARE abc SCROLL CURSOR FOR
SELECT * FROM Person.Person;
```

After a cursor name is associated with a cursor, the name cannot be used for another cursor of the same scope (GLOBAL or LOCAL) until this cursor has been deallocated.

A cursor variable is associated with a cursor using one of two methods:

- By name using a SET statement that sets a cursor to a cursor variable.

  ```
  DECLARE @MyCrsrRef CURSOR;
  SET @MyCrsrRef = abc;
  ```

- A cursor can also be created and associated with a variable without having a cursor name defined.

```
DECLARE @MyCursor CURSOR;
SET @MyCursor = CURSOR LOCAL SCROLL FOR
SELECT * FROM Person.Person;
```

A DEALLOCATE @*cursor_variable_name* statement removes only the reference of the named variable to the cursor. The variable is not deallocated until it goes out of scope at the end of the batch, stored procedure, or trigger. After a DEALLOCATE @*cursor_variable_name* statement, the variable can be associated with another cursor using the SET statement.

```
USE AdventureWorks2012;
GO

DECLARE @MyCursor CURSOR;
SET @MyCursor = CURSOR LOCAL SCROLL FOR
    SELECT * FROM Sales.SalesPerson;

DEALLOCATE @MyCursor;

SET @MyCursor = CURSOR LOCAL SCROLL FOR
    SELECT * FROM Sales.SalesTerritory;
GO
```

A cursor variable does not have to be explicitly deallocated. The variable is implicitly deallocated when it goes out of scope.

Permissions

DEALLOCATE permissions default to any valid user.

Examples

The following script shows how cursors persist until the last name or until the variable referencing them has been deallocated.

```
USE AdventureWorks2012;
GO
-- Create and open a global named cursor that
-- is visible outside the batch.
DECLARE abc CURSOR GLOBAL SCROLL FOR
    SELECT * FROM Sales.SalesPerson;
OPEN abc;
GO
-- Reference the named cursor with a cursor variable.
DECLARE @MyCrsrRef1 CURSOR;
SET @MyCrsrRef1 = abc;
-- Now deallocate the cursor reference.
DEALLOCATE @MyCrsrRef1;
-- Cursor abc still exists.
FETCH NEXT FROM abc;
GO
-- Reference the named cursor again.
DECLARE @MyCrsrRef2 CURSOR;
SET @MyCrsrRef2 = abc;
-- Now deallocate cursor name abc.
DEALLOCATE abc;
-- Cursor still exists, referenced by @MyCrsrRef2.
FETCH NEXT FROM @MyCrsrRef2;
-- Cursor finally is deallocated when last referencing
-- variable goes out of scope at the end of the batch.
GO
-- Create an unnamed cursor.
DECLARE @MyCursor CURSOR;
SET @MyCursor = CURSOR LOCAL SCROLL FOR
SELECT * FROM Sales.SalesTerritory;
-- The following statement deallocates the cursor
-- because no other variables reference it.
DEALLOCATE @MyCursor;
GO
```

See Also

CLOSE (Transact-SQL)
Cursors
DECLARE @local_variable (Transact-SQL)
FETCH (Transact-SQL)
OPEN (Transact-SQL)

DECLARE CURSOR (Transact-SQL)

6/20/2018 • 10 minutes to read • Edit Online

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

Defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. DECLARE CURSOR accepts both a syntax based on the ISO standard and a syntax using a set of Transact-SQL extensions.

Transact-SQL Syntax Conventions

Syntax

```
ISO Syntax
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
     FOR select_statement
     [ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
[;]
Transact-SQL Extended Syntax
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
     [ FORWARD_ONLY | SCROLL ]
     [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
     [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
     [ TYPE_WARNING ]
     FOR select_statement
     [ FOR UPDATE [ OF column_name [ ,...n ] ] ]
[;]
```

Arguments

cursor_name
Is the name of the Transact-SQL server cursor defined. *cursor_name* must conform to the rules for identifiers.

INSENSITIVE
Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in **tempdb**; therefore, modifications made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow modifications. When ISO syntax is used, if INSENSITIVE is omitted, committed deletes and updates made to the underlying tables (by any user) are reflected in subsequent fetches.

SCROLL
Specifies that all fetch options (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) are available. If SCROLL is not specified in an ISO DECLARE CURSOR, NEXT is the only fetch option supported. SCROLL cannot be specified if FAST_FORWARD is also specified.

select_statement
Is a standard SELECT statement that defines the result set of the cursor. The keywords FOR BROWSE, and INTO are not allowed within *select_statement* of a cursor declaration.

SQL Server implicitly converts the cursor to another type if clauses in *select_statement* conflict with the functionality of the requested cursor type.

READ ONLY
Prevents updates made through this cursor. The cursor cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. This option overrides the default capability of a cursor to be updated.

UPDATE [OF *column_name* [**,**...*n*]]
Defines updatable columns within the cursor. If OF *column_name* [**,**...*n*] is specified, only the columns listed allow modifications. If UPDATE is specified without a column list, all columns can be updated.

cursor_name
Is the name of the Transact-SQL server cursor defined. *cursor_name* must conform to the rules for identifiers.

LOCAL
Specifies that the scope of the cursor is local to the batch, stored procedure, or trigger in which the cursor was created. The cursor name is only valid within this scope. The cursor can be referenced by local cursor variables in the batch, stored procedure, or trigger, or a stored procedure OUTPUT parameter. An OUTPUT parameter is used to pass the local cursor back to the calling batch, stored procedure, or trigger, which can assign the parameter to a cursor variable to reference the cursor after the stored procedure terminates. The cursor is implicitly deallocated when the batch, stored procedure, or trigger terminates, unless the cursor was passed back in an OUTPUT parameter. If it is passed back in an OUTPUT parameter, the cursor is deallocated when the last variable referencing it is deallocated or goes out of scope.

GLOBAL
Specifies that the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection. The cursor is only implicitly deallocated at disconnect.

> **NOTE**
> If neither GLOBAL or LOCAL is specified, the default is controlled by the setting of the **default to local cursor** database option.

FORWARD_ONLY
Specifies that the cursor can only be scrolled from the first to the last row. FETCH NEXT is the only supported fetch option. If FORWARD_ONLY is specified without the STATIC, KEYSET, or DYNAMIC keywords, the cursor operates as a DYNAMIC cursor. When neither FORWARD_ONLY nor SCROLL is specified, FORWARD_ONLY is the default, unless the keywords STATIC, KEYSET, or DYNAMIC are specified. STATIC, KEYSET, and DYNAMIC cursors default to SCROLL. Unlike database APIs such as ODBC and ADO, FORWARD_ONLY is supported with STATIC, KEYSET, and DYNAMIC Transact-SQL cursors.

STATIC
Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in **tempdb**; therefore, modifications made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow modifications.

KEYSET
Specifies that the membership and order of rows in the cursor are fixed when the cursor is opened. The set of keys that uniquely identify the rows is built into a table in **tempdb** known as the **keyset**.

> **NOTE**
> If the query references at least one table without a unique index, the keyset cursor is converted to a static cursor.

Changes to nonkey values in the base tables, either made by the cursor owner or committed by other users, are visible as the owner scrolls around the cursor. Inserts made by other users are not visible (inserts cannot be made through a Transact-SQL server cursor). If a row is deleted, an attempt to fetch the row returns an

@@FETCH_STATUS of -2. Updates of key values from outside the cursor resemble a delete of the old row followed by an insert of the new row. The row with the new values is not visible, and attempts to fetch the row with the old values return an @@FETCH_STATUS of -2. The new values are visible if the update is done through the cursor by specifying the WHERE CURRENT OF clause.

DYNAMIC
Defines a cursor that reflects all data changes made to the rows in its result set as you scroll around the cursor. The data values, order, and membership of the rows can change on each fetch. The ABSOLUTE fetch option is not supported with dynamic cursors.

FAST_FORWARD
Specifies a FORWARD_ONLY, READ_ONLY cursor with performance optimizations enabled. FAST_FORWARD cannot be specified if SCROLL or FOR_UPDATE is also specified.

> **NOTE**
> Both FAST_FORWARD and FORWARD_ONLY can be used in the same DECLARE CURSOR statement.

READ_ONLY
Prevents updates made through this cursor. The cursor cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. This option overrides the default capability of a cursor to be updated.

SCROLL_LOCKS
Specifies that positioned updates or deletes made through the cursor are guaranteed to succeed. SQL Server locks the rows as they are read into the cursor to ensure their availability for later modifications. SCROLL_LOCKS cannot be specified if FAST_FORWARD or STATIC is also specified.

OPTIMISTIC
Specifies that positioned updates or deletes made through the cursor do not succeed if the row has been updated since it was read into the cursor. SQL Server does not lock rows as they are read into the cursor. It instead uses comparisons of **timestamp** column values, or a checksum value if the table has no **timestamp** column, to determine whether the row was modified after it was read into the cursor. If the row was modified, the attempted positioned update or delete fails. OPTIMISTIC cannot be specified if FAST_FORWARD is also specified.

TYPE_WARNING
Specifies that a warning message is sent to the client when the cursor is implicitly converted from the requested type to another.

select_statement
Is a standard SELECT statement that defines the result set of the cursor. The keywords COMPUTE, COMPUTE BY, FOR BROWSE, and INTO are not allowed within *select_statement* of a cursor declaration.

> **NOTE**
> You can use a query hint within a cursor declaration; however, if you also use the FOR UPDATE OF clause, specify OPTION (*query_hint*) after FOR UPDATE OF.

SQL Server implicitly converts the cursor to another type if clauses in *select_statement* conflict with the functionality of the requested cursor type. For more information, see Implicit Cursor Conversions.

FOR UPDATE [OF *column_name* [**,**...*n*]]
Defines updatable columns within the cursor. If OF *column_name* [**,**...*n*] is supplied, only the columns listed allow modifications. If UPDATE is specified without a column list, all columns can be updated, unless the READ_ONLY concurrency option was specified.

Remarks

DECLARE CURSOR defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. The OPEN statement populates the result set, and FETCH returns a row from the result set. The CLOSE statement releases the current result set associated with the cursor. The DEALLOCATE statement releases the resources used by the cursor.

The first form of the DECLARE CURSOR statement uses the ISO syntax for declaring cursor behaviors. The second form of DECLARE CURSOR uses Transact-SQL extensions that allow you to define cursors using the same cursor types used in the database API cursor functions of ODBC or ADO.

You cannot mix the two forms. If you specify the SCROLL or INSENSITIVE keywords before the CURSOR keyword, you cannot use any keywords between the CURSOR and FOR *select_statement* keywords. If you specify any keywords between the CURSOR and FOR *select_statement* keywords, you cannot specify SCROLL or INSENSITIVE before the CURSOR keyword.

If a DECLARE CURSOR using Transact-SQL syntax does not specify READ_ONLY, OPTIMISTIC, or SCROLL_LOCKS, the default is as follows:

- If the SELECT statement does not support updates (insufficient permissions, accessing remote tables that do not support updates, and so on), the cursor is READ_ONLY.
- STATIC and FAST_FORWARD cursors default to READ_ONLY.
- DYNAMIC and KEYSET cursors default to OPTIMISTIC.

  Cursor names can be referenced only by other Transact-SQL statements. They cannot be referenced by database API functions. For example, after declaring a cursor, the cursor name cannot be referenced from OLE DB, ODBC or ADO functions or methods. The cursor rows cannot be fetched using the fetch functions or methods of the APIs; the rows can be fetched only by Transact-SQL FETCH statements.

  After a cursor has been declared, these system stored procedures can be used to determine the characteristics of the cursor.

SYSTEM STORED PROCEDURES	DESCRIPTION
sp_cursor_list	Returns a list of cursors currently visible on the connection and their attributes.
sp_describe_cursor	Describes the attributes of a cursor, such as whether it is a forward-only or scrolling cursor.
sp_describe_cursor_columns	Describes the attributes of the columns in the cursor result set.
sp_describe_cursor_tables	Describes the base tables accessed by the cursor.

Variables may be used as part of the *select_statement* that declares a cursor. Cursor variable values do not change after a cursor is declared.

Permissions

DECLARE CURSOR permissions default to any user that has SELECT permissions on the views, tables, and columns used in the cursor.

Limitations and Restrictions

You cannot use cursors or triggers on a table with a clustered columnstore index. This restriction does not apply to nonclustered columnstore indexes; you can use cursors and triggers on a table with a nonclustered columnstore index.

Examples

A. Using simple cursor and syntax

The result set generated at the opening of this cursor includes all rows and all columns in the table. This cursor can be updated, and all updates and deletes are represented in fetches made against this cursor. `FETCH NEXT` is the only fetch available because the `SCROLL` option has not been specified.

```
DECLARE vend_cursor CURSOR
    FOR SELECT * FROM Purchasing.Vendor
OPEN vend_cursor
FETCH NEXT FROM vend_cursor;
```

B. Using nested cursors to produce report output

The following example shows how cursors can be nested to produce complex reports. The inner cursor is declared for each vendor.

```
SET NOCOUNT ON;

DECLARE @vendor_id int, @vendor_name nvarchar(50),
    @message varchar(80), @product nvarchar(50);

PRINT '-------- Vendor Products Report --------';

DECLARE vendor_cursor CURSOR FOR
SELECT VendorID, Name
FROM Purchasing.Vendor
WHERE PreferredVendorStatus = 1
ORDER BY VendorID;

OPEN vendor_cursor

FETCH NEXT FROM vendor_cursor
INTO @vendor_id, @vendor_name

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT ' '
    SELECT @message = '----- Products From Vendor: ' +
        @vendor_name

    PRINT @message

    -- Declare an inner cursor based
    -- on vendor_id from the outer cursor.

    DECLARE product_cursor CURSOR FOR
    SELECT v.Name
    FROM Purchasing.ProductVendor pv, Production.Product v
    WHERE pv.ProductID = v.ProductID AND
    pv.VendorID = @vendor_id  -- Variable value from the outer cursor

    OPEN product_cursor
    FETCH NEXT FROM product_cursor INTO @product

    IF @@FETCH_STATUS <> 0
        PRINT '         <<None>>'

    WHILE @@FETCH_STATUS = 0
    BEGIN

        SELECT @message = '          ' + @product
        PRINT @message
        FETCH NEXT FROM product_cursor INTO @product
        END

    CLOSE product_cursor
    DEALLOCATE product_cursor
        -- Get the next vendor.
    FETCH NEXT FROM vendor_cursor
    INTO @vendor_id, @vendor_name
END
CLOSE vendor_cursor;
DEALLOCATE vendor_cursor;
```

See Also

@@FETCH_STATUS (Transact-SQL)
CLOSE (Transact-SQL)
Cursors (Transact-SQL)
DEALLOCATE (Transact-SQL)

FETCH (Transact-SQL)
SELECT (Transact-SQL)
sp_configure (Transact-SQL)

FETCH (Transact-SQL)

6/20/2018 • 5 minutes to read • Edit Online

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

Retrieves a specific row from a Transact-SQL server cursor.

Transact-SQL Syntax Conventions

Syntax

```
FETCH
          [ [ NEXT | PRIOR | FIRST | LAST
                    | ABSOLUTE { n | @nvar }
                    | RELATIVE { n | @nvar }
               ]
               FROM
          ]
{ { [ GLOBAL ] cursor_name } | @cursor_variable_name }
[ INTO @variable_name [ ,...n ] ]
```

Arguments

NEXT
Returns the result row immediately following the current row and increments the current row to the row returned. If FETCH NEXT is the first fetch against a cursor, it returns the first row in the result set. NEXT is the default cursor fetch option.

PRIOR
Returns the result row immediately preceding the current row, and decrements the current row to the row returned. If FETCH PRIOR is the first fetch against a cursor, no row is returned and the cursor is left positioned before the first row.

FIRST
Returns the first row in the cursor and makes it the current row.

LAST
Returns the last row in the cursor and makes it the current row.

ABSOLUTE { *n*| @*nvar*}
If *n* or @*nvar* is positive, returns the row *n* rows from the front of the cursor and makes the returned row the new current row. If *n* or @*nvar* is negative, returns the row *n* rows before the end of the cursor and makes the returned row the new current row. If *n* or @*nvar* is 0, no rows are returned. *n* must be an integer constant and @*nvar* must be **smallint**, **tinyint**, or **int**.

RELATIVE { *n*| @*nvar*}
If *n* or @*nvar* is positive, returns the row *n* rows beyond the current row and makes the returned row the new current row. If *n* or @*nvar* is negative, returns the row *n* rows prior to the current row and makes the returned row the new current row. If *n* or @*nvar* is 0, returns the current row. If FETCH RELATIVE is specified with *n* or @*nvar* set to negative numbers or 0 on the first fetch done against a cursor, no rows are returned. *n* must be an integer constant and @*nvar* must be **smallint**, **tinyint**, or **int**.

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 exist with *cursor_name* as their name, *cursor_name* to the global cursor if GLOBAL is specified and to the local cursor if GLOBAL is not specified.

@*cursor_variable_name*
Is the name of a cursor variable referencing the open cursor from which the fetch should be made.

INTO @*variable_name*[,*...n*]
Allows data from the columns of a fetch to be placed into local variables. Each variable in the list, from left to right, is associated with the corresponding column in the cursor result set. The data type of each variable must either match or be a supported implicit conversion of the data type of the corresponding result set column. The number of variables must match the number of columns in the cursor select list.

Remarks

If the SCROLL option is not specified in an ISO style DECLARE CURSOR statement, NEXT is the only FETCH option supported. If SCROLL is specified in an ISO style DECLARE CURSOR, all FETCH options are supported.

When the Transact-SQL DECLARE cursor extensions are used, these rules apply:

- If either FORWARD_ONLY or FAST_FORWARD is specified, NEXT is the only FETCH option supported.
- If DYNAMIC, FORWARD_ONLY or FAST_FORWARD are not specified, and one of KEYSET, STATIC, or SCROLL are specified, all FETCH options are supported.
- DYNAMIC SCROLL cursors support all the FETCH options except ABSOLUTE.

  The @@FETCH_STATUS function reports the status of the last FETCH statement. The same information is recorded in the fetch_status column in the cursor returned by sp_describe_cursor. This status information should be used to determine the validity of the data returned by a FETCH statement prior to attempting any operation against that data. For more information, see @@FETCH_STATUS (Transact-SQL).

Permissions

FETCH permissions default to any valid user.

Examples

A. Using FETCH in a simple cursor

The following example declares a simple cursor for the rows in the `Person.Person` table with a last name that starts with `B`, and uses `FETCH NEXT` to step through the rows. The `FETCH` statements return the value for the column specified in `DECLARE CURSOR` as a single-row result set.

```
USE AdventureWorks2012;
GO
DECLARE contact_cursor CURSOR FOR
SELECT LastName FROM Person.Person
WHERE LastName LIKE 'B%'
ORDER BY LastName;

OPEN contact_cursor;

-- Perform the first fetch.
FETCH NEXT FROM contact_cursor;

-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
   -- This is executed as long as the previous fetch succeeds.
   FETCH NEXT FROM contact_cursor;
END

CLOSE contact_cursor;
DEALLOCATE contact_cursor;
GO
```

B. Using FETCH to store values in variables

The following example is similar to example A, except the output of the `FETCH` statements is stored in local variables instead of being returned directly to the client. The `PRINT` statement combines the variables into a single string and returns them to the client.

```
USE AdventureWorks2012;
GO
-- Declare the variables to store the values returned by FETCH.
DECLARE @LastName varchar(50), @FirstName varchar(50);

DECLARE contact_cursor CURSOR FOR
SELECT LastName, FirstName FROM Person.Person
WHERE LastName LIKE 'B%'
ORDER BY LastName, FirstName;

OPEN contact_cursor;

-- Perform the first fetch and store the values in variables.
-- Note: The variables are in the same order as the columns
-- in the SELECT statement.

FETCH NEXT FROM contact_cursor
INTO @LastName, @FirstName;

-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN

   -- Concatenate and display the current values in the variables.
   PRINT 'Contact Name: ' + @FirstName + ' ' +  @LastName

   -- This is executed as long as the previous fetch succeeds.
   FETCH NEXT FROM contact_cursor
   INTO @LastName, @FirstName;
END

CLOSE contact_cursor;
DEALLOCATE contact_cursor;
GO
```

C. Declaring a SCROLL cursor and using the other FETCH options

The following example creates a `SCROLL` cursor to allow full scrolling capabilities through the `LAST`, `PRIOR`, `RELATIVE`, and `ABSOLUTE` options.

```
USE AdventureWorks2012;
GO
-- Execute the SELECT statement alone to show the
-- full result set that is used by the cursor.
SELECT LastName, FirstName FROM Person.Person
ORDER BY LastName, FirstName;

-- Declare the cursor.
DECLARE contact_cursor SCROLL CURSOR FOR
SELECT LastName, FirstName FROM Person.Person
ORDER BY LastName, FirstName;

OPEN contact_cursor;

-- Fetch the last row in the cursor.
FETCH LAST FROM contact_cursor;

-- Fetch the row immediately prior to the current row in the cursor.
FETCH PRIOR FROM contact_cursor;

-- Fetch the second row in the cursor.
FETCH ABSOLUTE 2 FROM contact_cursor;

-- Fetch the row that is three rows after the current row.
FETCH RELATIVE 3 FROM contact_cursor;

-- Fetch the row that is two rows prior to the current row.
FETCH RELATIVE -2 FROM contact_cursor;

CLOSE contact_cursor;
DEALLOCATE contact_cursor;
GO
```

See Also

CLOSE (Transact-SQL)
DEALLOCATE (Transact-SQL)
DECLARE CURSOR (Transact-SQL)
OPEN (Transact-SQL)

OPEN (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Opens a Transact-SQL server cursor and populates the cursor by executing the Transact-SQL statement specified on the DECLARE CURSOR or SET *cursor_variable* statement.

Transact-SQL Syntax Conventions

Syntax

```
OPEN { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Arguments

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

cursor_name
Is the name of a declared cursor. If both a global and a local cursor exist with *cursor_name* as their name, *cursor_name* refers to the global cursor if GLOBAL is specified; otherwise, *cursor_name* refers to the local cursor.

cursor_variable_name
Is the name of a cursor variable that references a cursor.

Remarks

If the cursor is declared with the INSENSITIVE or STATIC option, OPEN creates a temporary table to hold the result set. OPEN fails when the size of any row in the result set exceeds the maximum row size for SQL Server tables. If the cursor is declared with the KEYSET option, OPEN creates a temporary table to hold the keyset. The temporary tables are stored in tempdb.

After a cursor has been opened, use the @@CURSOR_ROWS function to receive the number of qualifying rows in the last opened cursor.

> **NOTE**
> SQL Server does not support generating keyset-driven or static Transact-SQL cursors asynchronously. Transact-SQL cursor operations such as OPEN or FETCH are batched, so there is no need for the asynchronous generation of Transact-SQL cursors. SQL Server continues to support asynchronous keyset-driven or static application programming interface (API) server cursors where low latency OPEN is a concern, due to client round trips for each cursor operation.

Examples

The following example opens a cursor and fetches all the rows.

```
DECLARE Employee_Cursor CURSOR FOR
SELECT LastName, FirstName
FROM AdventureWorks2012.HumanResources.vEmployee
WHERE LastName like 'B%';

OPEN Employee_Cursor;

FETCH NEXT FROM Employee_Cursor;
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM Employee_Cursor
END;

CLOSE Employee_Cursor;
DEALLOCATE Employee_Cursor;
```

See Also

CLOSE (Transact-SQL)
@@CURSOR_ROWS (Transact-SQL)
DEALLOCATE (Transact-SQL)
DECLARE CURSOR (Transact-SQL)
FETCH (Transact-SQL)

Expressions (Transact-SQL)

8/27/2018 • 4 minutes to read • Edit Online

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

Is a combination of symbols and operators that the SQL Server Database Engine evaluates to obtain a single data value. Simple expressions can be a single constant, variable, column, or scalar function. Operators can be used to join two or more simple expressions into a complex expression.

Transact-SQL Syntax Conventions

Syntax

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

{ constant | scalar_function | [ table_name. ] column | variable
    | ( expression ) | ( scalar_subquery )
    | { unary_operator } expression
    | expression { binary_operator } expression
    | ranking_windowed_function | aggregate_windowed_function
}
```

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

-- Expression in a SELECT statement
<expression> ::=
{
    constant
    | scalar_function
    | column
    | variable
    | ( expression  )
    | { unary_operator } expression
    | expression { binary_operator } expression
}
[ COLLATE Windows_collation_name ]

-- Scalar Expression in a DECLARE, SET, IF…ELSE, or WHILE statement
<scalar_expression> ::=
{
    constant
    | scalar_function
    | variable
    | ( expression  )
    | (scalar_subquery )
    | { unary_operator } expression
    | expression { binary_operator } expression
}
[ COLLATE { Windows_collation_name ]
```

Arguments

TERM	DEFINITION
constant	Is a symbol that represents a single, specific data value. For more information, see Constants (Transact-SQL).
scalar_function	Is a unit of Transact-SQL syntax that provides a specific service and returns a single value. *scalar_function* can be built-in scalar functions, such as the SUM, GETDATE, or CAST functions, or scalar user-defined functions.
[*table_name*.]	Is the name or alias of a table.
column	Is the name of a column. Only the name of the column is allowed in an expression.
variable	Is the name of a variable, or parameter. For more information, see DECLARE @local_variable (Transact-SQL).
(*expression* **)**	Is any valid expression as defined in this topic. The parentheses are grouping operators that make sure that all the operators in the expression within the parentheses are evaluated before the resulting expression is combined with another.

(*scalar_subquery* **)**	Is a subquery that returns one value. For example:
	`SELECT MAX(UnitPrice)`
	`FROM Products`

{ *unary_operator* }	Unary operators can be applied only to expressions that evaluate to any one of the data types of the numeric data type category. Is an operator that has only one numeric operand: + indicates a positive number. - indicates a negative number. ~ indicates the one's complement operator.
{ *binary_operator* }	Is an operator that defines the way two expressions are combined to yield a single result. *binary_operator* can be an arithmetic operator, the assignment operator (=), a bitwise operator, a comparison operator, a logical operator, the string concatenation operator (+), or a unary operator. For more information about operators, see Operators (Transact-SQL).
ranking_windowed_function	Is any Transact-SQL ranking function. For more information, see Ranking Functions (Transact-SQL).
aggregate_windowed_function	Is any Transact-SQL aggregate function with the OVER clause. For more information, see OVER Clause (Transact-SQL).

Expression Results

For a simple expression made up of a single constant, variable, scalar function, or column name: the data type, collation, precision, scale, and value of the expression is the data type, collation, precision, scale, and value of the referenced element.

When two expressions are combined by using comparison or logical operators, the resulting data type is Boolean and the value is one of the following: TRUE, FALSE, or UNKNOWN. For more information about Boolean data types, see Comparison Operators (Transact-SQL).

When two expressions are combined by using arithmetic, bitwise, or string operators, the operator determines the resulting data type.

Complex expressions made up of many symbols and operators evaluate to a single-valued result. The data type, collation, precision, and value of the resulting expression is determined by combining the component expressions, two at a time, until a final result is reached. The sequence in which the expressions are combined is defined by the precedence of the operators in the expression.

Remarks

Two expressions can be combined by an operator if they both have data types supported by the operator and at least one of these conditions is true:

- The expressions have the same data type.
- The data type with the lower precedence can be implicitly converted to the data type with the higher data type precedence.

  If the expressions do not meet these conditions, the CAST or CONVERT functions can be used to explicitly convert the data type with the lower precedence to either the data type with the higher precedence or to an intermediate data type that can be implicitly converted to the data type with the higher precedence.

  If there is no supported implicit or explicit conversion, the two expressions cannot be combined.

  The collation of any expression that evaluates to a character string is set by following the rules of collation precedence. For more information, see Collation Precedence (Transact-SQL).

  In a programming language such as C or Microsoft Visual Basic, an expression always evaluates to a single result. Expressions in a Transact-SQL select list follow a variation on this rule: The expression is evaluated individually for each row in the result set. A single expression may have a different value in each row of the result set, but each row has only one value for the expression. For example, in the following `SELECT` statement both the reference to `ProductID` and the term `1+2` in the select list are expressions:

```
USE AdventureWorks2012;
GO
SELECT ProductID, 1+2
FROM Production.Product;
GO
```

The expression `1+2` evaluates to `3` in each row in the result set. Although the expression `ProductID` generates a unique value in each result set row, each row only has one value for `ProductID`.

See Also

AT TIME ZONE (Transact-SQL)
CASE (Transact-SQL)
CAST and CONVERT (Transact-SQL)

COALESCE (Transact-SQL)
Data Type Conversion (Database Engine)
Data Type Precedence (Transact-SQL)
Data Types (Transact-SQL)
Built-in Functions (Transact-SQL)
LIKE (Transact-SQL)
NULLIF (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)

CASE (Transact-SQL)

8/27/2018 • 8 minutes to read • Edit Online

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

Evaluates a list of conditions and returns one of multiple possible result expressions.

The CASE expression has two formats:

- The simple CASE expression compares an expression to a set of simple expressions to determine the result.
- The searched CASE expression evaluates a set of Boolean expressions to determine the result.

  Both formats support an optional ELSE argument.

  CASE can be used in any statement or clause that allows a valid expression. For example, you can use CASE in statements such as SELECT, UPDATE, DELETE and SET, and in clauses such as select_list, IN, WHERE, ORDER BY, and HAVING.

  Transact-SQL Syntax Conventions

Syntax

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

Simple CASE expression:
CASE input_expression
     WHEN when_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
Searched CASE expression:
CASE
     WHEN Boolean_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
```

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

CASE
     WHEN when_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
```

Arguments

input_expression
Is the expression evaluated when the simple CASE format is used. *input_expression* is any valid expression.

WHEN *when_expression*
Is a simple expression to which *input_expression* is compared when the simple CASE format is used. *when_expression* is any valid expression. The data types of *input_expression* and each *when_expression* must be the same or must be an implicit conversion.

THEN *result_expression*
Is the expression returned when *input_expression* equals *when_expression* evaluates to TRUE, or *Boolean_expression* evaluates to TRUE. *result expression* is any valid expression.

ELSE *else_result_expression*
Is the expression returned if no comparison operation evaluates to TRUE. If this argument is omitted and no comparison operation evaluates to TRUE, CASE returns NULL. *else_result_expression* is any valid expression. The data types of *else_result_expression* and any *result_expression* must be the same or must be an implicit conversion.

WHEN *Boolean_expression*
Is the Boolean expression evaluated when using the searched CASE format. *Boolean_expression* is any valid Boolean expression.

Return Types

Returns the highest precedence type from the set of types in *result_expressions* and the optional *else_result_expression*. For more information, see Data Type Precedence (Transact-SQL).

Return Values

Simple CASE expression:

The simple CASE expression operates by comparing the first expression to the expression in each WHEN clause for equivalency. If these expressions are equivalent, the expression in the THEN clause will be returned.

- Allows only an equality check.
- In the order specified, evaluates input_expression = when_expression for each WHEN clause.
- Returns the *result_expression* of the first *input_expression* = *when_expression* that evaluates to TRUE.
- If no *input_expression* = *when_expression* evaluates to TRUE, the SQL Server Database Engine returns the *else_result_expression* if an ELSE clause is specified, or a NULL value if no ELSE clause is specified.

Searched CASE expression:

- Evaluates, in the order specified, *Boolean_expression* for each WHEN clause.
- Returns *result_expression* of the first *Boolean_expression* that evaluates to TRUE.
- If no *Boolean_expression* evaluates to TRUE, the Database Engine returns the *else_result_expression* if an ELSE clause is specified, or a NULL value if no ELSE clause is specified.

Remarks

SQL Server allows for only 10 levels of nesting in CASE expressions.

The CASE expression cannot be used to control the flow of execution of Transact-SQL statements, statement blocks, user-defined functions, and stored procedures. For a list of control-of-flow methods, see Control-of-Flow Language (Transact-SQL).

The CASE expression evaluates its conditions sequentially and stops with the first condition whose condition is satisfied. In some situations, an expression is evaluated before a CASE expression receives the results of the expression as its input. Errors in evaluating these expressions are possible. Aggregate expressions that appear in WHEN arguments to a CASE expression are evaluated first, then provided to the CASE expression. For example, the following query produces a divide by zero error when producing the value of the MAX aggregate. This occurs prior to evaluating the CASE expression.

```
WITH Data (value) AS
(
SELECT 0
UNION ALL
SELECT 1
)
SELECT
   CASE
      WHEN MIN(value) <= 0 THEN 0
      WHEN MAX(1/value) >= 100 THEN 1
   END
FROM Data ;
```

You should only depend on order of evaluation of the WHEN conditions for scalar expressions (including non-correlated sub-queries that return scalars), not for aggregate expressions.

Examples

A. Using a SELECT statement with a simple CASE expression

Within a `SELECT` statement, a simple `CASE` expression allows for only an equality check; no other comparisons are made. The following example uses the `CASE` expression to change the display of product line categories to make them more understandable.

```
USE AdventureWorks2012;
GO
SELECT   ProductNumber, Category =
      CASE ProductLine
         WHEN 'R' THEN 'Road'
         WHEN 'M' THEN 'Mountain'
         WHEN 'T' THEN 'Touring'
         WHEN 'S' THEN 'Other sale items'
         ELSE 'Not for sale'
      END,
   Name
FROM Production.Product
ORDER BY ProductNumber;
GO
```

B. Using a SELECT statement with a searched CASE expression

Within a `SELECT` statement, the searched `CASE` expression allows for values to be replaced in the result set based on comparison values. The following example displays the list price as a text comment based on the price range for a product.

```
USE AdventureWorks2012;
GO
SELECT   ProductNumber, Name, "Price Range" =
      CASE
         WHEN ListPrice =  0 THEN 'Mfg item - not for resale'
         WHEN ListPrice < 50 THEN 'Under $50'
         WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'
         WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'
         ELSE 'Over $1000'
      END
FROM Production.Product
ORDER BY ProductNumber ;
GO
```

C. Using CASE in an ORDER BY clause

The following examples uses the CASE expression in an ORDER BY clause to 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;
```

D. Using CASE in an UPDATE statement

The following example uses the CASE expression in an UPDATE statement to determine the value that is set for the column `VacationHours` for employees with `SalariedFlag` set to 0. When subtracting 10 hours from `VacationHours` results in a negative value, `VacationHours` is increased by 40 hours; otherwise, `VacationHours` is increased by 20 hours. The OUTPUT clause is used to display the before and after vacation values.

```
USE AdventureWorks2012;
GO
UPDATE HumanResources.Employee
SET VacationHours =
    ( CASE
         WHEN ((VacationHours - 10.00) < 0) THEN VacationHours + 40
         ELSE (VacationHours + 20.00)
       END
    )
OUTPUT Deleted.BusinessEntityID, Deleted.VacationHours AS BeforeValue,
       Inserted.VacationHours AS AfterValue
WHERE SalariedFlag = 0;
```

E. Using CASE in a SET statement

The following example uses the CASE expression in a SET statement in the table-valued function `dbo.GetContactInfo`. In the **AdventureWorks2012** database, all data related to people is stored in the `Person.Person` table. For example, the person may be an employee, vendor representative, or a customer. The function returns the first and last name of a given `BusinessEntityID` and the contact type for that person.The CASE expression in the SET statement determines the value to display for the column `ContactType` based on the existence of the `BusinessEntityID` column in the `Employee`, `Vendor`, or `Customer` tables.

```
USE AdventureWorks2012;
GO
CREATE FUNCTION dbo.GetContactInformation(@BusinessEntityID int)
    RETURNS @retContactInformation TABLE
(
BusinessEntityID int NOT NULL,
FirstName nvarchar(50) NULL,
LastName nvarchar(50) NULL,
```

```
ContactType nvarchar(50) NULL,
    PRIMARY KEY CLUSTERED (BusinessEntityID ASC)
)
AS
-- Returns the first name, last name and contact type for the specified contact.
BEGIN
    DECLARE
        @FirstName nvarchar(50),
        @LastName nvarchar(50),
        @ContactType nvarchar(50);

    -- Get common contact information
    SELECT
        @BusinessEntityID = BusinessEntityID,
@FirstName = FirstName,
        @LastName = LastName
    FROM Person.Person
    WHERE BusinessEntityID = @BusinessEntityID;

    SET @ContactType =
        CASE
            -- Check for employee
            WHEN EXISTS(SELECT * FROM HumanResources.Employee AS e
                WHERE e.BusinessEntityID = @BusinessEntityID)
                THEN 'Employee'

            -- Check for vendor
            WHEN EXISTS(SELECT * FROM Person.BusinessEntityContact AS bec
                WHERE bec.BusinessEntityID = @BusinessEntityID)
                THEN 'Vendor'

            -- Check for store
            WHEN EXISTS(SELECT * FROM Purchasing.Vendor AS v
                WHERE v.BusinessEntityID = @BusinessEntityID)
                THEN 'Store Contact'

            -- Check for individual consumer
            WHEN EXISTS(SELECT * FROM Sales.Customer AS c
                WHERE c.PersonID = @BusinessEntityID)
                THEN 'Consumer'
        END;

    -- Return the information to the caller
    IF @BusinessEntityID IS NOT NULL
    BEGIN
        INSERT @retContactInformation
        SELECT @BusinessEntityID, @FirstName, @LastName, @ContactType;
    END;

    RETURN;
END;
GO

SELECT BusinessEntityID, FirstName, LastName, ContactType
FROM dbo.GetContactInformation(2200);
GO
SELECT BusinessEntityID, FirstName, LastName, ContactType
FROM dbo.GetContactInformation(5);
```

F. Using CASE in a HAVING clause

The following example uses the CASE expression in a HAVING clause to restrict the rows returned by the SELECT statement. The statement returns the maximum hourly rate for each job title in the `HumanResources.Employee` table. The HAVING clause restricts the titles to those that are held by men with a maximum pay rate greater than 40 dollars or women with a maximum pay rate greater than 42 dollars.

```
USE AdventureWorks2012;
GO
SELECT JobTitle, MAX(ph1.Rate)AS MaximumRate
FROM HumanResources.Employee AS e
JOIN HumanResources.EmployeePayHistory AS ph1 ON e.BusinessEntityID = ph1.BusinessEntityID
GROUP BY JobTitle
HAVING (MAX(CASE WHEN Gender = 'M'
        THEN ph1.Rate
        ELSE NULL END) > 40.00
     OR MAX(CASE WHEN Gender  = 'F'
        THEN ph1.Rate
        ELSE NULL END) > 42.00)
ORDER BY MaximumRate DESC;
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

G. Using a SELECT statement with a CASE expression

Within a SELECT statement, the CASE expression allows for values to be replaced in the result set based on comparison values. The following example uses the CASE expression to change the display of product line categories to make them more understandable. When a value does not exist, the text "Not for sale' is displayed.

```
-- Uses AdventureWorks

SELECT   ProductAlternateKey, Category =
      CASE ProductLine
         WHEN 'R' THEN 'Road'
         WHEN 'M' THEN 'Mountain'
         WHEN 'T' THEN 'Touring'
         WHEN 'S' THEN 'Other sale items'
         ELSE 'Not for sale'
      END,
   EnglishProductName
FROM dbo.DimProduct
ORDER BY ProductKey;
```

H. Using CASE in an UPDATE statement

The following example uses the CASE expression in an UPDATE statement to determine the value that is set for the column `VacationHours` for employees with `SalariedFlag` set to 0. When subtracting 10 hours from `VacationHours` results in a negative value, `VacationHours` is increased by 40 hours; otherwise, `VacationHours` is increased by 20 hours.

```
-- Uses AdventureWorks

UPDATE dbo.DimEmployee
SET VacationHours =
    ( CASE
         WHEN ((VacationHours - 10.00) < 0) THEN VacationHours + 40
         ELSE (VacationHours + 20.00)
       END
    )
WHERE SalariedFlag = 0;
```

See Also

Expressions (Transact-SQL)
SELECT (Transact-SQL)
COALESCE (Transact-SQL)

IIF (Transact-SQL)
CHOOSE (Transact-SQL)

COALESCE (Transact-SQL)

8/27/2018 • 6 minutes to read • Edit Online

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

Evaluates the arguments in order and returns the current value of the first expression that initially does not evaluate to `NULL`. For example, `SELECT COALESCE(NULL, NULL, 'third_value', 'fourth_value');` returns the third value because the third value is the first value that is not null.

Transact-SQL Syntax Conventions

Syntax

```
COALESCE ( expression [ ,...n ] )
```

Arguments

expression
Is an expression of any type.

Return Types

Returns the data type of *expression* with the highest data type precedence. If all expressions are nonnullable, the result is typed as nonnullable.

Remarks

If all arguments are `NULL`, `COALESCE` returns `NULL`. At least one of the null values must be a typed `NULL`.

Comparing COALESCE and CASE

The `COALESCE` expression is a syntactic shortcut for the `CASE` expression. That is, the code `COALESCE` (*expression1*,...*n*) is rewritten by the query optimizer as the following `CASE` expression:

```
CASE
WHEN (expression1 IS NOT NULL) THEN expression1
WHEN (expression2 IS NOT NULL) THEN expression2
...
ELSE expressionN
END
```

This means that the input values (*expression1*, *expression2*, *expressionN*, etc.) are evaluated multiple times. Also, in compliance with the SQL standard, a value expression that contains a subquery is considered non-deterministic and the subquery is evaluated twice. In either case, different results can be returned between the first evaluation and subsequent evaluations.

For example, when the code `COALESCE((subquery), 1)` is executed, the subquery is evaluated twice. As a result, you can get different results depending on the isolation level of the query. For example, the code can return `NULL` under the `READ COMMITTED` isolation level in a multi-user environment. To ensure stable results are returned, use

the `SNAPSHOT ISOLATION` isolation level, or replace `COALESCE` with the `ISNULL` function. Alternatively, you can rewrite the query to push the subquery into a subselect as shown in the following example:

```
SELECT CASE WHEN x IS NOT NULL THEN x ELSE 1 END
from
(
SELECT (SELECT Nullable FROM Demo WHERE SomeCol = 1) AS x
) AS T;
```

Comparing COALESCE and ISNULL

The `ISNULL` function and the `COALESCE` expression have a similar purpose but can behave differently.

1. Because `ISNULL` is a function, it is evaluated only once. As described above, the input values for the `COALESCE` expression can be evaluated multiple times.
2. Data type determination of the resulting expression is different. `ISNULL` uses the data type of the first parameter, `COALESCE` follows the `CASE` expression rules and returns the data type of value with the highest precedence.
3. The NULLability of the result expression is different for `ISNULL` and `COALESCE`. The `ISNULL` return value is always considered NOT NULLable (assuming the return value is a non-nullable one) whereas `COALESCE` with non-null parameters is considered to be `NULL`. So the expressions `ISNULL(NULL, 1)` and `COALESCE(NULL, 1)`, although equivalent, have different nullability values. This makes a difference if you are using these expressions in computed columns, creating key constraints or making the return value of a scalar UDF deterministic so that it can be indexed as shown in the following example:

   ```
   USE tempdb;
   GO
   -- This statement fails because the PRIMARY KEY cannot accept NULL values
   -- and the nullability of the COALESCE expression for col2
   -- evaluates to NULL.
   CREATE TABLE #Demo
   (
   col1 integer NULL,
   col2 AS COALESCE(col1, 0) PRIMARY KEY,
   col3 AS ISNULL(col1, 0)
   );

   -- This statement succeeds because the nullability of the
   -- ISNULL function evaluates AS NOT NULL.

   CREATE TABLE #Demo
   (
   col1 integer NULL,
   col2 AS COALESCE(col1, 0),
   col3 AS ISNULL(col1, 0) PRIMARY KEY
   );
   ```

4. Validations for `ISNULL` and `COALESCE` are also different. For example, a `NULL` value for `ISNULL` is converted to **int** whereas for `COALESCE`, you must provide a data type.
5. `ISNULL` takes only two parameters whereas `COALESCE` takes a variable number of parameters.

Examples

A. Running a simple example

The following example shows how `COALESCE` selects the data from the first column that has a nonnull value. This

example uses the AdventureWorks2012 database.

```
SELECT Name, Class, Color, ProductNumber,
COALESCE(Class, Color, ProductNumber) AS FirstNotNull
FROM Production.Product;
```

B. Running a complex example

In the following example, the `wages` table includes three columns that contain information about the yearly wages of the employees: the hourly wage, salary, and commission. However, an employee receives only one type of pay. To determine the total amount paid to all employees, use `COALESCE` to receive only the nonnull value found in `hourly_wage`, `salary`, and `commission`.

```
SET NOCOUNT ON;
GO
USE tempdb;
IF OBJECT_ID('dbo.wages') IS NOT NULL
    DROP TABLE wages;
GO
CREATE TABLE dbo.wages
(
    emp_id        tinyint   identity,
    hourly_wage   decimal   NULL,
    salary        decimal   NULL,
    commission    decimal   NULL,
    num_sales     tinyint   NULL
);
GO
INSERT dbo.wages (hourly_wage, salary, commission, num_sales)
VALUES
    (10.00, NULL, NULL, NULL),
    (20.00, NULL, NULL, NULL),
    (30.00, NULL, NULL, NULL),
    (40.00, NULL, NULL, NULL),
    (NULL, 10000.00, NULL, NULL),
    (NULL, 20000.00, NULL, NULL),
    (NULL, 30000.00, NULL, NULL),
    (NULL, 40000.00, NULL, NULL),
    (NULL, NULL, 15000, 3),
    (NULL, NULL, 25000, 2),
    (NULL, NULL, 20000, 6),
    (NULL, NULL, 14000, 4);
GO
SET NOCOUNT OFF;
GO
SELECT CAST(COALESCE(hourly_wage * 40 * 52,
   salary,
   commission * num_sales) AS money) AS 'Total Salary'
FROM dbo.wages
ORDER BY 'Total Salary';
GO
```

Here is the result set.

```
Total Salary
------------
10000.00
20000.00
20800.00
30000.00
40000.00
41600.00
45000.00
50000.00
56000.00
62400.00
83200.00
120000.00

(12 row(s) affected)
```

C: Simple Example

The following example demonstrates how `COALESCE` selects the data from the first column that has a non-null value. Assume for this example that the `Products` table contains this data:

```
Name         Color      ProductNumber
------------ ---------- -------------
Socks, Mens  NULL       PN1278
Socks, Mens  Blue       PN1965
NULL         White      PN9876
```

We then run the following COALESCE query:

```
SELECT Name, Color, ProductNumber, COALESCE(Color, ProductNumber) AS FirstNotNull
FROM Products ;
```

Here is the result set.

```
Name         Color      ProductNumber  FirstNotNull
------------ ---------- -------------  ------------
Socks, Mens  NULL       PN1278         PN1278
Socks, Mens  Blue       PN1965         Blue
NULL         White      PN9876         White
```

Notice that in the first row, the `FirstNotNull` value is `PN1278`, not `Socks, Mens`. This is because the `Name` column was not specified as a parameter for `COALESCE` in the example.

D: Complex Example

The following example uses `COALESCE` to compare the values in three columns and return only the non-null value found in the columns.

```
CREATE TABLE dbo.wages
(
    emp_id        tinyint   NULL,
    hourly_wage   decimal   NULL,
    salary        decimal   NULL,
    commission    decimal   NULL,
    num_sales     tinyint   NULL
);
INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (1, 10.00, NULL, NULL, NULL);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (2, 20.00, NULL, NULL, NULL);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (3, 30.00, NULL, NULL, NULL);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (4, 40.00, NULL, NULL, NULL);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (5, NULL, 10000.00, NULL, NULL);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (6, NULL, 20000.00, NULL, NULL);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (7, NULL, 30000.00, NULL, NULL);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (8, NULL, 40000.00, NULL, NULL);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (9, NULL, NULL, 15000, 3);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (10,NULL, NULL, 25000, 2);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (11, NULL, NULL, 20000, 6);

INSERT INTO dbo.wages (emp_id, hourly_wage, salary, commission, num_sales)
VALUES (12, NULL, NULL, 14000, 4);

SELECT CAST(COALESCE(hourly_wage * 40 * 52,
   salary,
   commission * num_sales) AS decimal(10,2)) AS TotalSalary
FROM dbo.wages
ORDER BY TotalSalary;
```

Here is the result set.

```
Total Salary
------------
10000.00
20000.00
20800.00
30000.00
40000.00
41600.00
45000.00
50000.00
56000.00
62400.00
83200.00
120000.00
```

See Also

ISNULL (Transact-SQL)
CASE (Transact-SQL)

NULLIF (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Returns a null value if the two specified expressions are equal. For example, `SELECT NULLIF(4,4) AS Same, NULLIF(5,7) AS Different;` returns NULL for the first column (4 and 4) because the two input values are the same. The second column returns the first value (5) because the two input values are different.

 Transact-SQL Syntax Conventions

Syntax

```
NULLIF ( expression , expression )
```

Arguments

expression
Is any valid scalar expression.

Return Types

Returns the same type as the first *expression*.

NULLIF returns the first *expression* if the two expressions are not equal. If the expressions are equal, NULLIF returns a null value of the type of the first *expression*.

Remarks

NULLIF is equivalent to a searched CASE expression in which the two expressions are equal and the resulting expression is NULL.

We recommend that you not use time-dependent functions, such as RAND(), within a NULLIF function. This could cause the function to be evaluated twice and to return different results from the two invocations.

Examples

A. Returning budget amounts that have not changed

The following example creates a `budgets` table to show a department (`dept`) its current budget (`current_year`) and its previous budget (`previous_year`). For the current year, `NULL` is used for departments with budgets that have not changed from the previous year, and `0` is used for budgets that have not yet been determined. To find out the average of only those departments that receive a budget and to include the budget value from the previous year (use the `previous_year` value, where the `current_year` is `NULL`), combine the `NULLIF` and `COALESCE` functions.

```
CREATE TABLE dbo.budgets
(
   dept            tinyint   IDENTITY,
   current_year      decimal   NULL,
   previous_year   decimal   NULL
);
INSERT budgets VALUES(100000, 150000);
INSERT budgets VALUES(NULL, 300000);
INSERT budgets VALUES(0, 100000);
INSERT budgets VALUES(NULL, 150000);
INSERT budgets VALUES(300000, 250000);
GO
SET NOCOUNT OFF;
SELECT AVG(NULLIF(COALESCE(current_year,
   previous_year), 0.00)) AS 'Average Budget'
FROM budgets;
GO
```

Here is the result set.

```
Average Budget
--------------
212500.000000
(1 row(s) affected)
```

B. Comparing NULLIF and CASE

To show the similarity between `NULLIF` and `CASE`, the following queries evaluate whether the values in the `MakeFlag` and `FinishedGoodsFlag` columns are the same. The first query uses `NULLIF`. The second query uses the `CASE` expression.

```
USE AdventureWorks2012;
GO
SELECT ProductID, MakeFlag, FinishedGoodsFlag,
   NULLIF(MakeFlag,FinishedGoodsFlag)AS 'Null if Equal'
FROM Production.Product
WHERE ProductID < 10;
GO

SELECT ProductID, MakeFlag, FinishedGoodsFlag,'Null if Equal' =
   CASE
       WHEN MakeFlag = FinishedGoodsFlag THEN NULL
       ELSE MakeFlag
   END
FROM Production.Product
WHERE ProductID < 10;
GO
```

C: Returning budget amounts that contain no data

The following example creates a `budgets` table, loads data, and uses `NULLIF` to return a null if neither `current_year` nor `previous_year` contains data.

```
CREATE TABLE budgets (
   dept           tinyint,
   current_year   decimal(10,2),
   previous_year  decimal(10,2)
);

INSERT INTO budgets VALUES(1, 100000, 150000);
INSERT INTO budgets VALUES(2, NULL, 300000);
INSERT INTO budgets VALUES(3, 0, 100000);
INSERT INTO budgets VALUES(4, NULL, 150000);
INSERT INTO budgets VALUES(5, 300000, 300000);

SELECT dept, NULLIF(current_year,
   previous_year) AS LastBudget
FROM budgets;
```

Here is the result set.

```
dept   LastBudget
----   -----------
1      100000.00
2      null
3      0.00
4      null
5      null
```

See Also

CASE (Transact-SQL)
decimal and numeric (Transact-SQL)
System Functions (Transact-SQL)

Operators (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

An operator is a symbol specifying an action that is performed on one or more expressions. The following tables lists the operator categories that SQL Server uses.

Arithmetic Operators	Logical Operators
Assignment Operator	Scope Resolution Operator
Bitwise Operators	Set Operators
Comparison Operators	String Concatenation Operator
Compound Operators	Unary Operators

See Also

Operator Precedence (Transact-SQL)

Unary Operators - Positive

6/20/2018 • 2 minutes to read • Edit Online

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

Returns the value of a numeric expression (a unary operator). Unary operators perform an operation on only one expression of any one of the data types of the numeric data type category.

OPERATOR	MEANING
+ (Positive)	Numeric value is positive.
- (Negative)	Numeric value is negative.
~ (Bitwise NOT)	Returns the ones complement of the number.

The + (Positive) and - (Negative) operators can be used on any expression of any one of the data types of the numeric data type category. The ~ (Bitwise NOT) operator can be used only on expressions of any one of the data types of the integer data type category.

Transact-SQL Syntax Conventions

Syntax

```
+ numeric_expression
```

Arguments

numeric_expression
Is any valid expression of any one of the data types in the numeric data type category, except the **datetime** and **smalldatetime** data types.

Result Types

Returns the data type of *numeric_expression*.

Remarks

Although a unary plus can appear before any numeric expression, it performs no operation on the value returned from the expression. Specifically, it will not return the positive value of a negative expression. To return positive value of a negative expression, use the ABS function.

Examples

A. Setting a variable to a positive value

The following example sets a variable to a positive value.

```
DECLARE @MyNumber decimal(10,2);
SET @MyNumber = +123.45;
SELECT @MyNumber;
GO
```

Here is the result set:

```
-----------
123.45

(1 row(s) affected)
```

B. Using the unary plus operator with a negative value

The following example shows using the unary plus with a negative expression and the ABS() function on the same negative expression. The unary plus does not affect the expression, but the ABS function returns the positive value of the expression.

```
USE tempdb;
GO
DECLARE @Num1 int;
SET @Num1 = -5;
SELECT +@Num1, ABS(@Num1);
GO
```

Here is the result set:

```
----------- -----------
-5          5

(1 row(s) affected)
```

See Also

Data Types (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)
ABS (Transact-SQL)

Unary Operators - Negative

8/27/2018 • 2 minutes to read • Edit Online

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

Returns the negative of the value of a numeric expression (a unary operator). Unary operators perform an operation on only one expression of any one of the data types of the numeric data type category.

OPERATOR	MEANING
+ (Positive)	Numeric value is positive.
- (Negative)	Numeric value is negative.
~ (Bitwise NOT)	Returns the ones complement of the number.

The + (Positive) and - (Negative) operators can be used on any expression of any one of the data types of the numeric data type category. The ~ (Bitwise NOT) operator can be used only on expressions of any one of the data types of the integer data type category.

Transact-SQL Syntax Conventions

Syntax

```
- numeric_expression
```

Arguments

numeric_expression
Is any valid expression of any one of the data types of the numeric data type category, except the date and time category.

Result Types

Returns the data type of *numeric_expression*, except that an unsigned **tinyint** expression is promoted to a signed **smallint** result.

Examples

A. Setting a variable to a negative value

The following example sets a variable to a negative value.

```
USE tempdb;
GO
DECLARE @MyNumber decimal(10,2);
SET @MyNumber = -123.45;
SELECT @MyNumber AS NegativeValue;
GO
```

Here is the result set.

```
NegativeValue
--------------------------------------
-123.45

(1 row(s) affected)
```

B. Changing a variable to a negative value

The following example changes a variable to a negative value.

```
USE tempdb;
GO
DECLARE @Num1 int;
SET @Num1 = 5;
SELECT @Num1 AS VariableValue, -@Num1 AS NegativeValue;
GO
```

Here is the result set.

```
VariableValue NegativeValue
------------- -------------
5             -5

(1 row(s) affected)
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

C. Returning the negative of a positive constant

The following example returns the negative of a positive constant.

```
USE ssawPDW;

SELECT TOP (1) - 17 FROM DimEmployee;
```

Returns

```
-17
```

D. Returning the positive of a negative constant

The following example returns the positive of a negative constant.

```
USE ssawPDW;

SELECT TOP (1) – ( - 17) FROM DimEmployee;
```

Returns

```
17
```

E. Returning the negative of a column

The following example returns the negative of the `BaseRate` value for each employee in the `dimEmployee` table.

```
USE ssawPDW;

SELECT - BaseRate FROM DimEmployee;
```

See Also

Data Types (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)

Set Operators - EXCEPT and INTERSECT (Transact-SQL)

8/27/2018 • 5 minutes to read • Edit Online

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

Returns distinct rows by comparing the results of two queries.

EXCEPT returns distinct rows from the left input query that aren't output by the right input query.

INTERSECT returns distinct rows that are output by both the left and right input queries operator.

The basic rules for combining the result sets of two queries that use EXCEPT or INTERSECT are the following:

- The number and the order of the columns must be the same in all queries.
- The data types must be compatible.

Transact-SQL Syntax Conventions

Syntax

```
{ <query_specification> | ( <query_expression> ) }
{ EXCEPT | INTERSECT }
{ <query_specification> | ( <query_expression> ) }
```

Arguments

<query_specification> | (*<query_expression>*)
Is a query specification or query expression that returns data to be compared with the data from another query specification or query expression. The definitions of the columns that are part of an EXCEPT or INTERSECT operation do not have to be the same, but they must be comparable through implicit conversion. When data types differ, the type that is used to perform the comparison and return results is determined based on the rules for data type precedence.

When the types are the same but differ in precision, scale, or length, the result is determined based on the same rules for combining expressions. For more information, see Precision, Scale, and Length (Transact-SQL).

The query specification or expression cannot return **xml**, **text**, **ntext**, **image**, or nonbinary CLR user-defined type columns because these data types are not comparable.

EXCEPT
Returns any distinct values from the query to the left of the EXCEPT operator that are not also returned from the right query.

INTERSECT
Returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operator.

Remarks

When the data types of comparable columns that are returned by the queries to the left and right of the EXCEPT or

INTERSECT operators are character data types with different collations, the required comparison is performed according to the rules of collation precedence. If this conversion cannot be performed, the SQL Server Database Engine returns an error.

When comparing column values for determining DISTINCT rows, two NULL values are considered equal.

The column names of the result set that are returned by EXCEPT or INTERSECT are the same names as those returned by the query on the left side of the operator.

Column names or aliases in ORDER BY clauses must reference column names returned by the left-side query.

The nullability of any column in the result set returned by EXCEPT or INTERSECT is the same as the nullability of the corresponding column that is returned by the query on the left side of the operator.

If EXCEPT or INTERSECT is used together with other operators in an expression, it is evaluated in the context of the following precedence:

1. Expressions in parentheses
2. The INTERSECT operator
3. EXCEPT and UNION evaluated from left to right based on their position in the expression

   If EXCEPT or INTERSECT is used to compare more than two sets of queries, data type conversion is determined by comparing two queries at a time, and following the previously mentioned rules of expression evaluation.

   EXCEPT and INTERSECT cannot be used in distributed partitioned view definitions, query notifications.

   EXCEPT and INTERSECT may be used in distributed queries, but are only executed on the local server and not pushed to the linked server. Therefore, using EXCEPT and INTERSECT in distributed queries may affect performance.

   Fast forward-only and static cursors are fully supported in the result set when they are used with an EXCEPT or INTERSECT operation. If a keyset-driven or dynamic cursor is used together with an EXCEPT or INTERSECT operation, the cursor of the result set of the operation is converted to a static cursor.

   When an EXCEPT operation is displayed by using the Graphical Showplan feature in SQL Server Management Studio, the operation appears as a left anti semi join, and an INTERSECT operation appears as a left semi join.

Examples

The following examples show using the `INTERSECT` and `EXCEPT` operators. The first query returns all values from the `Production.Product` table for comparison to the results with `INTERSECT` and `EXCEPT`.

```
-- Uses AdventureWorks

SELECT ProductID
FROM Production.Product ;
--Result: 504 Rows
```

The following query returns any distinct values that are returned by both the query on the left and right sides of the `INTERSECT` operator.

```
-- Uses AdventureWorks

SELECT ProductID
FROM Production.Product
INTERSECT
SELECT ProductID
FROM Production.WorkOrder ;
--Result: 238 Rows (products that have work orders)
```

The following query returns any distinct values from the query to the left of the `EXCEPT` operator that are not also found on the right query.

```
-- Uses AdventureWorks

SELECT ProductID
FROM Production.Product
EXCEPT
SELECT ProductID
FROM Production.WorkOrder ;
--Result: 266 Rows (products without work orders)
```

The following query returns any distinct values from the query to the left of the `EXCEPT` operator that are not also found on the right query. The tables are reversed from the previous example.

```
-- Uses AdventureWorks

SELECT ProductID
FROM Production.WorkOrder
EXCEPT
SELECT ProductID
FROM Production.Product ;
--Result: 0 Rows (work orders without products)
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

The following examples show how to use the `INTERSECT` and `EXCEPT` operators. The first query returns all values from the `FactInternetSales` table for comparison to the results with `INTERSECT` and `EXCEPT`.

```
-- Uses AdventureWorks

SELECT CustomerKey
FROM FactInternetSales;
--Result: 60398 Rows
```

The following query returns any distinct values that are returned by both the query on the left and right sides of the `INTERSECT` operator.

```
-- Uses AdventureWorks

SELECT CustomerKey
FROM FactInternetSales
INTERSECT
SELECT CustomerKey
FROM DimCustomer
WHERE DimCustomer.Gender = 'F'
ORDER BY CustomerKey;
--Result: 9133 Rows (Sales to customers that are female.)
```

The following query returns any distinct values from the query to the left of the `EXCEPT` operator that are not also found on the right query.

```
-- Uses AdventureWorks

SELECT CustomerKey
FROM FactInternetSales
EXCEPT
SELECT CustomerKey
FROM DimCustomer
WHERE DimCustomer.Gender = 'F'
ORDER BY CustomerKey;
--Result: 9351 Rows (Sales to customers that are not female.)
```

Set Operators - UNION (Transact-SQL)

8/27/2018 • 7 minutes to read • Edit Online

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

Combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union. The UNION operation is different from using joins that combine columns from two tables.

The following are basic rules for combining the result sets of two queries by using UNION:

- The number and the order of the columns must be the same in all queries.
- The data types must be compatible.

Transact-SQL Syntax Conventions

Syntax

```
    { <query_specification> | ( <query_expression> ) }
  UNION [ ALL ]
  <query_specification | ( <query_expression> )
[ UNION [ ALL ] <query_specification> | ( <query_expression> )
    [ ...n ] ]
```

Arguments

<query_specification> | (<query_expression>) Is a query specification or query expression that returns data to be combined with the data from another query specification or query expression. The definitions of the columns that are part of a UNION operation do not have to be the same, but they must be compatible through implicit conversion. When data types differ, the resulting data type is determined based on the rules for data type precedence. When the types are the same but differ in precision, scale, or length, the result is determined based on the same rules for combining expressions. For more information, see Precision, Scale, and Length (Transact-SQL).

Columns of the **xml** data type must be equivalent. All columns must be either typed to an XML schema or untyped. If typed, they must be typed to the same XML schema collection.

UNION
Specifies that multiple result sets are to be combined and returned as a single result set.

ALL
Incorporates all rows into the results. This includes duplicates. If not specified, duplicate rows are removed.

Examples

A. 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.

```
-- Uses AdventureWorks

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.
-- Uses AdventureWorks

SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE ProductModelID NOT IN (3, 4)
UNION
SELECT ProductModelID, Name
FROM dbo.Gloves
ORDER BY Name;
GO
```

B. 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.

```
-- Uses AdventureWorks

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

-- Uses AdventureWorks

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;
```

C. 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.

```
-- Uses AdventureWorks

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 */
-- Uses AdventureWorks

SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE ProductModelID NOT IN (3, 4)
ORDER BY Name
UNION
SELECT ProductModelID, Name
FROM dbo.Gloves;
GO

/* CORRECT */
-- Uses AdventureWorks

SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE ProductModelID NOT IN (3, 4)
UNION
SELECT ProductModelID, Name
FROM dbo.Gloves
ORDER BY Name;
GO
```

D. 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.

```
-- Uses AdventureWorks

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
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

E. Using a simple UNION

In the following example, the result set includes the contents of the `CustomerKey` columns of both the `FactInternetSales` and `DimCustomer` tables. Since the ALL keyword is not used, duplicates are excluded from the results.

```
-- Uses AdventureWorks

SELECT CustomerKey
FROM FactInternetSales
UNION
SELECT CustomerKey
FROM DimCustomer
ORDER BY CustomerKey;
```

F. Using UNION of two SELECT statements with ORDER BY

When any SELECT statement in a UNION statement includes an ORDER BY clause, that clause should be placed after all SELECT statements. The following example shows the incorrect and correct use of `UNION` in two `SELECT` statements in which a column is ordered with ORDER BY.

```
-- Uses AdventureWorks

-- INCORRECT
SELECT CustomerKey
FROM FactInternetSales
ORDER BY CustomerKey
UNION
SELECT CustomerKey
FROM DimCustomer
ORDER BY CustomerKey;

-- CORRECT
USE AdventureWorksPDW2012;

SELECT CustomerKey
FROM FactInternetSales
UNION
SELECT CustomerKey
FROM DimCustomer
ORDER BY CustomerKey;
```

G. Using UNION of two SELECT statements with WHERE and ORDER BY

The following example shows the incorrect and correct use of `UNION` in two `SELECT` statements where WHERE and ORDER BY are needed.

```
-- Uses AdventureWorks

-- INCORRECT
SELECT CustomerKey
FROM FactInternetSales
WHERE CustomerKey >= 11000
ORDER BY CustomerKey
UNION
SELECT CustomerKey
FROM DimCustomer
ORDER BY CustomerKey;

-- CORRECT
USE AdventureWorksPDW2012;

SELECT CustomerKey
FROM FactInternetSales
WHERE CustomerKey >= 11000
UNION
SELECT CustomerKey
FROM DimCustomer
ORDER BY CustomerKey;
```

H. Using UNION of three SELECT statements to show effects of ALL and parentheses

The following examples use `UNION` to combine the results of **the same table** in order to demonstrate the effects of ALL and parentheses when using `UNION`.

The first example uses `UNION ALL` to show duplicated records and returns each row in the source table three times. The second example uses `UNION` without `ALL` to eliminate the duplicate rows from the combined results of the three `SELECT` statements and returns only the unduplicated rows from the source table.

The third example uses `ALL` with the first `UNION` and parentheses enclosing the second `UNION` that is not using `ALL`. The second `UNION` is processed first because it is in parentheses. It returns only the unduplicated rows from the table because the `ALL` option is not used and duplicates are removed. These 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.

```
-- Uses AdventureWorks

SELECT CustomerKey, FirstName, LastName
FROM DimCustomer
UNION ALL
SELECT CustomerKey, FirstName, LastName
FROM DimCustomer
UNION ALL
SELECT CustomerKey, FirstName, LastName
FROM DimCustomer;

SELECT CustomerKey, FirstName, LastName
FROM DimCustomer
UNION
SELECT CustomerKey, FirstName, LastName
FROM DimCustomer
UNION
SELECT CustomerKey, FirstName, LastName
FROM DimCustomer;

SELECT CustomerKey, FirstName, LastName
FROM DimCustomer
UNION ALL
(
SELECT CustomerKey, FirstName, LastName
FROM DimCustomer
UNION
SELECT CustomerKey, FirstName, LastName
FROM DimCustomer
);
```

See Also

SELECT (Transact-SQL)
SELECT Examples (Transact-SQL)

Arithmetic Operators (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Arithmetic operators perform mathematical operations on two expressions of one or more of the data types of the numeric data type category. For more information about data type categories, see Transact-SQL Syntax Conventions.

OPERATOR	MEANING
+ (Add)	Addition
- (Subtract)	Subtraction
* (Multiply)	Multiplication
/ (Divide)	Division
% (Modulo)	Returns the integer remainder of a division. For example, 12 % 5 = 2 because the remainder of 12 divided by 5 is 2.

The plus (+) and minus (-) operators can also be used to perform arithmetic operations on **datetime** and **smalldatetime** values.

For more information about the precision and scale of the result of an arithmetic operation, see Precision, Scale, and Length (Transact-SQL).

See Also

Mathematical Functions (Transact-SQL)
Data Types (Transact-SQL)
Expressions (Transact-SQL)

+ (Addition) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Adds two numbers. This addition arithmetic operator can also add a number, in days, to a date.

Transact-SQL Syntax Conventions

Syntax

```
expression + expression
```

Arguments

expression
Is any valid expression of any one of the data types in the numeric category except the **bit** data type. Cannot be used with **date**, **time**, **datetime2**, or **datetimeoffset** data types.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Examples

A. Using the addition operator to calculate the total number of hours away from work for each employee.

This example finds the total number of hours away from work for each employee by adding the number of hours taken for vacation and the number of hours taken as sick leave.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName, VacationHours, SickLeaveHours,
    VacationHours + SickLeaveHours AS 'Total Hours Away'
FROM HumanResources.Employee AS e
    JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID
ORDER BY 'Total Hours Away' ASC;
GO
```

B. Using the addition operator to add days to date and time values

This example adds a number of days to a `datetime` date.

```
SET NOCOUNT ON
DECLARE @startdate datetime, @adddays int;
SET @startdate = 'January 10, 1900 12:00 AM';
SET @adddays = 5;
SET NOCOUNT OFF;
SELECT @startdate + 1.25 AS 'Start Date',
   @startdate + @adddays AS 'Add Date';
```

Here is the result set.

```
Start Date                  Add Date
--------------------------- ---------------------------
1900-01-11 06:00:00.000     1900-01-15 00:00:00.000

(1 row(s) affected)
```

C. Adding character and integer data types

The following example adds an **int** data type value and a character value by converting the character data type to **int**. If a character that is not valid exists in the **char** string, the Transact-SQL returns an error.

```
DECLARE @addvalue int;
SET @addvalue = 15;
SELECT '125127' + @addvalue;
```

Here is the result set.

```
-----------------------
125142

(1 row(s) affected)
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

D: Using the addition operator to calculate the total number of hours away from work for each employee

The following example finds the total number of hours away from work for each employee by adding the number of hours taken for vacation and the number of hours taken as sick leave and sorts the results in ascending order.

```
-- Uses AdventureWorks

SELECT FirstName, LastName, VacationHours, SickLeaveHours,
    VacationHours + SickLeaveHours AS TotalHoursAway
FROM DimEmployee
ORDER BY TotalHoursAway ASC;
```

See Also

Operators (Transact-SQL)
Compound Operators (Transact-SQL)
+= (Addition Assignment) (Transact-SQL)
CAST and CONVERT (Transact-SQL)
Data Type Conversion (Database Engine)
Data Types (Transact-SQL)

Built-in Functions (Transact-SQL)
SELECT (Transact-SQL)

+= (Addition Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Adds two numbers and sets a value to the result of the operation. For example, if a variable @x equals 35, then @x += 2 takes the original value of @x, add 2 and sets @x to that new value (37).

Transact-SQL Syntax Conventions

Syntax

```
expression += expression
```

Arguments

expression
Is any valid expression of any data type in the numeric category except the **bit** data type.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

For more information, see + (Addition) (Transact-SQL).

See Also

Compound Operators (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)
+= (String Concatenation Assignment) (Transact-SQL)

- (Subtraction) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Subtracts two numbers (an arithmetic subtraction operator). Can also subtract a number, in days, from a date.

Transact-SQL Syntax Conventions

Syntax

```
expression - expression
```

Arguments

expression
Is any valid expression of any one of the data types of the numeric data type category, except the **bit** data type. Cannot be used with **date**, **time**, **datetime2**, or **datetimeoffset** data types.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Examples

A. Using subtraction in a SELECT statement

The following example calculates the difference in tax rate between the state or province with the highest tax rate and the state or province with the lowest tax rate.

Applies to: SQL Server and SQL Database.

```
-- Uses AdventureWorks

SELECT MAX(TaxRate) - MIN(TaxRate) AS 'Tax Rate Difference'
FROM Sales.SalesTaxRate
WHERE StateProvinceID IS NOT NULL;
GO
```

You can change the order of execution by using parentheses. Calculations inside parentheses are evaluated first. If parentheses are nested, the most deeply nested calculation has precedence.

B. Using date subtraction

The following example subtracts a number of days from a `datetime` date.

Applies to: SQL Server and SQL Database.

```
-- Uses AdventureWorks

DECLARE @altstartdate datetime;
SET @altstartdate = CONVERT(DATETIME, ''January 10, 1900 3:00 AM', 101);
SELECT @altstartdate - 1.5 AS 'Subtract Date';
```

Here is the result set:

```
Subtract Date
----------------------
1900-01-08 15:00:00.000

(1 row(s) affected)
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

C: Using subtraction in a SELECT statement

The following example calculates the difference in a base rate between the employee with the highest base rate and the employee with the lowest tax rate, from the `dimEmployee` table.

```
-- Uses AdventureWorks

SELECT MAX(BaseRate) - MIN(BaseRate) AS BaseRateDifference
FROM DimEmployee;
```

See Also

-= (Subtraction Assignment) (Transact-SQL)
Compound Operators (Transact-SQL)
Arithmetic Operators (Transact-SQL)
- (Negative) (Transact-SQL)
Data Types (Transact-SQL)
Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
SELECT (Transact-SQL)

-= (Subtraction Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Subtracts two numbers and sets a value to the result of the operation. For example, if a variable @x equals 35, then @x -= 2 takes the original value of @x, subtracts 2 and sets @x to that new value (33).

Syntax

```
expression -= expression
```

Arguments

expression
Is any valid expression of any one of the data types in the numeric category except the **bit** data type.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

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

See Also

Compound Operators (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)

* (Multiplication) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Multiplies two expressions (an arithmetic multiplication operator).

Transact-SQL Syntax Conventions

Syntax

```
expression * expression
```

Arguments

expression
Is any valid expression of any one of the data types of the numeric data type category, except the **datetime** and **smalldatetime** data types.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Examples

The following example retrieves the product identification number, name, the list price and the new list price of all the mountain bicycles in the `Product` table. The new list price is calculated by using the `*` arithmetic operator to multiply `ListPrice` by `1.15`.

```
-- Uses AdventureWorks

SELECT ProductID, Name, ListPrice, ListPrice * 1.15 AS NewPrice
FROM Production.Product
WHERE Name LIKE 'Mountain-%'
ORDER BY ProductID ASC;
GO
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

The following example retrieves the first and last name of employees in the `dimEmployee` table, and calculates the pay for `VacationHours` for each..

```
-- Uses AdventureWorks

SELECT FirstName, LastName, BaseRate * VacationHours AS VacationPay
FROM DimEmployee
ORDER BY lastName ASC;
```

See Also

Data Types (Transact-SQL)
Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)
*= (Multiplication Assignment) (Transact-SQL)
Compound Operators (Transact-SQL)

*= (Multiplication Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Multiplies two numbers and sets a value to the result of the operation. For example, if a variable @x equals 35, then @x *= 2 takes the original value of @x, multiplies by 2 and sets @x to that new value (70).

 Transact-SQL Syntax Conventions

Syntax

```
expression *= expression
```

Arguments

expression
Is any valid expression of any one of the data types in the numeric category except the **bit** data type.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

For more information, see * (Multiplication) (Transact-SQL).

See Also

Compound Operators (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)

/ (Division) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Divides one number by another (an arithmetic division operator).

Transact-SQL Syntax Conventions

Syntax

```
dividend / divisor
```

Arguments

dividend
Is the numeric expression to divide. *dividend* can be any valid expression of any one of the data types of the numeric data type category, except the **datetime** and **smalldatetime** data types.

divisor
Is the numeric expression by which to divide the dividend. *divisor* can be any valid expression of any one of the data types of the numeric data type category, except the **datetime** and **smalldatetime** data types.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

If an integer *dividend* is divided by an integer *divisor*, the result is an integer that has any fractional part of the result truncated.

Remarks

The actual value returned by the / operator is the quotient of the first expression divided by the second expression.

Examples

The following example uses the division arithmetic operator to calculate the sales target per month for the sales people at Adventure Works Cycles.

```
-- Uses AdventureWorks

SELECT s.BusinessEntityID AS SalesPersonID, FirstName, LastName, SalesQuota, SalesQuota/12 AS 'Sales Target
Per Month'
FROM Sales.SalesPerson AS s
JOIN HumanResources.Employee AS e
    ON s.BusinessEntityID = e.BusinessEntityID
JOIN Person.Person AS p
    ON e.BusinessEntityID = p.BusinessEntityID;
```

Here is a partial result set.

```
SalesPersonID FirstName    LastName          SalesQuota  Sales Target Per Month
------------- ------------ ----------------- ----------- ------------------
274           Stephen      Jiang             NULL        NULL
275           Michael      Blythe            300000.00   25000.00
276           Linda        Mitchell          250000.00   20833.3333
277           Jillian      Carson            250000.00   20833.3333
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

The following example uses the division arithmetic operator to calculate a simple ratio of each employees' vacation hours to sick hours.

```
-- Uses AdventureWorks

SELECT FirstName, LastName, VacationHours/SickLeaveHours AS PersonalTimeRatio
FROM DimEmployee;
```

See Also

Data Types (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)
/= (Division Assignment) (Transact-SQL)
Compound Operators (Transact-SQL)

/= (Division Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Divides one number by another and sets a value to the result of the operation. For example, if a variable @x equals 34, then `@x /= 2` takes the original value of @x, divides by 2 and sets @x to that new value (17).

Transact-SQL Syntax Conventions

Syntax

```
expression /= expression
```

Arguments

expression
Is any valid expression of any one of the data types in the numeric category except the **bit** data type.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

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

Examples

The following example, sets a variable to 17. Then uses the `/=` operator to set the variable to half of it's original value.

```
DECLARE @myVariable decimal(5,2);
SET @myVariable = 17.5;
SET @myVariable /= 2;
SELECT @myVariable AS ResultVariable;
```

Here is the result set.

RESULTVARIABLE
8.75

See Also

Compound Operators (Transact-SQL)
Expressions (Transact-SQL)

Operators (Transact-SQL)

% (Modulus) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Returns the remainder of one number divided by another.

Transact-SQL Syntax Conventions

Syntax

```
dividend % divisor
```

Arguments

dividend
Is the numeric expression to divide. *dividend* must be a valid expression of any one of the data types in the integer and monetary data type categories, or the **numeric** data type.

divisor
Is the numeric expression by which to divide the dividend. *divisor* must be any valid expression of any one of the data types in the integer and monetary data type categories, or the **numeric** data type.

Result Types

Determined by data types of the two arguments.

Remarks

You can use the modulo arithmetic operator in the select list of the SELECT statement with any combination of column names, numeric constants, or any valid expression of the integer and monetary data type categories or the **numeric** data type.

Examples

A. Simple example

The following example divides the number 38 by 5. This results in 7 as the integer portion of the result and demonstrates how modulo returns the remainder of 3.

```
SELECT 38 / 5 AS Integer, 38 % 5 AS Remainder ;
```

B. Example using columns in a table

The following example returns the product ID number, the unit price of the product, and the modulo (remainder) of dividing the price of each product, converted to an integer value, into the number of products ordered.

```
-- Uses AdventureWorks

SELECT TOP(100)ProductID, UnitPrice, OrderQty,
   CAST((UnitPrice) AS int) % OrderQty AS Modulo
FROM Sales.SalesOrderDetail;
GO
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

C: Simple example

The following example shows results for the `%` operator when dividing 3 by 2.

```
-- Uses AdventureWorks

SELECT TOP(1) 3%2 FROM dimEmployee;
```

Here is the result set.

```
---------
1
```

See Also

Built-in Functions (Transact-SQL)
LIKE (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
%= (Modulus Assignment) (Transact-SQL)
Compound Operators (Transact-SQL)

%= (Modulus Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Divides one number by another and sets a value to the result of the operation. For example, if a variable @x equals 38, then @x %= 5 takes the original value of @x, divides by 5 and sets @x to the remainder of that division (3).

Transact-SQL Syntax Conventions

Syntax

```
expression %= expression
```

Arguments

expression
Is any valid expression of any one of the data types in the numeric category except the **bit** data type.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

For more information, see % (Modulus) (Transact-SQL).

See Also

Compound Operators (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)

= (Assignment Operator) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The equal sign (=) is the only Transact-SQL assignment operator. In the following example, the `@MyCounter` variable is created, and then the assignment operator sets `@MyCounter` to a value returned by an expression.

```
DECLARE @MyCounter INT;
SET @MyCounter = 1;
```

The assignment operator can also be used to establish the relationship between a column heading and the expression that defines the values for the column. The following example displays the column headings `FirstColumnHeading` and `SecondColumnHeading`. The string `xyz` is displayed in the `FirstColumnHeading` column heading for all rows. Then, each product ID from the `Product` table is listed in the `SecondColumnHeading` column heading.

```
-- Uses AdventureWorks

SELECT FirstColumnHeading = 'xyz',
       SecondColumnHeading = ProductID
FROM Production.Product;
GO
```

See Also

Operators (Transact-SQL)
Compound Operators (Transact-SQL)
Expressions (Transact-SQL)

Bitwise Operators (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Bitwise operators perform bit manipulations between two expressions of any of the data types of the integer data type category.
Bitwise operators convert two integer values to binary bits, perform the AND, OR, or NOT operation on each bit, producing a result. Then converts the result to an integer.

For example, the integer 170 converts to binary 1010 1010. The integer 75 converts to binary 0100 1011.

OPERATOR	BITWISE MATH
AND If bits at any location are both 1, the result is 1.	1010 1010 = 170 0100 1011 = 75 ---------- 0000 1010 = 10
OR If either bit at any location is 1, the result is 1.	1010 1010 = 170 0100 1011 = 75 ---------- 1110 1011 = 235
NOT Reverses the bit value at every bit location.	1010 1010 = 170 ---------- 0101 0101 = 85

See the following topics:

- & (Bitwise AND)
- &= (Bitwise AND Assignment)
- | (Bitwise OR)
- |= (Bitwise OR Assignment)
- ^ (Bitwise Exclusive OR)
- ^= (Bitwise Exclusive OR Assignment)
- ~ (Bitwise NOT)

The operands for bitwise operators can be any one of the data types of the integer or binary string data type categories (except for the **image** data type), except that both operands cannot be any one of the data types of the binary string data type category. The following table shows the supported operand data types.

LEFT OPERAND	RIGHT OPERAND
binary	**int**, **smallint**, or **tinyint**
bit	**int**, **smallint**, **tinyint**, or **bit**
int	**int**, **smallint**, **tinyint**, **binary**, or **varbinary**

LEFT OPERAND	RIGHT OPERAND
smallint	**int**, **smallint**, **tinyint**, **binary**, or **varbinary**
tinyint	**int**, **smallint**, **tinyint**, **binary**, or **varbinary**
varbinary	**int**, **smallint**, or **tinyint**

See Also

Operators (Transact-SQL)
Data Types (Transact-SQL)
Compound Operators (Transact-SQL)

& (Bitwise AND) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Performs a bitwise logical AND operation between two integer values.

Transact-SQL Syntax Conventions

Syntax

```
expression & expression
```

Arguments

expression
Is any valid expression of any of the data types of the integer data type category, or the **bit**, or the **binary** or **varbinary** data types. *expression* is treated as a binary number for the bitwise operation.

> **NOTE**
> In a bitwise operation, only one *expression* can be of either **binary** or **varbinary** data type.

Result Types

int if the input values are **int**.

smallint if the input values are **smallint**.

tinyint if the input values are **tinyint** or **bit**.

Remarks

The **&** bitwise operator performs a bitwise logical AND between the two expressions, taking each corresponding bit for both expressions. The bits in the result are set to 1 if and only if both bits (for the current bit being resolved) in the input expressions have a value of 1; otherwise, the bit in the result is set to 0.

If the left and right expressions have different integer data types (for example, the left *expression* is **smallint** and the right *expression* is **int**), the argument of the smaller data type is converted to the larger data type. In this case, the **smallint***expression* is converted to an **int**.

Examples

The following example creates a table using the **int** data type to store the values and inserts two values into one row.

```
CREATE TABLE bitwise
(
a_int_value int NOT NULL,
b_int_value int NOT NULL
);
GO
INSERT bitwise VALUES (170, 75);
GO
```

This query performs the bitwise AND between the `a_int_value` and `b_int_value` columns.

```
SELECT a_int_value & b_int_value
FROM bitwise;
GO
```

Here is the result set:

```
-----------
10

(1 row(s) affected)
```

The binary representation of 170 (`a_int_value` or `A`) is `0000 0000 1010 1010`. The binary representation of 75 (`b_int_value` or `B`) is `0000 0000 0100 1011`. Performing the bitwise AND operation on these two values produces the binary result `0000 0000 0000 1010`, which is decimal 10.

```
(A & B)
0000 0000 1010 1010
0000 0000 0100 1011
-------------------
0000 0000 0000 1010
```

See Also

Expressions (Transact-SQL)
Operators (Transact-SQL)
Bitwise Operators (Transact-SQL)
&= (Bitwise AND Assignment) (Transact-SQL)
Compound Operators (Transact-SQL)

&= (Bitwise AND Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Performs a bitwise logical AND operation between two integer values, and sets a value to the result of the operation.

Transact-SQL Syntax Conventions

Syntax

```
expression &= expression
```

Arguments

expression
Is any valid expression of any one of the data types in the numeric category except the **bit** data type.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

For more information, see & (Bitwise AND) (Transact-SQL).

See Also

Compound Operators (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)
Bitwise Operators (Transact-SQL)

| (Bitwise OR) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Performs a bitwise logical OR operation between two specified integer values as translated to binary expressions within Transact-SQL statements.

Transact-SQL Syntax Conventions

Syntax

```
expression | expression
```

Arguments

expression
Is any valid expression of the integer data type category, or the **bit**, **binary**, or **varbinary** data types. *expression* is treated as a binary number for the bitwise operation.

> **NOTE**
> Only one *expression* can be of either **binary** or **varbinary** data type in a bitwise operation.

Result Types

Returns an **int** if the input values are **int**, a **smallint** if the input values are **smallint**, or a **tinyint** if the input values are **tinyint**.

Remarks

The bitwise | operator performs a bitwise logical OR between the two expressions, taking each corresponding bit for both expressions. The bits in the result are set to 1 if either or both bits (for the current bit being resolved) in the input expressions have a value of 1; if neither bit in the input expressions is 1, the bit in the result is set to 0.

If the left and right expressions have different integer data types (for example, the left *expression* is **smallint** and the right *expression* is **int**), the argument of the smaller data type is converted to the larger data type. In this example, the **smallint***expression* is converted to an **int**.

Examples

The following example creates a table with **int** data types to show the original values and puts the table into one row.

```
CREATE TABLE bitwise
(
 a_int_value int NOT NULL,
b_int_value int NOT NULL
);
GO
INSERT bitwise VALUES (170, 75);
GO
```

The following query performs the bitwise OR on the **a_int_value** and **b_int_value** columns.

```
SELECT a_int_value | b_int_value
FROM bitwise;
GO
```

Here is the result set.

```
-----------
235

(1 row(s) affected)
```

The binary representation of 170 (**a_int_value** or `A`, below) is `0000 0000 1010 1010`. The binary representation of 75 (**b_int_value** or `B`, below) is `0000 0000 0100 1011`. Performing the bitwise OR operation on these two values produces the binary result `0000 0000 1110 1011`, which is decimal 235.

```
(A | B)
0000 0000 1010 1010
0000 0000 0100 1011
-------------------
0000 0000 1110 1011
```

See Also

Operators (Transact-SQL)
Bitwise Operators (Transact-SQL)
|= (Bitwise OR Assignment) (Transact-SQL)
Compound Operators (Transact-SQL)

|= (Bitwise OR Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Performs a bitwise logical OR operation between two specified integer values as translated to binary expressions within Transact-SQL statements, and sets a value to the result of the operation.

Transact-SQL Syntax Conventions

Syntax

```
expression |= expression
```

Arguments

expression
Is any valid expression of any one of the data types in the numeric category except the **bit** data type.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

For more information, see | (Bitwise OR) (Transact-SQL).

See Also

Compound Operators (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)
Bitwise Operators (Transact-SQL)

^ (Bitwise Exclusive OR) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Performs a bitwise exclusive OR operation between two integer values.

Transact-SQL Syntax Conventions

Syntax

```
expression ^ expression
```

Arguments

expression
Is any valid expression of any one of the data types of the integer data type category, or the **bit**, or the **binary** or **varbinary** data types. *expression* is treated as a binary number for the bitwise operation.

> **NOTE**
> Only one *expression* can be of either **binary** or **varbinary** data type in a bitwise operation.

Result Types

int if the input values are **int**.

smallint if the input values are **smallint**.

tinyint if the input values are **tinyint**.

Remarks

The **^** bitwise operator performs a bitwise logical exclusive OR between the two expressions, taking each corresponding bit for both expressions. The bits in the result are set to 1 if either (but not both) bits (for the current bit being resolved) in the input expressions have a value of 1. If both bits are 0 or both bits are 1, the bit in the result is cleared to a value of 0.

If the left and right expressions have different integer data types (for example, the left *expression* is **smallint** and the right *expression* is **int**), the argument of the smaller data type is converted to the larger data type. In this case, the **smallint***expression* is converted to an **int**.

Examples

The following example creates a table using the **int** data type to store the original values and inserts two values into one row.

```
CREATE TABLE bitwise
(
a_int_value int NOT NULL,
b_int_value int NOT NULL
);
GO
INSERT bitwise VALUES (170, 75);
GO
```

The following query performs the bitwise exclusive OR on the `a_int_value` and `b_int_value` columns.

```
SELECT a_int_value ^ b_int_value
FROM bitwise;
GO
```

Here is the result set:

```
-----------
225

(1 row(s) affected)
```

The binary representation of 170 (`a_int_value` or `A`) is `0000 0000 1010 1010`. The binary representation of 75 (`b_int_value` or `B`) is `0000 0000 0100 1011`. Performing the bitwise exclusive OR operation on these two values produces the binary result `0000 0000 1110 0001`, which is decimal 225.

```
(A ^ B)
         0000 0000 1010 1010
         0000 0000 0100 1011
         -------------------
         0000 0000 1110 0001
```

See Also

Expressions (Transact-SQL)
Operators (Transact-SQL)
Bitwise Operators (Transact-SQL)
^= (Bitwise Exclusive OR Assignment) (Transact-SQL)
Compound Operators (Transact-SQL)

^= (Bitwise Exclusive OR Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Performs a bitwise exclusive OR operation between two integer values, and sets a value to the result of the operation.

Transact-SQL Syntax Conventions

Syntax

```
expression ^= expression
```

Arguments

expression
Is any valid expression of any one of the data types in the numeric category except the **bit** data type.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

For more information, see ^ (Bitwise Exclusive OR) (Transact-SQL).

See Also

Compound Operators (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)
Bitwise Operators (Transact-SQL)

~ (Bitwise NOT) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Performs a bitwise logical NOT operation on an integer value.

Transact-SQL Syntax Conventions

Syntax

```
~ expression
```

Arguments

expression
Is any valid expression of any one of the data types of the integer data type category, the **bit**, or the **binary** or **varbinary** data types. *expression* is treated as a binary number for the bitwise operation.

> **NOTE**
> Only one *expression* can be of either **binary** or **varbinary** data type in a bitwise operation.

Result Types

int if the input values are **int**.

smallint if the input values are **smallint**.

tinyint if the input values are **tinyint**.

bit if the input values are **bit**.

Remarks

The **~** bitwise operator performs a bitwise logical NOT for the *expression*, taking each bit in turn. If *expression* has a value of 0, the bits in the result set are set to 1; otherwise, the bit in the result is cleared to a value of 0. In other words, ones are changed to zeros and zeros are changed to ones.

IMPORTANT

When you perform any kind of bitwise operation, the storage length of the expression used in the bitwise operation is important. We recommend that you use this same number of bytes when storing values. For example, storing the decimal value of 5 as a **tinyint**, **smallint**, or **int** produces a value stored with different numbers of bytes: **tinyint** stores data using 1 byte; **smallint** stores data using 2 bytes, and **int** stores data using 4 bytes. Therefore, performing a bitwise operation on an **int** decimal value can produce different results from those using a direct binary or hexadecimal translation, especially when the ~ (bitwise NOT) operator is used. The bitwise NOT operation may occur on a variable of a shorter length. In this case, when the shorter length is converted to a longer data type variable, the bits in the upper 8 bits may not be set to the expected value. We recommend that you convert the smaller data type variable to the larger data type, and then perform the NOT operation on the result.

Examples

The following example creates a table using the **int** data type to store the values and inserts the two values into one row.

```
CREATE TABLE bitwise
(
a_int_value int NOT NULL,
b_int_value int NOT NULL
);
GO
INSERT bitwise VALUES (170, 75);
GO
```

The following query performs the bitwise NOT on the `a_int_value` and `b_int_value` columns.

```
SELECT ~ a_int_value, ~ b_int_value
FROM bitwise;
```

Here is the result set:

```
--- ---
-171  -76

(1 row(s) affected)
```

The binary representation of 170 (`a_int_value` or `A`) is `0000 0000 1010 1010`. Performing the bitwise NOT operation on this value produces the binary result `1111 1111 0101 0101`, which is decimal -171. The binary representation for 75 is `0000 0000 0100 1011`. Performing the bitwise NOT operation produces `1111 1111 1011 0100`, which is decimal -76.

```
 (~A)
        0000 0000 1010 1010
        -------------------
        1111 1111 0101 0101
(~B)
        0000 0000 0100 1011
        -------------------
        1111 1111 1011 0100
```

See Also

Expressions (Transact-SQL)
Operators (Transact-SQL)
Bitwise Operators (Transact-SQL)

Comparison Operators (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Comparison operators test whether two expressions are the same. Comparison operators can be used on all expressions except expressions of the **text**, **ntext**, or **image** data types. The following table lists the Transact-SQL comparison operators.

OPERATOR	MEANING
= (Equals)	Equal to
> (Greater Than)	Greater than
< (Less Than)	Less than
>= (Greater Than or Equal To)	Greater than or equal to
<= (Less Than or Equal To)	Less than or equal to
<> (Not Equal To)	Not equal to
!= (Not Equal To)	Not equal to (not ISO standard)
!< (Not Less Than)	Not less than (not ISO standard)
!> (Not Greater Than)	Not greater than (not ISO standard)

Boolean Data Type

The result of a comparison operator has the **Boolean** data type. This has three values: TRUE, FALSE, and UNKNOWN. Expressions that return a **Boolean** data type are known as Boolean expressions.

Unlike other SQL Server data types, a **Boolean** data type cannot be specified as the data type of a table column or variable, and cannot be returned in a result set.

When SET ANSI_NULLS is ON, an operator that has one or two NULL expressions returns UNKNOWN. When SET ANSI_NULLS is OFF, the same rules apply, except for the equals (=) and not equals (<>) operators. When SET ANSI_NULLS is OFF, these operators treat NULL as a known value, equivalent to any other NULL, and only return TRUE or FALSE (never UNKNOWN).

Expressions with **Boolean** data types are used in the WHERE clause to filter the rows that qualify for the search conditions and in control-of-flow language statements such as IF and WHILE, for example:

```
-- Uses AdventureWorks

DECLARE @MyProduct int;
SET @MyProduct = 750;
IF (@MyProduct <> 0)
   SELECT ProductID, Name, ProductNumber
   FROM Production.Product
   WHERE ProductID = @MyProduct;
```

See Also

Expressions (Transact-SQL)
Operators (Transact-SQL)

= (Equals) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Compares the equality of two expressions (a comparison operator) in SQL Server.

Transact-SQL Syntax Conventions

Syntax

```
expression = expression
```

Arguments

expression
Is any valid expression. If the expressions are not of the same data type, the data type for one expression must be implicitly convertible to the data type of the other. The conversion is based on the rules of data type precedence.

Result Types

Boolean

Remarks

When you compare using a NULL expression, the result depends on the `ANSI_NULLS` setting:

- If `ANSI_NULLS` is set to ON, the result of any comparison with NULL is UNKNOWN, following the ANSI convention that NULL is an unknown value and cannot be compared with any other value, including other NULLs.
- If `ANSI_NULLS` is set to OFF, the result of comparing NULL to NULL is TRUE, and the result of comparing NULL to any other value is FALSE.

For more information, see SET ANSI_NULLS (Transact-SQL).

A boolean expression resulting in UNKNOWN behaves similarly to FALSE in most, but not all cases. See NULL and UNKNOWN (Transact-SQL) and NOT (Transact-SQL) for more information.

Examples

A. Using = in a simple query

The following example uses the Equals operator to return all rows in the `HumanResources.Department` table in which the value in the `GroupName` column is equal to the word 'Manufacturing'.

```
-- Uses AdventureWorks

SELECT DepartmentID, Name
FROM HumanResources.Department
WHERE GroupName = 'Manufacturing';
```

Here is the result set.

```
DepartmentID Name
------------ --------------------------------------------------
7            Production
8            Production Control

(2 row(s) affected)
```

B. Comparing NULL and non-NULL values

The following example uses the Equals (`=`) and Not Equal To (`<>`) comparison operators to make comparisons with `NULL` and nonnull values in a table. The example also shows that `IS NULL` is not affected by the `SET ANSI_NULLS` setting.

```
-- Create table t1 and insert 3 rows.
CREATE TABLE dbo.t1 (a INT NULL);
INSERT INTO dbo.t1 VALUES (NULL),(0),(1);
GO

-- Print message and perform SELECT statements.
PRINT 'Testing default setting';
DECLARE @varname int;
SET @varname = NULL;

SELECT a
FROM t1
WHERE a = @varname;

SELECT a
FROM t1
WHERE a <> @varname;

SELECT a
FROM t1
WHERE a IS NULL;
GO

-- SET ANSI_NULLS to ON and test.
PRINT 'Testing ANSI_NULLS ON';
SET ANSI_NULLS ON;
GO
DECLARE @varname int;
SET @varname = NULL

SELECT a
FROM t1
WHERE a = @varname;

SELECT a
FROM t1
WHERE a <> @varname;

SELECT a
FROM t1
WHERE a IS NULL;
GO

-- SET ANSI_NULLS to OFF and test.
PRINT 'Testing SET ANSI_NULLS OFF';
SET ANSI_NULLS OFF;
GO
DECLARE @varname int;
SET @varname = NULL;
SELECT a
FROM t1
WHERE a = @varname;

SELECT a
FROM t1
WHERE a <> @varname;

SELECT a
FROM t1
WHERE a IS NULL;
GO

-- Drop table t1.
DROP TABLE dbo.t1;
```

Here is the result set.

```
Testing default setting
a
-----------
NULL

(1 row(s) affected)

a
-----------
0
1

(2 row(s) affected)

a
-----------
NULL

(1 row(s) affected)

Testing ANSI_NULLS ON
a
-----------

(0 row(s) affected)

a
-----------

(0 row(s) affected)

a
-----------
NULL

(1 row(s) affected)

Testing SET ANSI_NULLS OFF
a
-----------
NULL

(1 row(s) affected)

a
-----------
0
1

(2 row(s) affected)

a
-----------
NULL

(1 row(s) affected)
```

See Also

Data Types (Transact-SQL)
Expressions (Transact-SQL)
Operators (Transact-SQL)

> (Greater Than) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Compares two expressions (a comparison operator) in SQL Server 2017. When you compare nonnull expressions, the result is TRUE if the left operand has a value higher than the right operand; otherwise, the result is FALSE. If either or both operands are NULL, see the topic SET ANSI_NULLS (Transact-SQL).

Transact-SQL Syntax Conventions

Syntax

```
expression > expression
```

Arguments

expression
Is any valid expression. Both expressions must have implicitly convertible data types. The conversion depends on the rules of data type precedence.

Result Types

Boolean

Examples

A. Using > in a simple query

The following example returns all rows in the `HumanResources.Department` table that have a value in `DepartmentID` that is greater than the value 13.

```
--Uses AdventureWorks

SELECT DepartmentID, Name
FROM HumanResources.Department
WHERE DepartmentID > 13
ORDER BY DepartmentID;
```

Here is the result set.

```
DepartmentID Name
------------ --------------------------------------------------
14           Facilities and Maintenance
15           Shipping and Receiving
16           Executive

(3 row(s) affected)
```

B. Using > to compare two variables

```
DECLARE @a int = 45, @b int = 40;
SELECT IIF ( @a > @b, 'TRUE', 'FALSE' ) AS Result;
```

Here is the result set.

```
Result
------
TRUE

(1 row(s) affected)
```

See Also

IIF (Transact-SQL)
Data Types (Transact-SQL)
Operators (Transact-SQL)

< (Less Than) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand has a value lower than the right operand; otherwise, the result is FALSE. If either or both operands are NULL, see the topic SET ANSI_NULLS (Transact-SQL).

Transact-SQL Syntax Conventions

Syntax

```
expression < expression
```

Arguments

expression
Is any valid expression. Both expressions must have implicitly convertible data types. The conversion depends on the rules of data type precedence.

Result Types

Boolean

Examples

A. Using < in a simple query

The following example returns all rows in the `HumanResources.Department` table that have a value in `DepartmentID` that is less than the value 3.

```
-- Uses AdventureWorks

SELECT DepartmentID, Name
FROM HumanResources.Department
WHERE DepartmentID < 3
ORDER BY DepartmentID;
```

Here is the result set.

```
DepartmentID Name
------------ --------------------------------------------------
1            Engineering
2            Tool Design

(2 row(s) affected)
```

B. Using < to compare two variables

```
DECLARE @a int = 45, @b int = 40;
SELECT IIF ( @a < @b, 'TRUE', 'FALSE' ) AS Result;
```

Here is the result set.

```
Result
------
FALSE

(1 row(s) affected)
```

See Also

IIF (Transact-SQL)
Data Types (Transact-SQL)
Operators (Transact-SQL)

>= (Greater Than or Equal To) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Compares two expressions for greater than or equal (a comparison operator).

Transact-SQL Syntax Conventions

Syntax

```
expression >= expression
```

Arguments

expression
Is any valid expression. Both expressions must have implicitly convertible data types. The conversion depends on the rules of data type precedence.

Result Types

Boolean

Remarks

When you compare nonnull expressions, the result is TRUE if the left operand has a greater or equal value than the right operand; otherwise, the result is FALSE.

Unlike the = (equality) comparison operator, the result of the >= comparison of two NULL values does not depend on the ANSI_NULLS setting.

Examples

A. Using >= in a simple query

The following example returns all rows in the `HumanResources.Department` table that have a value in `DepartmentID` that is greater than or equal to the value 13.

```
-- Uses AdventureWorks

SELECT DepartmentID, Name
FROM HumanResources.Department
WHERE DepartmentID >= 13
ORDER BY DepartmentID;
```

Here is the result set.

```
DepartmentID Name
------------ --------------------------------------------------
13           Quality Assurance
14           Facilities and Maintenance
15           Shipping and Receiving
16           Executive

(4 row(s) affected)
```

See Also

Data Types (Transact-SQL)
Expressions (Transact-SQL)
= (Equals) (Transact-SQL)
> (Greater Than) (Transact-SQL)
Operators (Transact-SQL)

<= (Less Than or Equal To) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand has a value lower than or equal to the right operand; otherwise, the result is FALSE.

Unlike the = (equality) comparison operator, the result of the >= comparison of two NULL values does not depend on the ANSI_NULLS setting.

Transact-SQL Syntax Conventions

Syntax

```
expression <= expression
```

Arguments

expression
Is any valid expression. Both expressions must have implicitly convertible data types. The conversion depends on the rules of data type precedence.

Result Types

Boolean

Examples

A. Using <= in a simple query

The following example returns all rows in the `HumanResources.Department` table that have a value in `DepartmentID` that is less than or equal to the value 3.

```
-- Uses AdventureWorks

SELECT DepartmentID, Name
FROM HumanResources.Department
WHERE DepartmentID <= 3
ORDER BY DepartmentID;
```

Here is the result set.

```
DepartmentID Name
------------ --------------------------------------------------
1            Engineering
2            Tool Design
3            Sales

(3 row(s) affected)
```

See Also

Data Types (Transact-SQL)
Operators (Transact-SQL)

Not Equal To (Transact SQL) - traditional

8/27/2018 • 2 minutes to read • Edit Online

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

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand is not equal to the right operand; otherwise, the result is FALSE. If either or both operands are NULL, see the topic SET ANSI_NULLS (Transact-SQL).

Transact-SQL Syntax Conventions

Syntax

```
expression <> expression
```

Arguments

expression
Is any valid expression. Both expressions must have implicitly convertible data types. The conversion depends on the rules of data type precedence.

Result Types

Boolean

Examples

A. Using <> in a simple query

The following example returns all rows in the `Production.ProductCategory` table that do not have value in `ProductCategoryID` that is equal to the value 3 or the value 2.

```
-- Uses AdventureWorks

SELECT ProductCategoryID, Name
FROM Production.ProductCategory
WHERE ProductCategoryID <> 3 AND ProductCategoryID <> 2;
```

Here is the result set.

```
ProductCategoryID Name
----------------- --------------------------------------------------
1                 Bikes
4                 Accessories

(2 row(s) affected)
```

See Also

Data Types (Transact-SQL)

Operators (Transact-SQL)
Comparison Operators (Transact-SQL)

!< (Not Less Than) (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand does not have a value lower than the right operand; otherwise, the result is FALSE. If either or both operands are NULL, see the topic SET ANSI_NULLS (Transact-SQL).

Transact-SQL Syntax Conventions

Syntax

```
expression !< expression
```

Arguments

expression
Is any valid expression. Both expressions must have implicitly convertible data types. The conversion depends on the rules of data type precedence.

Result Types

Boolean

See Also

Data Types (Transact-SQL)
Operators (Transact-SQL)

Not Equal To (Transact SQL) - exclamation

8/27/2018 • 2 minutes to read • Edit Online

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

Tests whether one expression is not equal to another expression (a comparison operator). If either or both operands are NULL, NULL is returned. Functions the same as the <> (Not Equal To) comparison operator.

See Also

Expressions (Transact-SQL)
Operators (Transact-SQL)

!> (Not Greater Than) (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand does not have a greater value than the right operand; otherwise, the result is FALSE. Unlike the = (equality) comparison operator, the result of the !> comparison of two NULL values does not depend on the ANSI_NULLS setting.

Transact-SQL Syntax Conventions

Syntax

```
expression !> expression
```

Arguments

expression
Is any valid expression. Both expressions must have implicitly convertible data types. The conversion depends on the rules of data type precedence.

Result Types

Boolean

See Also

Data Types (Transact-SQL)
Operators (Transact-SQL)

Compound Operators (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Compound operators execute some operation and set an original value to the result of the operation. For example, if a variable @x equals 35, then @x += 2 takes the original value of @x, add 2 and sets @x to that new value (37).

Transact-SQL provides the following compound operators:

OPERATOR	LINK TO MORE INFORMATION	ACTION
+=	+= (Add Assignment) (Transact-SQL)	Adds some amount to the original value and sets the original value to the result.
-=	-= (Subtract Assignment) (Transact-SQL)	Subtracts some amount from the original value and sets the original value to the result.
*=	*= (Multiply Assignment) (Transact-SQL)	Multiplies by an amount and sets the original value to the result.
/=	(Divide Assignment) (Transact-SQL)	Divides by an amount and sets the original value to the result.
%=	Modulus Assignment (Transact-SQL)	Divides by an amount and sets the original value to the modulo.
&=	&= (Bitwise AND Assignment) (Transact-SQL)	Performs a bitwise AND and sets the original value to the result.
^=	^= (Bitwise Exclusive OR Assignment) (Transact-SQL)	Performs a bitwise exclusive OR and sets the original value to the result.
|=	|= (Bitwise OR Assignment) (Transact-SQL)	Performs a bitwise OR and sets the original value to the result.

Syntax

```
expression operator expression
```

Arguments

expression

Is any valid expression of any one of the data types in the numeric category.

Result Types

Returns the data type of the argument with the higher precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

For more information, see the topics related to each operator.

Examples

The following examples demonstrate compound operations.

```
DECLARE @x1 int = 27;
SET @x1 += 2 ;
SELECT @x1 AS Added_2;

DECLARE @x2 int = 27;
SET @x2 -= 2 ;
SELECT @x2 AS Subtracted_2;

DECLARE @x3 int = 27;
SET @x3 *= 2 ;
SELECT @x3 AS Multiplied_by_2;

DECLARE @x4 int = 27;
SET @x4 /= 2 ;
SELECT @x4 AS Divided_by_2;

DECLARE @x5 int = 27;
SET @x5 %= 2 ;
SELECT @x5 AS Modulo_of_27_divided_by_2;

DECLARE @x6 int = 9;
SET @x6 &= 13 ;
SELECT @x6 AS Bitwise_AND;

DECLARE @x7 int = 27;
SET @x7 ^= 2 ;
SELECT @x7 AS Bitwise_Exclusive_OR;

DECLARE @x8 int = 27;
SET @x8 |= 2 ;
SELECT @x8 AS Bitwise_OR;
```

See Also

Operators (Transact-SQL)
Bitwise Operators (Transact-SQL)

Logical Operators (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Logical operators test for the truth of some condition. Logical operators, like comparison operators, return a **Boolean** data type with a value of TRUE, FALSE, or UNKNOWN.

OPERATOR	MEANING
ALL	TRUE if all of a set of comparisons are TRUE.
AND	TRUE if both Boolean expressions are TRUE.
ANY	TRUE if any one of a set of comparisons are TRUE.
BETWEEN	TRUE if the operand is within a range.
EXISTS	TRUE if a subquery contains any rows.
IN	TRUE if the operand is equal to one of a list of expressions.
LIKE	TRUE if the operand matches a pattern.
NOT	Reverses the value of any other Boolean operator.
OR	TRUE if either Boolean expression is TRUE.
SOME	TRUE if some of a set of comparisons are TRUE.

See Also

Operator Precedence (Transact-SQL)

ALL (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Compares a scalar value with a single-column set of values.

Transact-SQL Syntax Conventions

Syntax

```
scalar_expression { = | <> | != | > | >= | !> | < | <= | !< } ALL ( subquery )
```

Arguments

scalar_expression
Is any valid expression.

{ = | <> | != | > | >= | !> | < | <= | !< }
Is a comparison operator.

subquery
Is a subquery that returns a result set of one column. The data type of the returned column must be the same data type as the data type of *scalar_expression*.

Is a restricted SELECT statement, in which the ORDER BY clause and the INTO keyword are not allowed.

Result Types

Boolean

Result Value

Returns TRUE when the comparison specified is TRUE for all pairs (*scalar_expression*,*x*), when *x* is a value in the single-column set; otherwise returns FALSE.

Remarks

ALL requires the *scalar_expression* to compare positively to every value that is returned by the subquery. For instance, if the subquery returns values of 2 and 3, *scalar_expression* <= ALL (subquery) would evaluate as TRUE for a *scalar_expression* of 2. If the subquery returns values of 2 and 3, *scalar_expression* = ALL (subquery) would evaluate as FALSE, because some of the values of the subquery (the value of 3) would not meet the criteria of the expression.

For statements that require the *scalar_expression* to compare positively to only one value that is returned by the subquery, see SOME | ANY (Transact-SQL).

This topic refers to ALL when it is used with a subquery. ALL can also be used with UNION and SELECT.

Examples

The following example creates a stored procedure that determines whether all the components of a specified `SalesOrderID` in the AdventureWorks2012 database can be manufactured in the specified number of days. The example uses a subquery to create a list of the number of `DaysToManufacture` value for all of the components of the specific `SalesOrderID`, and then confirms that all the `DaysToManufacture` are within the number of days specified.

```
-- Uses AdventureWorks

CREATE PROCEDURE DaysToBuild @OrderID int, @NumberOfDays int
AS
IF
@NumberOfDays >= ALL
   (
    SELECT DaysToManufacture
    FROM Sales.SalesOrderDetail
    JOIN Production.Product
    ON Sales.SalesOrderDetail.ProductID = Production.Product.ProductID
    WHERE SalesOrderID = @OrderID
   )
PRINT 'All items for this order can be manufactured in specified number of days or less.'
ELSE
PRINT 'Some items for this order cannot be manufactured in specified number of days or less.' ;
```

To test the procedure, execute the procedure by using the `SalesOrderID 49080`, which has one component requiring `2` days and two components that require 0 days. The first statement below meets the criteria. The second query does not.

```
EXECUTE DaysToBuild 49080, 2 ;
```

Here is the result set.

```
All items for this order can be manufactured in specified number of days or less.
```

```
EXECUTE DaysToBuild 49080, 1 ;
```

Here is the result set.

```
Some items for this order cannot be manufactured in specified number of days or less.
```

See Also

CASE (Transact-SQL)
Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
LIKE (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)
IN (Transact-SQL)

AND (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Combines two Boolean expressions and returns **TRUE** when both expressions are **TRUE**. When more than one logical operator is used in a statement, the **AND** operators are evaluated first. You can change the order of evaluation by using parentheses.

Transact-SQL Syntax Conventions

Syntax

```
boolean_expression AND boolean_expression
```

Arguments

boolean_expression
Is any valid expression that returns a Boolean value: **TRUE**, **FALSE**, or **UNKNOWN**.

Result Types

Boolean

Result Value

Returns TRUE when both expressions are TRUE.

Remarks

The following chart shows the outcomes when you compare TRUE and FALSE values by using the AND operator.

	TRUE	**FALSE**	**UNKNOWN**
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Examples

A. Using the AND operator

The following example selects information about employees who have both the title of `Marketing Assistant` and more than `41` vacation hours available.

```
-- Uses AdventureWorks

SELECT  BusinessEntityID, LoginID, JobTitle, VacationHours
FROM HumanResources.Employee
WHERE JobTitle = 'Marketing Assistant'
AND VacationHours > 41 ;
```

B. Using the AND operator in an IF statement

The following examples show how to use AND in an IF statement. In the first statement, both `1 = 1` and `2 = 2` are true; therefore, the result is true. In the second example, the argument `2 = 17` is false; therefore, the result is false.

```
IF 1 = 1 AND 2 = 2
BEGIN
   PRINT 'First Example is TRUE'
END
ELSE PRINT 'First Example is FALSE';
GO

IF 1 = 1 AND 2 = 17
BEGIN
   PRINT 'Second Example is TRUE'
END
ELSE PRINT 'Second Example is FALSE' ;
GO
```

See Also

Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)

ANY (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Compares a scalar value with a single-column set of values. For more information, see SOME | ANY (Transact-SQL).

BETWEEN (Transact-SQL)

8/27/2018 • 3 minutes to read • Edit Online

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

Specifies a range to test.

Transact-SQL Syntax Conventions

Syntax

```
test_expression [ NOT ] BETWEEN begin_expression AND end_expression
```

Arguments

test_expression
Is the expression to test for in the range defined by *begin_expression*and *end_expression*. *test_expression* must be the same data type as both *begin_expression* and *end_expression*.

NOT
Specifies that the result of the predicate be negated.

begin_expression
Is any valid expression. *begin_expression* must be the same data type as both *test_expression* and *end_expression*.

end_expression
Is any valid expression. *end_expression* must be the same data type as both *test_expression*and *begin_expression*.

AND
Acts as a placeholder that indicates *test_expression* should be within the range indicated by *begin_expression* and *end_expression*.

Result Types

Boolean

Result Value

BETWEEN returns **TRUE** if the value of *test_expression* is greater than or equal to the value of *begin_expression* and less than or equal to the value of *end_expression*.

NOT BETWEEN returns **TRUE** if the value of *test_expression* is less than the value of *begin_expression* or greater than the value of *end_expression*.

Remarks

To specify an exclusive range, use the greater than (>) and less than operators (<). If any input to the BETWEEN or NOT BETWEEN predicate is NULL, the result is UNKNOWN.

Examples

A. Using BETWEEN

The following example returns information about the database roles in a database. The first query returns all the roles. The second example uses the `BETWEEN` clause to limit the roles to the specified `database_id` values.

```
SELECT principal_id, name
FROM sys.database_principals
WHERE type = 'R';

SELECT principal_id, name
FROM sys.database_principals
WHERE type = 'R'
AND principal_id BETWEEN 16385 AND 16390;
GO
```

Here is the result set.

```
principal_id    name
------------  ----
0               public
16384           db_owner
16385           db_accessadmin
16386           db_securityadmin
16387           db_ddladmin
16389           db_backupoperator
16390           db_datareader
16391           db_datawriter
16392           db_denydatareader
16393           db_denydatawriter
```

```
principal_id    name
------------  ----
16385           db_accessadmin
16386           db_securityadmin
16387           db_ddladmin
16389           db_backupoperator
16390           db_datareader
```

B. Using > and < instead of BETWEEN

The following example uses greater than (`>`) and less than (`<`) operators and, because these operators are not inclusive, returns nine rows instead of ten that were returned in the previous example.

```
-- Uses AdventureWorks

SELECT e.FirstName, e.LastName, ep.Rate
FROM HumanResources.vEmployee e
JOIN HumanResources.EmployeePayHistory ep
    ON e.BusinessEntityID = ep.BusinessEntityID
WHERE ep.Rate > 27 AND ep.Rate < 30
ORDER BY ep.Rate;
GO
```

Here is the result set.

```
FirstName   LastName            Rate
---------   ------------------  ---------
Paula       Barreto de Mattos   27.1394
Janaina     Bueno               27.4038
Dan         Bacon               27.4038
Ramesh      Meyyappan           27.4038
Karen       Berg                27.4038
David       Bradley             28.7500
Hazem       Abolrous            28.8462
Ovidiu      Cracium             28.8462
Rob         Walters             29.8462
```

C. Using NOT BETWEEN

The following example finds all rows outside a specified range of `27` through `30`.

```
-- Uses AdventureWorks

SELECT e.FirstName, e.LastName, ep.Rate
FROM HumanResources.vEmployee e
JOIN HumanResources.EmployeePayHistory ep
    ON e.BusinessEntityID = ep.BusinessEntityID
WHERE ep.Rate NOT BETWEEN 27 AND 30
ORDER BY ep.Rate;
GO
```

D. Using BETWEEN with datetime values

The following example retrieves rows in which **datetime** values are between `'20011212'` and `'20020105'`, inclusive.

```
-- Uses AdventureWorks

SELECT BusinessEntityID, RateChangeDate
FROM HumanResources.EmployeePayHistory
WHERE RateChangeDate BETWEEN '20011212' AND '20020105';
```

Here is the result set.

```
BusinessEntityID RateChangeDate
---------------- ----------------------
3                2001-12-12 00:00:00.000
4                2002-01-05 00:00:00.000
```

The query retrieves the expected rows because the date values in the query and the **datetime** values stored in the `RateChangeDate` column have been specified without the time part of the date. When the time part is unspecified, it defaults to 12:00 A.M. Note that a row that contains a time part that is after 12:00 A.M. on 2002-01-05 would not be returned by this query because it falls outside the range.

See Also

> (Greater Than) (Transact-SQL)
< (Less Than) (Transact-SQL)
Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)

EXISTS (Transact-SQL)

8/27/2018 • 5 minutes to read • Edit Online

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

Specifies a subquery to test for the existence of rows.

 Transact-SQL Syntax Conventions

Syntax

```
EXISTS ( subquery )
```

Arguments

subquery
Is a restricted SELECT statement. The INTO keyword is not allowed. For more information, see the information about subqueries in SELECT (Transact-SQL).

Result Types

Boolean

Result Values

Returns TRUE if a subquery contains any rows.

Examples

A. Using NULL in a subquery to still return a result set

The following example returns a result set with `NULL` specified in the subquery and still evaluates to TRUE by using `EXISTS`.

```
-- Uses AdventureWorks

SELECT DepartmentID, Name
FROM HumanResources.Department
WHERE EXISTS (SELECT NULL)
ORDER BY Name ASC ;
```

B. Comparing queries by using EXISTS and IN

The following example compares two queries that are semantically equivalent. The first query uses `EXISTS` and the second query uses `IN`.

```
-- Uses AdventureWorks

SELECT a.FirstName, a.LastName
FROM Person.Person AS a
WHERE EXISTS
(SELECT *
    FROM HumanResources.Employee AS b
    WHERE a.BusinessEntityID = b.BusinessEntityID
    AND a.LastName = 'Johnson');
GO
```

The following query uses `IN`.

```
-- Uses AdventureWorks

SELECT a.FirstName, a.LastName
FROM Person.Person AS a
WHERE a.LastName IN
(SELECT a.LastName
    FROM HumanResources.Employee AS b
    WHERE a.BusinessEntityID = b.BusinessEntityID
    AND a.LastName = 'Johnson');
GO
```

Here is the result set for either query.

```
FirstName                                          LastName
-------------------------------------------------- ----------
Barry                                              Johnson
David                                              Johnson
Willis                                             Johnson

(3 row(s) affected)
```

C. Comparing queries by using EXISTS and = ANY

The following example shows two queries to find stores whose name is the same name as a vendor. The first query uses `EXISTS` and the second uses `=``ANY`.

```
-- Uses AdventureWorks

SELECT DISTINCT s.Name
FROM Sales.Store AS s
WHERE EXISTS
(SELECT *
    FROM Purchasing.Vendor AS v
    WHERE s.Name = v.Name) ;
GO
```

The following query uses `= ANY`.

```
-- Uses AdventureWorks

SELECT DISTINCT s.Name
FROM Sales.Store AS s
WHERE s.Name = ANY
(SELECT v.Name
    FROM Purchasing.Vendor AS v ) ;
GO
```

D. Comparing queries by using EXISTS and IN

The following example shows queries to find employees of departments that start with `P`.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName, e.JobTitle
FROM Person.Person AS p
JOIN HumanResources.Employee AS e
   ON e.BusinessEntityID = p.BusinessEntityID
WHERE EXISTS
(SELECT *
    FROM HumanResources.Department AS d
    JOIN HumanResources.EmployeeDepartmentHistory AS edh
       ON d.DepartmentID = edh.DepartmentID
    WHERE e.BusinessEntityID = edh.BusinessEntityID
    AND d.Name LIKE 'P%');
GO
```

The following query uses `IN`.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName, e.JobTitle
FROM Person.Person AS p JOIN HumanResources.Employee AS e
   ON e.BusinessEntityID = p.BusinessEntityID
JOIN HumanResources.EmployeeDepartmentHistory AS edh
   ON e.BusinessEntityID = edh.BusinessEntityID
WHERE edh.DepartmentID IN
(SELECT DepartmentID
   FROM HumanResources.Department
   WHERE Name LIKE 'P%');
GO
```

E. Using NOT EXISTS

NOT EXISTS works the opposite of EXISTS. The WHERE clause in NOT EXISTS is satisfied if no rows are returned by the subquery. The following example finds employees who are not in departments which have names that start with `P`.

```
SELECT p.FirstName, p.LastName, e.JobTitle
FROM Person.Person AS p
JOIN HumanResources.Employee AS e
   ON e.BusinessEntityID = p.BusinessEntityID
WHERE NOT EXISTS
(SELECT *
   FROM HumanResources.Department AS d
   JOIN HumanResources.EmployeeDepartmentHistory AS edh
      ON d.DepartmentID = edh.DepartmentID
   WHERE e.BusinessEntityID = edh.BusinessEntityID
   AND d.Name LIKE 'P%')
ORDER BY LastName, FirstName
GO
```

Here is the result set.

```
FirstName                     LastName                      Title
----------------------------- ----------------------------- ------------
Syed                          Abbas                         Pacific Sales Manager
Hazem                         Abolrous                      Quality Assurance Manager
Humberto                      Acevedo                       Application Specialist
Pilar                         Ackerman                      Shipping & Receiving Superviso
Francois                      Ajenstat                      Database Administrator
```

François	Ajenstat	Database Administrator
Amy	Alberts	European Sales Manager
Sean	Alexander	Quality Assurance Technician
Pamela	Ansman-Wolfe	Sales Representative
Zainal	Arifin	Document Control Manager
David	Barber	Assistant to CFO
Paula	Barreto de Mattos	Human Resources Manager
Shai	Bassli	Facilities Manager
Wanida	Benshoof	Marketing Assistant
Karen	Berg	Application Specialist
Karen	Berge	Document Control Assistant
Andreas	Berglund	Quality Assurance Technician
Matthias	Berndt	Shipping & Receiving Clerk
Jo	Berry	Janitor
Jimmy	Bischoff	Stocker
Michael	Blythe	Sales Representative
David	Bradley	Marketing Manager
Kevin	Brown	Marketing Assistant
David	Campbell	Sales Representative
Jason	Carlson	Information Services Manager
Fernando	Caro	Sales Representative
Sean	Chai	Document Control Assistant
Sootha	Charncherngkha	Quality Assurance Technician
Hao	Chen	HR Administrative Assistant
Kevin	Chrisulis	Network Administrator
Pat	Coleman	Janitor
Stephanie	Conroy	Network Manager
Debra	Core	Application Specialist
Ovidiu	Crăcium	Sr. Tool Designer
Grant	Culbertson	HR Administrative Assistant
Mary	Dempsey	Marketing Assistant
Thierry	D'Hers	Tool Designer
Terri	Duffy	VP Engineering
Susan	Eaton	Stocker
Terry	Eminhizer	Marketing Specialist
Gail	Erickson	Design Engineer
Janice	Galvin	Tool Designer
Mary	Gibson	Marketing Specialist
Jossef	Goldberg	Design Engineer
Sariya	Harnpadoungsataya	Marketing Specialist
Mark	Harrington	Quality Assurance Technician
Magnus	Hedlund	Facilities Assistant
Shu	Ito	Sales Representative
Stephen	Jiang	North American Sales Manager
Willis	Johnson	Recruiter
Brannon	Jones	Finance Manager
Tengiz	Kharatishvili	Control Specialist
Christian	Kleinerman	Maintenance Supervisor
Vamsi	Kuppa	Shipping & Receiving Clerk
David	Liu	Accounts Manager
Vidur	Luthra	Recruiter
Stuart	Macrae	Janitor
Diane	Margheim	Research & Development Enginee
Mindy	Martin	Benefits Specialist
Gigi	Matthew	Research & Development Enginee
Tete	Mensa-Annan	Sales Representative
Ramesh	Meyyappan	Application Specialist
Dylan	Miller	Research & Development Manager
Linda	Mitchell	Sales Representative
Barbara	Moreland	Accountant
Laura	Norman	Chief Financial Officer
Chris	Norred	Control Specialist
Jae	Pak	Sales Representative
Wanda	Parks	Janitor
Deborah	Poe	Accounts Receivable Specialist
Kim	Ralls	Stocker
Tsvi	Reiter	Sales Representative
Sharon	Salavaria	Design Engineer
Ken	Sanchez	Chief Executive Officer

```
José                            Saraiva                 Sales Representative
Mike                            Seamans                 Accountant
Ashvini                         Sharma                  Network Administrator
Janet                           Sheperdigian            Accounts Payable Specialist
Candy                           Spoon                   Accounts Receivable Specialist
Michael                         Sullivan                Sr. Design Engineer
Dragan                          Tomic                   Accounts Payable Specialist
Lynn                            Tsoflias                Sales Representative
Rachel                          Valdez                  Sales Representative
Garrett                         Vargar                  Sales Representative
Ranjit                          Varkey Chudukatil       Sales Representative
Bryan                           Walton                  Accounts Receivable Specialist
Jian Shuo                       Wang                    Engineering Manager
Brian                           Welcker                 VP Sales
Jill                            Williams                Marketing Specialist
Dan                             Wilson                  Database Administrator
John                            Wood                    Marketing Specialist
Peng                            Wu                      Quality Assurance Supervisor

(91 row(s) affected)
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

F. Using EXISTS

The following example identifies whether any rows in the `ProspectiveBuyer` table could be matches to rows in the `DimCustomer` table. The query will return rows only when both the `LastName` and `BirthDate` values in the two tables match.

```
-- Uses AdventureWorks

SELECT a.LastName, a.BirthDate
FROM DimCustomer AS a
WHERE EXISTS
(SELECT *
    FROM dbo.ProspectiveBuyer AS b
    WHERE (a.LastName = b.LastName) AND (a.BirthDate = b.BirthDate));
```

G. Using NOT EXISTS

NOT EXISTS works as the opposite as EXISTS. The WHERE clause in NOT EXISTS is satisfied if no rows are returned by the subquery. The following example finds rows in the `DimCustomer` table where the `LastName` and `BirthDate` do not match any entries in the `ProspectiveBuyers` table.

```
-- Uses AdventureWorks

SELECT a.LastName, a.BirthDate
FROM DimCustomer AS a
WHERE NOT EXISTS
(SELECT *
    FROM dbo.ProspectiveBuyer AS b
    WHERE (a.LastName = b.LastName) AND (a.BirthDate = b.BirthDate));
```

See Also

Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
WHERE (Transact-SQL)

IN (Transact-SQL)

8/27/2018 • 3 minutes to read • Edit Online

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

Determines whether a specified value matches any value in a subquery or a list.

Transact-SQL Syntax Conventions

Syntax

```
test_expression [ NOT ] IN
    ( subquery | expression [ ,...n ]
    )
```

Arguments

test_expression
Is any valid expression.

subquery
Is a subquery that has a result set of one column. This column must have the same data type as *test_expression*.

expression[**,**... *n*]
Is a list of expressions to test for a match. All expressions must be of the same type as *test_expression*.

Result Types

Boolean

Result Value

If the value of *test_expression* is equal to any value returned by *subquery* or is equal to any *expression* from the comma-separated list, the result value is TRUE; otherwise, the result value is FALSE.

Using NOT IN negates the *subquery* value or *expression*.

Caution

Any null values returned by *subquery* or *expression* that are compared to *test_expression* using IN or NOT IN return UNKNOWN. Using null values in together with IN or NOT IN can produce unexpected results.

Remarks

Explicitly including an extremely large number of values (many thousands of values separated by commas) within the parentheses, in an IN clause can consume resources and return errors 8623 or 8632. To work around this problem, store the items in the IN list in a table, and use a SELECT subquery within an IN clause.

Error 8623:

```
The query processor ran out of internal resources and could not produce a query plan. This is a rare event and
only expected for extremely complex queries or queries that reference a very large number of tables or
partitions. Please simplify the query. If you believe you have received this message in error, contact Customer
Support Services for more information.
```

Error 8632:

```
Internal error: An expression services limit has been reached. Please look for potentially complex expressions
in your query, and try to simplify them.
```

Examples

A. Comparing OR and IN

The following example selects a list of the names of employees who are design engineers, tool designers, or marketing assistants.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName, e.JobTitle
FROM Person.Person AS p
JOIN HumanResources.Employee AS e
    ON p.BusinessEntityID = e.BusinessEntityID
WHERE e.JobTitle = 'Design Engineer'
   OR e.JobTitle = 'Tool Designer'
   OR e.JobTitle = 'Marketing Assistant';
GO
```

However, you retrieve the same results by using IN.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName, e.JobTitle
FROM Person.Person AS p
JOIN HumanResources.Employee AS e
    ON p.BusinessEntityID = e.BusinessEntityID
WHERE e.JobTitle IN ('Design Engineer', 'Tool Designer', 'Marketing Assistant');
GO
```

Here is the result set from either query.

```
FirstName   LastName      Title
---------   ---------   --------------------
Sharon      Salavaria   Design Engineer
Gail        Erickson    Design Engineer
Jossef      Goldberg    Design Engineer
Janice      Galvin      Tool Designer
Thierry     D'Hers      Tool Designer
Wanida      Benshoof    Marketing Assistant
Kevin       Brown       Marketing Assistant
Mary        Dempsey     Marketing Assistant

(8 row(s) affected)
```

B. Using IN with a subquery

The following example finds all IDs for the salespeople in the `SalesPerson` table for employees who have a sales quota greater than $250,000 for the year, and then selects from the `Employee` table the names of all employees where `EmployeeID` that match the results from the `SELECT` subquery.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName
FROM Person.Person AS p
    JOIN Sales.SalesPerson AS sp
    ON p.BusinessEntityID = sp.BusinessEntityID
WHERE p.BusinessEntityID IN
   (SELECT BusinessEntityID
   FROM Sales.SalesPerson
   WHERE SalesQuota > 250000);
GO
```

Here is the result set.

```
FirstName   LastName
---------   --------
Tsvi         Reiter
Michael      Blythe
Tete         Mensa-Annan

(3 row(s) affected)
```

C. Using NOT IN with a subquery

The following example finds the salespersons who do not have a quota greater than $250,000. `NOT IN` finds the salespersons who do not match the items in the values list.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName
FROM Person.Person AS p
    JOIN Sales.SalesPerson AS sp
    ON p.BusinessEntityID = sp.BusinessEntityID
WHERE p.BusinessEntityID NOT IN
   (SELECT BusinessEntityID
   FROM Sales.SalesPerson
   WHERE SalesQuota > 250000);
GO
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

D. Using IN and NOT IN

The following example finds all entries in the `FactInternetSales` table that match `SalesReasonKey` values in the `DimSalesReason` table.

```
-- Uses AdventureWorks

SELECT * FROM FactInternetSalesReason
WHERE SalesReasonKey
IN (SELECT SalesReasonKey FROM DimSalesReason);
```

The following example finds all entries in the `FactInternetSalesReason` table that do not match `SalesReasonKey` values in the `DimSalesReason` table.

```
-- Uses AdventureWorks

SELECT * FROM FactInternetSalesReason
WHERE SalesReasonKey
NOT IN (SELECT SalesReasonKey FROM DimSalesReason);
```

E. Using IN with an expression list

The following example finds all IDs for the salespeople in the `DimEmployee` table for employees who have a first name that is either `Mike` or `Michael`.

```
-- Uses AdventureWorks

SELECT FirstName, LastName
FROM DimEmployee
WHERE FirstName IN ('Mike', 'Michael');
```

See Also

CASE (Transact-SQL)
Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)
ALL (Transact-SQL)
SOME | ANY (Transact-SQL)

LIKE (Transact-SQL)

8/27/2018 • 10 minutes to read • Edit Online

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

Determines whether a specific character string matches a specified pattern. A pattern can include regular characters and wildcard characters. During pattern matching, regular characters must exactly match the characters specified in the character string. However, wildcard characters can be matched with arbitrary fragments of the character string. Using wildcard characters makes the LIKE operator more flexible than using the = and != string comparison operators. If any one of the arguments is not of character string data type, the SQL Server Database Engine converts it to character string data type, if it is possible.

Transact-SQL Syntax Conventions

Syntax

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

match_expression [ NOT ] LIKE pattern [ ESCAPE escape_character ]
```

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

match_expression [ NOT ] LIKE pattern
```

Arguments

match_expression
Is any valid expression of character data type.

pattern
Is the specific string of characters to search for in *match_expression*, and can include the following valid wildcard characters. *pattern* can be a maximum of 8,000 bytes.

WILDCARD CHARACTER	DESCRIPTION	EXAMPLE
%	Any string of zero or more characters.	WHERE title LIKE '%computer%' finds all book titles with the word 'computer' anywhere in the book title.
_ (underscore)	Any single character.	WHERE au_fname LIKE '_ean' finds all four-letter first names that end with ean (Dean, Sean, and so on).

WILDCARD CHARACTER	DESCRIPTION	EXAMPLE
[ ]	Any single character within the specified range ([a-f]) or set ([abcdef]).	WHERE au_lname LIKE '[C-P]arsen' finds author last names ending with arsen and starting with any single character between C and P, for example Carsen, Larsen, Karsen, and so on. In range searches, the characters included in the range may vary depending on the sorting rules of the collation.
[^]	Any single character not within the specified range ([^a-f]) or set ([^abcdef]).	WHERE au_lname LIKE 'de[^l]%' all author last names starting with de and where the following letter is not l.

escape_character
Is a character that is put in front of a wildcard character to indicate that the wildcard should be interpreted as a regular character and not as a wildcard. *escape_character* is a character expression that has no default and must evaluate to only one character.

Result Types

Boolean

Result Value

LIKE returns TRUE if the *match_expression* matches the specified *pattern*.

Remarks

When you perform string comparisons by using LIKE, all characters in the pattern string are significant. This includes leading or trailing spaces. If a comparison in a query is to return all rows with a string LIKE 'abc ' (abc followed by a single space), a row in which the value of that column is abc (abc without a space) is not returned. However, trailing blanks, in the expression to which the pattern is matched, are ignored. If a comparison in a query is to return all rows with the string LIKE 'abc' (abc without a space), all rows that start with abc and have zero or more trailing blanks are returned.

A string comparison using a pattern that contains **char** and **varchar** data may not pass a LIKE comparison because of how the data is stored. You should understand the storage for each data type and where a LIKE comparison may fail. The following example passes a local **char** variable to a stored procedure and then uses pattern matching to find all of the employees whose last names start with a specified set of characters.

```
-- Uses AdventureWorks

CREATE PROCEDURE FindEmployee @EmpLName char(20)
AS
SELECT @EmpLName = RTRIM(@EmpLName) + '%';
SELECT p.FirstName, p.LastName, a.City
FROM Person.Person p JOIN Person.Address a ON p.BusinessEntityID = a.AddressID
WHERE p.LastName LIKE @EmpLName;
GO
EXEC FindEmployee @EmpLName = 'Barb';
GO
```

In the `FindEmployee` procedure, no rows are returned because the **char** variable (`@EmpLName`) contains trailing blanks whenever the name contains fewer than 20 characters. Because the `LastName` column is **varchar**, there are no trailing blanks. This procedure fails because the trailing blanks are significant.

However, the following example succeeds because trailing blanks are not added to a **varchar** variable.

```
-- Uses AdventureWorks

CREATE PROCEDURE FindEmployee @EmpLName varchar(20)
AS
SELECT @EmpLName = RTRIM(@EmpLName) + '%';
SELECT p.FirstName, p.LastName, a.City
FROM Person.Person p JOIN Person.Address a ON p.BusinessEntityID = a.AddressID
WHERE p.LastName LIKE @EmpLName;
GO
EXEC FindEmployee @EmpLName = 'Barb';
```

Here is the result set.

```
FirstName      LastName            City
----------     -------------------- ---------------
Angela         Barbariol            Snohomish
David          Barber               Snohomish
(2 row(s) affected)
```

Pattern Matching by Using LIKE

LIKE supports ASCII pattern matching and Unicode pattern matching. When all arguments (*match_expression*, *pattern*, and *escape_character*, if present) are ASCII character data types, ASCII pattern matching is performed. If any one of the arguments are of Unicode data type, all arguments are converted to Unicode and Unicode pattern matching is performed. When you use Unicode data (**nchar** or **nvarchar** data types) with LIKE, trailing blanks are significant; however, for non-Unicode data, trailing blanks are not significant. Unicode LIKE is compatible with the ISO standard. ASCII LIKE is compatible with earlier versions of SQL Server.

The following is a series of examples that show the differences in rows returned between ASCII and Unicode LIKE pattern matching.

```
-- ASCII pattern matching with char column
CREATE TABLE t (col1 char(30));
INSERT INTO t VALUES ('Robert King');
SELECT *
FROM t
WHERE col1 LIKE '% King';   -- returns 1 row

-- Unicode pattern matching with nchar column
CREATE TABLE t (col1 nchar(30));
INSERT INTO t VALUES ('Robert King');
SELECT *
FROM t
WHERE col1 LIKE '% King';   -- no rows returned

-- Unicode pattern matching with nchar column and RTRIM
CREATE TABLE t (col1 nchar (30));
INSERT INTO t VALUES ('Robert King');
SELECT *
FROM t
WHERE RTRIM(col1) LIKE '% King';   -- returns 1 row
```

> **NOTE**
>
> LIKE comparisons are affected by collation. For more information, see COLLATE (Transact-SQL).

Using the % Wildcard Character

If the LIKE '5%' symbol is specified, the Database Engine searches for the number 5 followed by any string of zero or more characters.

For example, the following query shows all dynamic management views in the **AdventureWorks2012** database, because they all start with the letters `dm`.

```
-- Uses AdventureWorks

SELECT Name
FROM sys.system_views
WHERE Name LIKE 'dm%';
GO
```

To see all objects that are not dynamic management views, use `NOT LIKE 'dm%'`. If you have a total of 32 objects and LIKE finds 13 names that match the pattern, NOT LIKE finds the 19 objects that do not match the LIKE pattern.

You may not always find the same names with a pattern such as `LIKE '[^d][^m]%'`. Instead of 19 names, you may find only 14, with all the names that start with `d` or have `m` as the second letter eliminated from the results, and the dynamic management view names. This is because match strings with negative wildcard characters are evaluated in steps, one wildcard at a time. If the match fails at any point in the evaluation, it is eliminated.

Using Wildcard Characters As Literals

You can use the wildcard pattern matching characters as literal characters. To use a wildcard character as a literal character, enclose the wildcard character in brackets. The following table shows several examples of using the LIKE keyword and the [ ] wildcard characters.

SYMBOL	MEANING
LIKE '5[%]'	5%
LIKE '[_]n'	_n
LIKE '[a-cdf]'	a, b, c, d, or f
LIKE '[-acdf]'	-, a, c, d, or f
LIKE '[[ ]'	[
LIKE ']'	]
LIKE 'abc[_]d%'	abc_d and abc_de
LIKE 'abc[def]'	abcd, abce, and abcf

Pattern Matching with the ESCAPE Clause

You can search for character strings that include one or more of the special wildcard characters. For example, the discounts table in a customers database may store discount values that include a percent sign (%). To search for the percent sign as a character instead of as a wildcard character, the ESCAPE keyword and escape character must be provided. For example, a sample database contains a column named comment that contains the text

30%. To search for any rows that contain the string 30% anywhere in the comment column, specify a WHERE clause such as `WHERE comment LIKE '%30!%%' ESCAPE '!'`. If ESCAPE and the escape character are not specified, the Database Engine returns any rows with the string 30.

If there is no character after an escape character in the LIKE pattern, the pattern is not valid and the LIKE returns FALSE. If the character after an escape character is not a wildcard character, the escape character is discarded and the character following the escape is treated as a regular character in the pattern. This includes the percent sign (%), underscore (_), and left bracket ([) wildcard characters when they are enclosed in double brackets ([ ]). Also, within the double bracket characters ([ ]), escape characters can be used and the caret (^), hyphen (-), and right bracket (]) can be escaped.

0x0000 (**char(0)**) is an undefined character in Windows collations and cannot be included in LIKE.

Examples

A. Using LIKE with the % wildcard character

The following example finds all telephone numbers that have area code `415` in the `PersonPhone` table.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName, ph.PhoneNumber
FROM Person.PersonPhone AS ph
INNER JOIN Person.Person AS p
ON ph.BusinessEntityID = p.BusinessEntityID
WHERE ph.PhoneNumber LIKE '415%'
ORDER by p.LastName;
GO
```

Here is the result set.

```
FirstName             LastName             Phone
-----------------     -------------------  ------------
Ruben                 Alonso               415-555-124
Shelby                Cook                 415-555-0121
Karen                 Hu                   415-555-0114
John                  Long                 415-555-0147
David                 Long                 415-555-0123
Gilbert               Ma                   415-555-0138
Meredith              Moreno               415-555-0131
Alexandra             Nelson               415-555-0174
Taylor                Patterson            415-555-0170
Gabrielle              Russell             415-555-0197
Dalton                 Simmons             415-555-0115
(11 row(s) affected)
```

B. Using NOT LIKE with the % wildcard character

The following example finds all telephone numbers in the `PersonPhone` table that have area codes other than `415`.

```
-- Uses AdventureWorks

SELECT p.FirstName, p.LastName, ph.PhoneNumber
FROM Person.PersonPhone AS ph
INNER JOIN Person.Person AS p
ON ph.BusinessEntityID = p.BusinessEntityID
WHERE ph.PhoneNumber NOT LIKE '415%' AND p.FirstName = 'Gail'
ORDER BY p.LastName;
GO
```

Here is the result set.

```
FirstName               LastName            Phone
----------------------- ------------------- -------------------
Gail                    Alexander           1 (11) 500 555-0120
Gail                    Butler              1 (11) 500 555-0191
Gail                    Erickson            834-555-0132
Gail                    Erickson            849-555-0139
Gail                    Griffin             450-555-0171
Gail                    Moore               155-555-0169
Gail                    Russell             334-555-0170
Gail                    Westover            305-555-0100
(8 row(s) affected)
```

C. Using the ESCAPE clause

The following example uses the `ESCAPE` clause and the escape character to find the exact character string `10-15%` in column `c1` of the `mytbl2` table.

```
USE tempdb;
GO
IF EXISTS(SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
      WHERE TABLE_NAME = 'mytbl2')
   DROP TABLE mytbl2;
GO
USE tempdb;
GO
CREATE TABLE mytbl2
(
 c1 sysname
);
GO
INSERT mytbl2 VALUES ('Discount is 10-15% off'), ('Discount is .10-.15 off');
GO
SELECT c1
FROM mytbl2
WHERE c1 LIKE '%10-15!% off%' ESCAPE '!';
GO
```

D. Using the [ ] wildcard characters

The following example finds employees on the `Person` table with the first name of `Cheryl` or `Sheryl`.

```
-- Uses AdventureWorks

SELECT BusinessEntityID, FirstName, LastName
FROM Person.Person
WHERE FirstName LIKE '[CS]heryl';
GO
```

The following example finds the rows for employees in the `Person` table with last names of `Zheng` or `Zhang`.

```
-- Uses AdventureWorks

SELECT LastName, FirstName
FROM Person.Person
WHERE LastName LIKE 'Zh[ae]ng'
ORDER BY LastName ASC, FirstName ASC;
GO
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

E. Using LIKE with the % wildcard character

The following example finds all employees in the `DimEmployee` table with telephone numbers that start with `612`.

```
-- Uses AdventureWorks

SELECT FirstName, LastName, Phone
FROM DimEmployee
WHERE phone LIKE '612%'
ORDER by LastName;
```

F. Using NOT LIKE with the % wildcard character

The following example finds all telephone numbers in the `DimEmployee` table that do not start with `612` . .

```
-- Uses AdventureWorks

SELECT FirstName, LastName, Phone
FROM DimEmployee
WHERE phone NOT LIKE '612%'
ORDER by LastName;
```

G. Using LIKE with the _ wildcard character

The following example finds all telephone numbers that have an area code starting with `6` and ending in `2` in the `DimEmployee` table. Note that the % wildcard character is also included at the end of the search pattern since the area code is the first part of the phone number and additional characters exist after in the column value.

```
-- Uses AdventureWorks

SELECT FirstName, LastName, Phone
FROM DimEmployee
WHERE phone LIKE '6_2%'
ORDER by LastName;
```

See Also

Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)

NOT (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Negates a Boolean input.

Transact-SQL Syntax Conventions

Syntax

```
[ NOT ] boolean_expression
```

Arguments

boolean_expression
Is any valid Boolean expression.

Result Types

Boolean

Result Value

NOT reverses the value of any Boolean expression.

Remarks

Using NOT negates an expression.

The following table shows the results of comparing TRUE and FALSE values using the NOT operator.

	NOT
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Examples

The following example finds all Silver colored bicycles that do not have a standard price over $400.

```
-- Uses AdventureWorks

SELECT ProductID, Name, Color, StandardCost
FROM Production.Product
WHERE ProductNumber LIKE 'BK-%' AND Color = 'Silver' AND NOT StandardCost > 400;
GO
```

Here is the result set.

```
ProductID   Name                    Color         StandardCost
---------   --------------------    ------      ------------
984         Mountain-500 Silver, 40  Silver        308.2179
985         Mountain-500 Silver, 42  Silver        308.2179
986         Mountain-500 Silver, 44  Silver        308.2179
987         Mountain-500 Silver, 48  Silver        308.2179
988         Mountain-500 Silver, 52  Silver        308.2179
(6 row(s) affected)
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

The following example restricts results to `SalesOrderNumber` to values starting with `SO6` and `ProductKeys` greater than or equal to 400.

```
-- Uses AdventureWorks

SELECT ProductKey, CustomerKey, OrderDateKey, ShipDateKey
FROM FactInternetSales
WHERE SalesOrderNumber LIKE 'SO6%' AND NOT ProductKey < 400;
```

See Also

Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)

OR (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Combines two conditions. When more than one logical operator is used in a statement, OR operators are evaluated after AND operators. However, you can change the order of evaluation by using parentheses.

Transact-SQL Syntax Conventions

Syntax

```
boolean_expression OR boolean_expression
```

Arguments

boolean_expression
Is any valid expression that returns TRUE, FALSE, or UNKNOWN.

Result Types

Boolean

Result Value

OR returns TRUE when either of the conditions is TRUE.

Remarks

The following table shows the result of the OR operator.

	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Examples

The following example uses the `vEmployeeDepartmentHistory` view to retrieve the names of `Quality Assurance` personnel who work either the evening shift or the night shift. If the parentheses are omitted, the query returns `Quality Assurance` employees who work the evening shift and all employees who work the night shift.

```
-- Uses AdventureWorks

SELECT FirstName, LastName, Shift
FROM HumanResources.vEmployeeDepartmentHistory
WHERE Department = 'Quality Assurance'
   AND (Shift = 'Evening' OR Shift = 'Night');
```

Here is the result set.

```
FirstName    LastName         Shift
------------ ---------------- -------
Andreas      Berglund         Evening
Sootha       Charncherngkha   Night
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

The following example retrieves the names of employees who either earn a `BaseRate` less than 20 or have a `HireDate` January 1, 2001 or later.

```
-- Uses AdventureWorks

SELECT FirstName, LastName, BaseRate, HireDate
FROM DimEmployee
WHERE BaseRate < 10 OR HireDate >= '2001-01-01';
```

See Also

Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)

SOME | ANY (Transact-SQL)

6/20/2018 • 3 minutes to read • Edit Online

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

Compares a scalar value with a single-column set of values. SOME and ANY are equivalent.

Transact-SQL Syntax Conventions

Syntax

```
scalar_expression { = | < > | ! = | > | > = | ! > | < | < = | ! < }
     { SOME | ANY } ( subquery )
```

Arguments

scalar_expression
Is any valid expression.

{ = | <> | != | > | >= | !> | < | <= | !< }
Is any valid comparison operator.

SOME | ANY
Specifies that a comparison should be made.

subquery
Is a subquery that has a result set of one column. The data type of the column returned must be the same data type as *scalar_expression*.

Result Types

Boolean

Result Value

SOME or ANY returns **TRUE** when the comparison specified is TRUE for any pair (*scalar_expression*,*x*) where *x* is a value in the single-column set; otherwise, returns **FALSE**.

Remarks

SOME requires the *scalar_expression* to compare positively to at least one value returned by the subquery. For statements that require the *scalar_expression* to compare positively to every value that is returned by the subquery, see ALL (Transact-SQL). For instance, if the subquery returns values of 2 and 3, *scalar_expression* = SOME (subquery) would evaluate as TRUE for a *scalar_express* of 2. If the subquery returns values of 2 and 3, *scalar_expression* = ALL (subquery) would evaluate as FALSE, because some of the values of the subquery (the value of 3) would not meet the criteria of the expression.

Examples

A. Running a simple example

The following statements create a simple table and add the values of `1`, `2`, `3`, and `4` to the `ID` column.

```
CREATE TABLE T1
(ID int) ;
GO
INSERT T1 VALUES (1) ;
INSERT T1 VALUES (2) ;
INSERT T1 VALUES (3) ;
INSERT T1 VALUES (4) ;
```

The following query returns `TRUE` because `3` is less than some of the values in the table.

```
IF 3 < SOME (SELECT ID FROM T1)
PRINT 'TRUE'
ELSE
PRINT 'FALSE' ;
```

The following query returns `FALSE` because `3` is not less than all of the values in the table.

```
IF 3 < ALL (SELECT ID FROM T1)
PRINT 'TRUE'
ELSE
PRINT 'FALSE' ;
```

B. Running a practical example

The following example creates a stored procedure that determines whether all the components of a specified `SalesOrderID` in the `AdventureWorks2012` database can be manufactured in the specified number of days. The example uses a subquery to create a list of the number of `DaysToManufacture` value for all the components of the specific `SalesOrderID`, and then tests whether any of the values that are returned by the subquery are greater than the number of days specified. If every value of `DaysToManufacture` that is returned is less than the number provided, the condition is TRUE and the first message is printed.

```
-- Uses AdventureWorks

CREATE PROCEDURE ManyDaysToComplete @OrderID int, @NumberOfDays int
AS
IF
@NumberOfDays < SOME
   (
    SELECT DaysToManufacture
    FROM Sales.SalesOrderDetail
    JOIN Production.Product
    ON Sales.SalesOrderDetail.ProductID = Production.Product.ProductID
    WHERE SalesOrderID = @OrderID
   )
PRINT 'At least one item for this order cannot be manufactured in specified number of days.'
ELSE
PRINT 'All items for this order can be manufactured in the specified number of days or less.' ;
```

To test the procedure, execute the procedure by using the `SalesOrderID``49080`, which has one component that requires `2` days and two components that require 0 days. The first statement meets the criteria. The second query does not.

```
EXECUTE ManyDaysToComplete 49080, 2 ;
```

Here is the result set.

```
All items for this order can be manufactured in the specified number of days or less.
```

```
EXECUTE ManyDaysToComplete 49080, 1 ;
```

Here is the result set.

```
At least one item for this order cannot be manufactured in specified number of days.
```

See Also

ALL (Transact-SQL)
CASE (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
WHERE (Transact-SQL)
IN (Transact-SQL)

:: (Scope Resolution) (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The scope resolution operator **::** provides access to static members of a compound data type. A compound data type is one that contains multiple simple data types and methods, such as the built-in CLR types and custom SQLCLR User-Defined Types (UDTs).

Examples

The following example shows how to use the scope resolution operator to access the `GetRoot()` member of the `hierarchyid` type.

```
DECLARE @hid hierarchyid;
SELECT @hid = hierarchyid::GetRoot();
PRINT @hid.ToString();
```

Here is the result set.

```
/
```

See Also

Operators (Transact-SQL)

String Operators (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server provides the following string operators. String concatenation operators can combine two or more character or binary strings, columns, or a combination of strings and column names into one expression. Wildcard string operators can matches one or more characters in a string comparison operation such as LIKE or PATINDEX.

Section Heading

+ (String Concatenation)

+= (String Concatenation Assignment)

% (Wildcard - Character(s) to Match)

[ ] (Wildcard - Character(s) to Match)

[^] (Wildcard - Character(s) Not to Match)

_ (Wildcard - Match One Character)

+ (String Concatenation) (Transact-SQL)

8/27/2018 • 4 minutes to read • Edit Online

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

An operator in a string expression that concatenates two or more character or binary strings, columns, or a combination of strings and column names into one expression (a string operator). For example `SELECT 'book'+'case';` returns `bookcase`.

Transact-SQL Syntax Conventions

Syntax

```
expression + expression
```

Arguments

expression
Is any valid expression of any one of the data types in the character and binary data type category, except the **image**, **ntext**, or **text** data types. Both expressions must be of the same data type, or one expression must be able to be implicitly converted to the data type of the other expression.

An explicit conversion to character data must be used when concatenating binary strings and any characters between the binary strings. The following example shows when `CONVERT`, or `CAST`, must be used with binary concatenation and when `CONVERT`, or `CAST`, does not have to be used.

```
DECLARE @mybin1 varbinary(5), @mybin2 varbinary(5)
SET @mybin1 = 0xFF
SET @mybin2 = 0xA5
-- No CONVERT or CAST function is required because this example
-- concatenates two binary strings.
SELECT @mybin1 + @mybin2
-- A CONVERT or CAST function is required because this example
-- concatenates two binary strings plus a space.
SELECT CONVERT(varchar(5), @mybin1) + ' '
   + CONVERT(varchar(5), @mybin2)
-- Here is the same conversion using CAST.
SELECT CAST(@mybin1 AS varchar(5)) + ' '
   + CAST(@mybin2 AS varchar(5))
```

Result Types

Returns the data type of the argument with the highest precedence. For more information, see Data Type Precedence (Transact-SQL).

Remarks

The + (String Concatenation) operator behaves differently when it works with an empty, zero-length string than when it works with NULL, or unknown values. A zero-length character string can be specified as two single quotation marks without any characters inside the quotation marks. A zero-length binary string can be specified as

0x without any byte values specified in the hexadecimal constant. Concatenating a zero-length string always concatenates the two specified strings. When you work with strings with a null value, the result of the concatenation depends on the session settings. Just like arithmetic operations that are performed on null values, when a null value is added to a known value the result is typically an unknown value, a string concatenation operation that is performed with a null value should also produce a null result. However, you can change this behavior by changing the setting of `CONCAT_NULL_YIELDS_NULL` for the current session. For more information, see SET CONCAT_NULL_YIELDS_NULL (Transact-SQL).

If the result of the concatenation of strings exceeds the limit of 8,000 bytes, the result is truncated. However, if at least one of the strings concatenated is a large value type, truncation does not occur.

Examples

A. Using string concatenation

The following example creates a single column under the column heading `Name` from multiple character columns, with the last name of the person followed by a comma, a single space, and then the first name of the person. The result set is in ascending, alphabetical order by the last name, and then by the first name.

```
-- Uses AdventureWorks

SELECT (LastName + ', ' + FirstName) AS Name
FROM Person.Person
ORDER BY LastName ASC, FirstName ASC;
```

B. Combining numeric and date data types

The following example uses the `CONVERT` function to concatenate **numeric** and **date** data types.

```
-- Uses AdventureWorks

SELECT 'The order is due on ' + CONVERT(varchar(12), DueDate, 101)
FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 50001;
GO
```

Here is the result set.

```
-----------------------------------------------
The order is due on 04/23/2007
(1 row(s) affected)
```

C. Using multiple string concatenation

The following example concatenates multiple strings to form one long string to display the last name and the first initial of the vice presidents at Adventure Works Cycles. A comma is added after the last name and a period after the first initial.

```
-- Uses AdventureWorks

SELECT (LastName + ',' + SPACE(1) + SUBSTRING(FirstName, 1, 1) + '.') AS Name, e.JobTitle
FROM Person.Person AS p
    JOIN HumanResources.Employee AS e
    ON p.BusinessEntityID = e.BusinessEntityID
WHERE e.JobTitle LIKE 'Vice%'
ORDER BY LastName ASC;
GO
```

Here is the result set.

```
Name              Title
------------      ---------------`
Duffy, T.         Vice President of Engineering
Hamilton, J.      Vice President of Production
Welcker, B.       Vice President of Sales

(3 row(s) affected)
```

D. Using large strings in concatenation

The following example concatenates multiple strings to form one long string and then tries to compute the length of the final string. The final length of resultset is 16000, because expression evaluation starts from left that is, @x + @z + @y => (@x + @z) + @y. In this case the result of (@x + @z) is truncated at 8000 bytes and then @y is added to the resultset, which makes the final string length 16000. Since @y is a large value type string, truncation does not occur.

```
DECLARE @x varchar(8000) = replicate('x', 8000)
DECLARE @y varchar(max) = replicate('y', 8000)
DECLARE @z varchar(8000) = replicate('z',8000)
SET @y = @x + @z + @y
-- The result of following select is 16000
SELECT len(@y) AS y
GO
```

Here is the result set.

```
y
-------
16000

(1 row(s) affected)
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

E. Using multiple string concatenation

The following example concatenates multiple strings to form one long string to display the last name and the first initial of the vice presidents within a sample database. A comma is added after the last name and a period after the first initial.

```
-- Uses AdventureWorks

SELECT (LastName + ', ' + SUBSTRING(FirstName, 1, 1) + '.') AS Name, Title
FROM DimEmployee
WHERE Title LIKE '%Vice Pres%'
ORDER BY LastName ASC;
```

Here is the result set.

```
Name              Title
------------      ---------------
Duffy, T.         Vice President of Engineering
Hamilton, J.      Vice President of Production
Welcker, B.       Vice President of Sales
```

See Also

+= (String Concatenation Assignment) (Transact-SQL)
ALTER DATABASE (Transact-SQL)
CAST and CONVERT (Transact-SQL)
Data Type Conversion (Database Engine)
Data Types (Transact-SQL)
Expressions (Transact-SQL)
Built-in Functions (Transact-SQL)
Operators (Transact-SQL)
SELECT (Transact-SQL)
SET Statements (Transact-SQL)

+= (String Concatenation Assignment) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Concatenates two strings and sets the string to the result of the operation. For example, if a variable @x equals 'Adventure', then @x += 'Works' takes the original value of @x, adds 'Works' to the string, and sets @x to that new value 'AdventureWorks'.

Transact-SQL Syntax Conventions

Syntax

```
expression += expression
```

Arguments

expression
Is any valid expression of any of the character data types.

Result Types

Returns the data type that is defined for the variable.

Remarks

SET @v1 += 'expression' is equivalent to SET @v1 = @v1 + ('expression'). Also, SET @v1 = @v2 + @v3 + @v4 is equivalent to SET @v1 = (@v2 + @v3) + @v4.

The += operator cannot be used without a variable. For example, the following code will cause an error:

```
SELECT 'Adventure' += 'Works'
```

Examples

A. Concatenation using += operator

The following example concatenates using the `+=` operator.

```
DECLARE @v1 varchar(40);
SET @v1 = 'This is the original.';
SET @v1 += ' More text.';
PRINT @v1;
```

Here is the result set.

```
This is the original. More text.
```

B. Order of evaluation while concatenating using += operator

The following example concatenates multiple strings to form one long string and then tries to compute the length of the final string. This example demonstrates the evaluation order and truncation rules, while using the concatenation operator.

```
DECLARE @x varchar(4000) = replicate('x', 4000)
DECLARE @z varchar(8000) = replicate('z',8000)
DECLARE @y varchar(max);

SET @y = '';
SET @y += @x + @z;
SELECT LEN(@y) AS Y; -- 8000

SET @y = '';
SET @y = @y + @x + @z;
SELECT LEN(@y) AS Y; -- 12000

SET @y = '';
SET @y = @y +(@x + @z);
SELECT LEN(@y) AS Y; -- 8000
-- or
SET @y = '';
SET @y = @x + @z + @y;
SELECT LEN(@y) AS Y; -- 8000
GO
```

Here is the result set.

```
Y
-------
8000

(1 row(s) affected)


Y
-------
12000

(1 row(s) affected)

Y
-------
8000

(1 row(s) affected)

Y
-------
8000

(1 row(s) affected)
```

See Also

Operators (Transact-SQL)
+= (Add Assignment) (Transact-SQL)
+ (String Concatenation) (Transact-SQL)

Percent character (Wildcard - Character(s) to Match) (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Matches any string of zero or more characters. This wildcard character can be used as either a prefix or a suffix.

Examples

The following example returns all the first names of people in the `Person` table of `AdventureWorks2012` that start with `Dan`.

```
-- Uses AdventureWorks

SELECT FirstName, LastName
FROM Person.Person
WHERE FirstName LIKE 'Dan%';
GO
```

See Also

LIKE (Transact-SQL)
Operators (Transact-SQL)
Expressions (Transact-SQL)
[ ] (Wildcard - Character(s) to Match)
[^] (Wildcard - Character(s) Not to Match)
_ (Wildcard - Match One Character)

[ ] (Wildcard - Character(s) to Match) (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Matches any single character within the specified range or set that is specified between brackets `[ ]`. These wildcard characters can be used in string comparisons that involve pattern matching, such as `LIKE` and `PATINDEX`.

Examples

A: Simple example

The following example returns the names of that start with the letter `m`. `[n-z]` specifies that the second letter must be somewhere in the range from `n` to `z`. The percent wildcard `%` allows any or no characters starting with the 3 character. The `model` and `msdb` databases meet this criteria. The `master` database does not and is excluded from the result set.

```
SELECT name FROM sys.databases
WHERE name LIKE 'm[n-z]%';
```

Here is the result set.

```
name
-----
model
msdb
```

You may have additional qualifying databases installed.

B: More complex example

The following example uses the [] operator to find the IDs and names of all Adventure Works employees who have addresses with a four-digit postal code.

```
-- Uses AdventureWorks

SELECT e.BusinessEntityID, p.FirstName, p.LastName, a.PostalCode
FROM HumanResources.Employee AS e
INNER JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID
INNER JOIN Person.BusinessEntityAddress AS ea ON e.BusinessEntityID = ea.BusinessEntityID
INNER JOIN Person.Address AS a ON a.AddressID = ea.AddressID
WHERE a.PostalCode LIKE '[0-9][0-9][0-9][0-9]';
```

Here is the result set:

```
EmployeeID      FirstName      LastName      PostalCode
----------      ---------      ---------     ----------
290             Lynn           Tsoflias      3000
```

See Also

LIKE (Transact-SQL)
PATINDEX (Transact-SQL)
% (Wildcard - Character(s) to Match) (Transact-SQL)
[^] (Wildcard - Character(s) Not to Match) (Transact-SQL)
_ (Wildcard - Match One Character) (Transact-SQL)

[^] (Wildcard - Character(s) Not to Match) (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Matches any single character that is not within the range or set specified between the square brackets.

Examples

The following example uses the [^] operator to find all the people in the `Contact` table who have first names that start with `Al` and have a third letter that is not the letter `a`.

```
-- Uses AdventureWorks

SELECT FirstName, LastName
FROM Person.Person
WHERE FirstName LIKE 'Al[^a]%'
ORDER BY FirstName;
```

See Also

LIKE (Transact-SQL)
PATINDEX (Transact-SQL)
% (Wildcard - Character(s) to Match) (Transact-SQL)
[ ] (Wildcard - Character(s) to Match) (Transact-SQL)
_ (Wildcard - Match One Character) (Transact-SQL)

_ (Wildcard - Match One Character) (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Use the underscore character _ to match any single character in a string comparison operation that involves pattern matching, such as `LIKE` and `PATINDEX`.

Examples

A: Simple example

The following example returns all database names that begin with the letter `m` and have the letter `d` as the third letter. The underscore character specifies that the second character of the name can be any letter. The `model` and `msdb` databases meet this criteria. The `master` database does not.

```
SELECT name FROM sys.databases
WHERE name LIKE 'm_d%';
```

Here is the result set.

```
name
-----
model
msdb
```

You may have additional databases that meet this criteria.

You can use multiple underscores to represent multiple characters. Changing the `LIKE` criteria to include two underscores `'m__%'` includes the master database in the result.

B: More complex example

The following example uses the _ operator to find all the people in the `Person` table, who have a three-letter first name that ends in `an`.

```
-- USE AdventureWorks2012

SELECT FirstName, LastName
FROM Person.Person
WHERE FirstName LIKE '_an'
ORDER BY FirstName;
```

C: Escaping the underscore character

The following example returns the names of the fixed database roles like `db_owner` and `db_ddladmin`, but it also returns the `dbo` user.

```
SELECT name FROM sys.database_principals
WHERE name LIKE 'db_%';
```

The underscore in the third character position is taken as a wildcard, and is not filtering for only principals starting with the letters `db_`. To escape the underscore enclose it in brackets `[_]`.

```
SELECT name FROM sys.database_principals
WHERE name LIKE 'db[_]%';
```

Now the `dbo` user is excluded.

Here is the result set.

```
name
-------------
db_owner
db_accessadmin
db_securityadmin
...
```

See Also

LIKE (Transact-SQL)
PATINDEX (Transact-SQL)
% (Wildcard - Character(s) to Match)
[ ] (Wildcard - Character(s) to Match)
[^] (Wildcard - Character(s) Not to Match)

Operator Precedence (Transact-SQL)

7/4/2018 • 2 minutes to read • Edit Online

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

When a complex expression has multiple operators, operator precedence determines the sequence in which the operations are performed. The order of execution can significantly affect the resulting value.

Operators have the precedence levels shown in the following table. An operator on higher levels is evaluated before an operator on a lower level.

LEVEL	OPERATORS
1	~ (Bitwise NOT)
2	* (Multiplication), / (Division), % (Modulus)
3	+ (Positive), - (Negative), + (Addition), + (Concatenation), - (Subtraction), & (Bitwise AND), ^ (Bitwise Exclusive OR), | (Bitwise OR)
4	=, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
5	NOT
6	AND
7	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
8	= (Assignment)

When two operators in an expression have the same operator precedence level, they are evaluated left to right based on their position in the expression. For example, in the expression that is used in the following `SET` statement, the subtraction operator is evaluated before the addition operator.

```
DECLARE @MyNumber int;
SET @MyNumber = 4 - 2 + 27;
-- Evaluates to 2 + 27 which yields an expression result of 29.
SELECT @MyNumber;
```

Use parentheses to override the defined precedence of the operators in an expression. Everything within the parentheses is evaluated first to yield a single value before that value can be used by any operator outside the parentheses.

For example, in the expression used in the following `SET` statement, the multiplication operator has a higher precedence than the addition operator. Therefore, it is evaluated first; the expression result is `13`.

```
DECLARE @MyNumber int;
SET @MyNumber = 2 * 4 + 5;
-- Evaluates to 8 + 5 which yields an expression result of 13.
SELECT @MyNumber;
```

In the expression used in the following `SET` statement, the parentheses cause the addition to be performed first. The expression result is `18`.

```
DECLARE @MyNumber int;
SET @MyNumber = 2 * (4 + 5);
-- Evaluates to 2 * 9 which yields an expression result of 18.
SELECT @MyNumber;
```

If an expression has nested parentheses, the most deeply nested expression is evaluated first. The following example contains nested parentheses, with the expression `5 - 3` in the most deeply nested set of parentheses. This expression yields a value of `2`. Then, the addition operator (`+`) adds this result to `4`. This yields a value of `6`. Finally, the `6` is multiplied by `2` to yield an expression result of `12`.

```
DECLARE @MyNumber int;
SET @MyNumber = 2 * (4 + (5 - 3) );
-- Evaluates to 2 * (4 + 2) which then evaluates to 2 * 6, and
-- yields an expression result of 12.
SELECT @MyNumber;
```

See Also

Logical Operators (Transact-SQL)
Operators (Transact-SQL)
Built-in Functions (Transact-SQL)

Transactions (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

A transaction is a single unit of work. If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database. If a transaction encounters errors and must be canceled or rolled back, then all of the data modifications are erased.

SQL Server operates in the following transaction modes:

Autocommit transactions
Each individual statement is a transaction.

Explicit transactions
Each transaction is explicitly started with the BEGIN TRANSACTION statement and explicitly ended with a COMMIT or ROLLBACK statement.

Implicit transactions
A new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT or ROLLBACK statement.

Batch-scoped transactions
Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.

> **NOTE**
> For special considerations related to Data Warehouse products, see Transactions (SQL Data Warehouse).

In This Section

SQL Server provides the following transaction statements:

BEGIN DISTRIBUTED TRANSACTION	ROLLBACK TRANSACTION
BEGIN TRANSACTION	ROLLBACK WORK
COMMIT TRANSACTION	SAVE TRANSACTION
COMMIT WORK	

See Also

SET IMPLICIT_TRANSACTIONS (Transact-SQL)
@@TRANCOUNT (Transact-SQL)

Transactions (SQL Data Warehouse)

7/4/2018 • 5 minutes to read • Edit Online

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

A transaction is a group of one or more database statements that are either wholly committed or wholly rolled back. Each transaction is atomic, consistent, isolated, and durable (ACID). If the transaction succeeds, all statements within it are committed. If the transaction fails, that is at least one of the statements in the group fails, then the entire group is rolled back.

The beginning and end of transactions depends on the AUTOCOMMIT setting and the BEGIN TRANSACTION, COMMIT, and ROLLBACK statements. SQL Data Warehouse supports the following types of transactions:

- *Explicit transactions* start with the BEGIN TRANSACTION statement and end with the COMMIT or ROLLBACK statement.
- *Auto-commit transactions* initiate automatically within a session and do not start with the BEGIN TRANSACTION statement. When the AUTOCOMMIT setting is ON, each statement runs in a transaction and no explicit COMMIT or ROLLBACK is necessary. When the AUTOCOMMIT setting is OFF, a COMMIT or ROLLBACK statement is required to determine the outcome of the transaction. In SQL Data Warehouse, autocommit transactions begin immediately after a COMMIT or ROLLBACK statement, or after a SET AUTOCOMMIT OFF statement.

Transact-SQL Syntax Conventions (Transact-SQL)

Syntax

```
BEGIN TRANSACTION [;]
COMMIT [ TRAN | TRANSACTION | WORK ] [;]
ROLLBACK [ TRAN | TRANSACTION | WORK ] [;]
SET AUTOCOMMIT { ON | OFF } [;]
SET IMPLICIT_TRANSACTIONS { ON | OFF } [;]
```

Arguments

BEGIN TRANSACTION
Marks the starting point of an explicit transaction.

COMMIT [WORK]
Marks the end of an explicit or autocommit transaction. This statement causes the changes in the transaction to be permanently committed to the database. The statement COMMIT is identical to COMMIT WORK, COMMIT TRAN, and COMMIT TRANSACTION.

ROLLBACK [WORK]
Rolls back a transaction to the beginning of the transaction. No changes for the transaction are committed to the database. The statement ROLLBACK is identical to ROLLBACK WORK, ROLLBACK TRAN, and ROLLBACK TRANSACTION.

SET AUTOCOMMIT { **ON** | OFF }
Determines how transactions can start and end.

ON

Each statement runs under its own transaction and no explicit COMMIT or ROLLBACK statement is necessary. Explicit transactions are allowed when AUTOCOMMIT is ON.

OFF
SQL Data Warehouse automatically initiates a transaction when a transaction is not already in progress. Any subsequent statements are run as part of the transaction and a COMMIT or ROLLBACK is necessary to determine the outcome of the transaction. As soon as a transaction commits or rolls back under this mode of operation, the mode remains OFF, and SQL Data Warehouse initiates a new transaction. Explicit transactions are not allowed when AUTOCOMMIT is OFF.

If you change the AUTOCOMMIT setting within an active transaction, the setting does affect the current transaction and does not take affect until the transaction is completed.

If AUTOCOMMIT is ON, running another SET AUTOCOMMIT ON statement has no effect. Likewise, if AUTOCOMMIT is OFF, running another SET AUTOCOMMIT OFF has no effect.

SET IMPLICIT_TRANSACTIONS { ON | **OFF** }
This toggles the same modes as SET AUTOCOMMIT. When ON, SET IMPLICIT_TRANSACTIONS sets the connection into implicit transaction mode. When OFF, it returns the connection to autocommit mode. For more information, see SET IMPLICIT_TRANSACTIONS (Transact-SQL).

Permissions

No specific permissions are required to run the transaction-related statements. Permissions are required to run the statements within the transaction.

Error Handling

If COMMIT or ROLLBACK are run and there is no active transaction, an error is raised.

If a BEGIN TRANSACTION is run while a transaction is already in progress, an error is raised. This can occur if a BEGIN TRANSACTION occurs after a successful BEGIN TRANSACTION statement or when the session is under SET AUTOCOMMIT OFF.

If an error other than a run-time statement error prevents the successful completion of an explicit transaction, SQL Data Warehouse automatically rolls back the transaction and frees all resources held by the transaction. For example, if the client's network connection to an instance of SQL Data Warehouse is broken or the client logs off the application, any uncommitted transactions for the connection are rolled back when the network notifies the instance of the break.

If a run-time statement error occurs in a batch, SQL Data Warehouse behaves consistent with SQL Server**XACT_ABORT** set to **ON** and the entire transaction is rolled back. For more information about the **XACT_ABORT** setting, see SET XACT_ABORT (Transact-SQL).

General Remarks

A session can only run one transaction at a given time; save points and nested transactions are not supported.

It is the responsibility of the SQL programmer to issue COMMIT only at a point when all data referenced by the transaction is logically correct.

When a session is terminated before a transaction completes, the transaction is rolled back.

Transaction modes are managed at the session level. For example, if one session begins an explicit transaction or sets AUTOCOMMIT to OFF, or sets IMPLICIT_TRANSACTIONS to ON, it has no effect on the transaction modes of any other session.

Limitations and Restrictions

You cannot roll back a transaction after a COMMIT statement is issued because the data modifications have been made a permanent part of the database.

The CREATE DATABASE (Azure SQL Data Warehouse) and DROP DATABASE (Transact-SQL) commands cannot be used inside an explicit transaction.

SQL Data Warehouse does not have a transaction sharing mechanism. This implies that at any given point in time, only one session can be doing work on any transaction in the system.

Locking Behavior

SQL Data Warehouse uses locking to ensure the integrity of transactions and maintain the consistency of databases when multiple users are accessing data at the same time. Locking is used by both implicit and explicit transactions. Each transaction requests locks of different types on the resources, such as tables or databases on which the transaction depends. All SQL Data Warehouse locks are table level or higher. The locks block other transactions from modifying the resources in a way that would cause problems for the transaction requesting the lock. Each transaction frees its locks when it no longer has a dependency on the locked resources; explicit transactions retain locks until the transaction completes when it is either committed or rolled back.

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

A. Using an explicit transaction

```
BEGIN TRANSACTION;
DELETE FROM HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
COMMIT;
```

B. Rolling back a transaction

The following example shows the effect of rolling back a transaction. In this example, the ROLLBACK statement will roll back the INSERT statement, but the created table will still exist.

```
CREATE TABLE ValueTable (id int);
BEGIN TRANSACTION;
       INSERT INTO ValueTable VALUES(1);
       INSERT INTO ValueTable VALUES(2);
ROLLBACK;
```

C. Setting AUTOCOMMIT

The following example sets the AUTOCOMMIT setting to `ON`.

```
SET AUTOCOMMIT ON;
```

The following example sets the AUTOCOMMIT setting to `OFF`.

```
SET AUTOCOMMIT OFF;
```

D. Using an implicit multi-statement transaction

```
SET AUTOCOMMIT OFF;
CREATE TABLE ValueTable (id int);
INSERT INTO ValueTable VALUES(1);
INSERT INTO ValueTable VALUES(2);
COMMIT;
```

See Also

SET IMPLICIT_TRANSACTIONS (Transact-SQL)
SET TRANSACTION ISOLATION LEVEL (Transact-SQL)
@@TRANCOUNT (Transact-SQL)

Transaction Isolation Levels

6/20/2018 • 2 minutes to read • Edit Online

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server does not guarantee that lock hints will be honored in queries that access metadata through catalog views, compatibility views, information schema views, metadata-emitting built-in functions.

Internally, the SQL Server Database Engine only honors the READ COMMITTED isolation level for metadata access. If a transaction has an isolation level that is, for example, SERIALIZABLE and within the transaction, an attempt is made to access metadata by using catalog views or metadata-emitting built-in functions, those queries will run until they are completed as READ COMMITTED. However, under snapshot isolation, access to metadata might fail because of concurrent DDL operations. This is because metadata is not versioned. Therefore, accessing the following under snapshot isolation might fail:

- Catalog views
- Compatibility views
- Information Schema Views
- Metadata-emitting built-in functions
- **sp_help** group of stored procedures
- SQL Server Native Client catalog procedures
- Dynamic management views and functions

For more information about isolation levels, see SET TRANSACTION ISOLATION LEVEL (Transact-SQL).

The following table provides a summary of metadata access under various isolation levels.

ISOLATION LEVEL	SUPPORTED	HONORED
READ UNCOMMITTED	No	Not guaranteed
READ COMMITTED	Yes	Yes
REPEATABLE READ	No	No
SNAPSHOT ISOLATION	No	No
SERIALIZABLE	No	No

BEGIN DISTRIBUTED TRANSACTION (Transact-SQL)

6/20/2018 • 3 minutes to read • Edit Online

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

Specifies the start of a Transact-SQL distributed transaction managed by Microsoft Distributed Transaction Coordinator (MS DTC).

Transact-SQL Syntax Conventions

Syntax

```
BEGIN DISTRIBUTED { TRAN | TRANSACTION }
     [ transaction_name | @tran_name_variable ]
[ ; ]
```

Arguments

transaction_name
Is a user-defined transaction name used to track the distributed transaction within MS DTC utilities. *transaction_name* must conform to the rules for identifiers and must be <= 32 characters.

@tran_name_variable
Is the name of a user-defined variable containing a transaction name used to track the distributed transaction within MS DTC utilities. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type.

Remarks

The instance of the SQL Server Database Engine executing the BEGIN DISTRIBUTED TRANSACTION statement is the transaction originator and controls the completion of the transaction. When a subsequent COMMIT TRANSACTION or ROLLBACK TRANSACTION statement is issued for the session, the controlling instance requests that MS DTC manage the completion of the distributed transaction across all of the instances involved.

Transaction-level snapshot isolation does not support distributed transactions.

The primary way remote instances of the Database Engine are enlisted in a distributed transaction is when a session already enlisted in the distributed transaction executes a distributed query referencing a linked server.

For example, if BEGIN DISTRIBUTED TRANSACTION is issued on ServerA, the session calls a stored procedure on ServerB and another stored procedure on ServerC. The stored procedure on ServerC executes a distributed query against ServerD, and then all four computers are involved in the distributed transaction. The instance of the Database Engine on ServerA is the originating controlling instance for the transaction.

The sessions involved in Transact-SQL distributed transactions do not get a transaction object they can pass to another session for it to explicitly enlist in the distributed transaction. The only way for a remote server to enlist in the transaction is to be the target of a distributed query or a remote stored procedure call.

When a distributed query is executed in a local transaction, the transaction is automatically promoted to a distributed transaction if the target OLE DB data source supports ITransactionLocal. If the target OLE DB data source does not support ITransactionLocal, only read-only operations are allowed in the distributed query.

A session already enlisted in the distributed transaction performs a remote stored procedure call referencing a remote server.

The **sp_configure remote proc trans** option controls whether calls to remote stored procedures in a local transaction automatically cause the local transaction to be promoted to a distributed transaction managed by MS DTC. The connection-level SET option REMOTE_PROC_TRANSACTIONS can be used to override the instance default established by **sp_configure remote proc trans**. With this option set on, a remote stored procedure call causes a local transaction to be promoted to a distributed transaction. The connection that creates the MS DTC transaction becomes the originator for the transaction. COMMIT TRANSACTION initiates an MS DTC coordinated commit. If the **sp_configure remote proc trans** option is ON, remote stored procedure calls in local transactions are automatically protected as part of distributed transactions without having to rewrite applications to specifically issue BEGIN DISTRIBUTED TRANSACTION instead of BEGIN TRANSACTION.

For more information about the distributed transaction environment and process, see the Microsoft Distributed Transaction Coordinator documentation.

Permissions

Requires membership in the public role.

Examples

This example deletes a candidate from the AdventureWorks2012 database on both the local instance of the Database Engine and an instance on a remote server. Both the local and remote databases will either commit or roll back the transaction.

> **NOTE**
> Unless MS DTC is currently installed on the computer running the instance of the Database Engine, this example produces an error message. For more information about installing MS DTC, see the Microsoft Distributed Transaction Coordinator documentation.

```
USE AdventureWorks2012;
GO
BEGIN DISTRIBUTED TRANSACTION;
-- Delete candidate from local instance.
DELETE AdventureWorks2012.HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
-- Delete candidate from remote instance.
DELETE RemoteServer.AdventureWorks2012.HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
COMMIT TRANSACTION;
GO
```

See Also

BEGIN TRANSACTION (Transact-SQL)
COMMIT TRANSACTION (Transact-SQL)
COMMIT WORK (Transact-SQL)
ROLLBACK TRANSACTION (Transact-SQL)
ROLLBACK WORK (Transact-SQL)
SAVE TRANSACTION (Transact-SQL)

BEGIN TRANSACTION (Transact-SQL)

8/27/2018 • 6 minutes to read • Edit Online

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

Marks the starting point of an explicit, local transaction. Explicit transactions start with the BEGIN TRANSACTION statement and end with the COMMIT or ROLLBACK statement.

Transact-SQL Syntax Conventions

Syntax

```
--Applies to SQL Server and Azure SQL Database

BEGIN { TRAN | TRANSACTION }
    [ { transaction_name | @tran_name_variable }
      [ WITH MARK [ 'description' ] ]
    ]
[ ; ]
```

```
--Applies to Azure SQL Data Warehouse and Parallel Data Warehouse

BEGIN { TRAN | TRANSACTION }
[ ; ]
```

Arguments

transaction_name
APPLIES TO: SQL Server (starting with 2008), Azure SQL Database

Is the name assigned to the transaction. *transaction_name* must conform to the rules for identifiers, but identifiers longer than 32 characters are not allowed. Use transaction names only on the outermost pair of nested BEGIN...COMMIT or BEGIN...ROLLBACK statements. *transaction_name* is always case sensitive, even when the instance of SQL Server is not case sensitive.

@tran_name_variable
APPLIES TO: SQL Server (starting with 2008), Azure SQL Database

Is the name of a user-defined variable containing a valid transaction name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type. If more than 32 characters are passed to the variable, only the first 32 characters will be used; the remaining characters will be truncated.

WITH MARK ['*description*']
APPLIES TO: SQL Server (starting with 2008), Azure SQL Database

Specifies that the transaction is marked in the log. *description* is a string that describes the mark. A *description* longer than 128 characters is truncated to 128 characters before being stored in the msdb.dbo.logmarkhistory table.

If WITH MARK is used, a transaction name must be specified. WITH MARK allows for restoring a transaction log to a named mark.

General Remarks

BEGIN TRANSACTION increments @@TRANCOUNT by 1.

BEGIN TRANSACTION represents a point at which the data referenced by a connection is logically and physically consistent. If errors are encountered, all data modifications made after the BEGIN TRANSACTION can be rolled back to return the data to this known state of consistency. Each transaction lasts until either it completes without errors and COMMIT TRANSACTION is issued to make the modifications a permanent part of the database, or errors are encountered and all modifications are erased with a ROLLBACK TRANSACTION statement.

BEGIN TRANSACTION starts a local transaction for the connection issuing the statement. Depending on the current transaction isolation level settings, many resources acquired to support the Transact-SQL statements issued by the connection are locked by the transaction until it is completed with either a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement. Transactions left outstanding for long periods of time can prevent other users from accessing these locked resources, and also can prevent log truncation.

Although BEGIN TRANSACTION starts a local transaction, it is not recorded in the transaction log until the application subsequently performs an action that must be recorded in the log, such as executing an INSERT, UPDATE, or DELETE statement. An application can perform actions such as acquiring locks to protect the transaction isolation level of SELECT statements, but nothing is recorded in the log until the application performs a modification action.

Naming multiple transactions in a series of nested transactions with a transaction name has little effect on the transaction. Only the first (outermost) transaction name is registered with the system. A rollback to any other name (other than a valid savepoint name) generates an error. None of the statements executed before the rollback is, in fact, rolled back at the time this error occurs. The statements are rolled back only when the outer transaction is rolled back.

The local transaction started by the BEGIN TRANSACTION statement is escalated to a distributed transaction if the following actions are performed before the statement is committed or rolled back:

- An INSERT, DELETE, or UPDATE statement that references a remote table on a linked server is executed. The INSERT, UPDATE, or DELETE statement fails if the OLE DB provider used to access the linked server does not support the ITransactionJoin interface.
- A call is made to a remote stored procedure when the REMOTE_PROC_TRANSACTIONS option is set to ON.

  The local copy of SQL Server becomes the transaction controller and uses Microsoft Distributed Transaction Coordinator (MS DTC) to manage the distributed transaction.

  A transaction can be explicitly executed as a distributed transaction by using BEGIN DISTRIBUTED TRANSACTION. For more information, see BEGIN DISTRIBUTED TRANSACTION (Transact-SQL).

  When SET IMPLICIT_TRANSACTIONS is set to ON, a BEGIN TRANSACTION statement creates two nested transactions. For more information see, SET IMPLICIT_TRANSACTIONS (Transact-SQL)

Marked Transactions

The WITH MARK option causes the transaction name to be placed in the transaction log. When restoring a database to an earlier state, the marked transaction can be used in place of a date and time. For more information, see Use Marked Transactions to Recover Related Databases Consistently (Full Recovery Model) and RESTORE (Transact-SQL).

Additionally, transaction log marks are necessary if you need to recover a set of related databases to a logically consistent state. Marks can be placed in the transaction logs of the related databases by a distributed transaction. Recovering the set of related databases to these marks results in a set of databases that are transactionally

consistent. Placement of marks in related databases requires special procedures.

The mark is placed in the transaction log only if the database is updated by the marked transaction. Transactions that do not modify data are not marked.

BEGIN TRAN *new_name* WITH MARK can be nested within an already existing transaction that is not marked. Upon doing so, *new_name* becomes the mark name for the transaction, despite the name that the transaction may already have been given. In the following example, `M2` is the name of the mark.

```
BEGIN TRAN T1;
UPDATE table1 ...;
BEGIN TRAN M2 WITH MARK;
UPDATE table2 ...;
SELECT * from table1;
COMMIT TRAN M2;
UPDATE table3 ...;
COMMIT TRAN T1;
```

When nesting transactions, trying to mark a transaction that is already marked results in a warning (not error) message:

"BEGIN TRAN T1 WITH MARK ...;"

"UPDATE table1 ...;"

"BEGIN TRAN M2 WITH MARK ...;"

"Server: Msg 3920, Level 16, State 1, Line 3"

"WITH MARK option only applies to the first BEGIN TRAN WITH MARK."

"The option is ignored."

Permissions

Requires membership in the public role.

Examples

A. Using an explicit transaction

APPLIES TO: SQL Server (starting with 2008), Azure SQL Database, Azure SQL Data Warehouse, Parallel Data Warehouse

This example uses AdventureWorks.

```
BEGIN TRANSACTION;
DELETE FROM HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
COMMIT;
```

B. Rolling back a transaction

APPLIES TO: SQL Server (starting with 2008), Azure SQL Database, Azure SQL Data Warehouse, Parallel Data Warehouse

The following example shows the effect of rolling back a transaction. In this example, the ROLLBACK statement will roll back the INSERT statement, but the created table will still exist.

```
CREATE TABLE ValueTable (id int);
BEGIN TRANSACTION;
       INSERT INTO ValueTable VALUES(1);
       INSERT INTO ValueTable VALUES(2);
ROLLBACK;
```

C. Naming a transaction

APPLIES TO: SQL Server (starting with 2008), Azure SQL Database

The following example shows how to name a transaction.

```
DECLARE @TranName VARCHAR(20);
SELECT @TranName = 'MyTransaction';

BEGIN TRANSACTION @TranName;
USE AdventureWorks2012;
DELETE FROM AdventureWorks2012.HumanResources.JobCandidate
    WHERE JobCandidateID = 13;

COMMIT TRANSACTION @TranName;
GO
```

D. Marking a transaction

APPLIES TO: SQL Server (starting with 2008), Azure SQL Database

The following example shows how to mark a transaction. The transaction `CandidateDelete` is marked.

```
BEGIN TRANSACTION CandidateDelete
    WITH MARK N'Deleting a Job Candidate';
GO
USE AdventureWorks2012;
GO
DELETE FROM AdventureWorks2012.HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
GO
COMMIT TRANSACTION CandidateDelete;
GO
```

See Also

BEGIN DISTRIBUTED TRANSACTION (Transact-SQL)
COMMIT TRANSACTION (Transact-SQL)
COMMIT WORK (Transact-SQL)
ROLLBACK TRANSACTION (Transact-SQL)
ROLLBACK WORK (Transact-SQL)
SAVE TRANSACTION (Transact-SQL)

COMMIT TRANSACTION (Transact-SQL)

8/27/2018 • 4 minutes to read • Edit Online

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

Marks the end of a successful implicit or explicit transaction. If @@TRANCOUNT is 1, COMMIT TRANSACTION makes all data modifications performed since the start of the transaction a permanent part of the database, frees the resources held by the transaction, and decrements @@TRANCOUNT to 0. If @@TRANCOUNT is greater than 1, COMMIT TRANSACTION decrements @@TRANCOUNT only by 1 and the transaction stays active.

Transact-SQL Syntax Conventions

Syntax

```
-- Applies to SQL Server (starting with 2008) and Azure SQL Database

COMMIT [ { TRAN | TRANSACTION }  [ transaction_name | @tran_name_variable ] ] [ WITH ( DELAYED_DURABILITY = {
OFF | ON } ) ]
[ ; ]
```

```
-- Applies to Azure SQL Data Warehouse and Parallel Data Warehouse Database

COMMIT [ TRAN | TRANSACTION ]
[ ; ]
```

Arguments

transaction_name
APPLIES TO: SQL Server and Azure SQL Database

Is ignored by the SQL Server Database Engine. *transaction_name* specifies a transaction name assigned by a previous BEGIN TRANSACTION. *transaction_name*must conform to the rules for identifiers, but cannot exceed 32 characters. *transaction_name* can be used as a readability aid by indicating to programmers which nested BEGIN TRANSACTION the COMMIT TRANSACTION is associated with.

@tran_name_variable
APPLIES TO: SQL Server and Azure SQL Database

Is the name of a user-defined variable containing a valid transaction name. The variable must be declared with a char, varchar, nchar, or nvarchar data type. If more than 32 characters are passed to the variable, only 32 characters will be used; the remaining characters are truncated.

DELAYED_DURABILITY
APPLIES TO: SQL Server and Azure SQL Database

Option that requests this transaction be committed with delayed durability. The request is ignored if the database has been altered with `DELAYED_DURABILITY = DISABLED` or `DELAYED_DURABILITY = FORCED`. See the topic Control Transaction Durability for more information.

Remarks

It is the responsibility of the Transact-SQL programmer to issue COMMIT TRANSACTION only at a point when all data referenced by the transaction is logically correct.

If the transaction committed was a Transact-SQL distributed transaction, COMMIT TRANSACTION triggers MS DTC to use a two-phase commit protocol to commit all of the servers involved in the transaction. If a local transaction spans two or more databases on the same instance of the Database Engine, the instance uses an internal two-phase commit to commit all of the databases involved in the transaction.

When used in nested transactions, commits of the inner transactions do not free resources or make their modifications permanent. The data modifications are made permanent and resources freed only when the outer transaction is committed. Each COMMIT TRANSACTION issued when @@TRANCOUNT is greater than 1 simply decrements @@TRANCOUNT by 1. When @@TRANCOUNT is finally decremented to 0, the entire outer transaction is committed. Because *transaction_name* is ignored by the Database Engine, issuing a COMMIT TRANSACTION referencing the name of an outer transaction when there are outstanding inner transactions only decrements @@TRANCOUNT by 1.

Issuing a COMMIT TRANSACTION when @@TRANCOUNT is 0 results in an error; there is no corresponding BEGIN TRANSACTION.

You cannot roll back a transaction after a COMMIT TRANSACTION statement is issued because the data modifications have been made a permanent part of the database.

The Database Engine increments the transaction count within a statement only when the transaction count is 0 at the start of the statement.

Permissions

Requires membership in the **public** role.

Examples

A. Committing a transaction

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

The following example deletes a job candidate. It uses AdventureWorks.

```
BEGIN TRANSACTION;
DELETE FROM HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
COMMIT TRANSACTION;
```

B. Committing a nested transaction

APPLIES TO: SQL Server and Azure SQL Database

The following example creates a table, generates three levels of nested transactions, and then commits the nested transaction. Although each `COMMIT TRANSACTION` statement has a *transaction_name* parameter, there is no relationship between the `COMMIT TRANSACTION` and `BEGIN TRANSACTION` statements. The *transaction_name* parameters are simply readability aids to help the programmer ensure that the proper number of commits are coded to decrement `@@TRANCOUNT` to 0 and thereby commit the outer transaction.

```
IF OBJECT_ID(N'TestTran',N'U') IS NOT NULL
    DROP TABLE TestTran;
GO
CREATE TABLE TestTran (Cola int PRIMARY KEY, Colb char(3));
GO
-- This statement sets @@TRANCOUNT to 1.
BEGIN TRANSACTION OuterTran;

PRINT N'Transaction count after BEGIN OuterTran = '
    + CAST(@@TRANCOUNT AS nvarchar(10));

INSERT INTO TestTran VALUES (1, 'aaa');

-- This statement sets @@TRANCOUNT to 2.
BEGIN TRANSACTION Inner1;

PRINT N'Transaction count after BEGIN Inner1 = '
    + CAST(@@TRANCOUNT AS nvarchar(10));

INSERT INTO TestTran VALUES (2, 'bbb');

-- This statement sets @@TRANCOUNT to 3.
BEGIN TRANSACTION Inner2;

PRINT N'Transaction count after BEGIN Inner2 = '
    + CAST(@@TRANCOUNT AS nvarchar(10));

INSERT INTO TestTran VALUES (3, 'ccc');

-- This statement decrements @@TRANCOUNT to 2.
-- Nothing is committed.
COMMIT TRANSACTION Inner2;

PRINT N'Transaction count after COMMIT Inner2 = '
    + CAST(@@TRANCOUNT AS nvarchar(10));

-- This statement decrements @@TRANCOUNT to 1.
-- Nothing is committed.
COMMIT TRANSACTION Inner1;

PRINT N'Transaction count after COMMIT Inner1 = '
    + CAST(@@TRANCOUNT AS nvarchar(10));

-- This statement decrements @@TRANCOUNT to 0 and
-- commits outer transaction OuterTran.
COMMIT TRANSACTION OuterTran;

PRINT N'Transaction count after COMMIT OuterTran = '
    + CAST(@@TRANCOUNT AS nvarchar(10));
```

See Also

BEGIN DISTRIBUTED TRANSACTION (Transact-SQL)
BEGIN TRANSACTION (Transact-SQL)
COMMIT WORK (Transact-SQL)
ROLLBACK TRANSACTION (Transact-SQL)
ROLLBACK WORK (Transact-SQL)
SAVE TRANSACTION (Transact-SQL)
@@TRANCOUNT (Transact-SQL)

COMMIT WORK (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Marks the end of a transaction.

Transact-SQL Syntax Conventions

Syntax

```
COMMIT [ WORK ]
[ ; ]
```

Remarks

This statement functions identically to COMMIT TRANSACTION, except COMMIT TRANSACTION accepts a user-defined transaction name. This COMMIT syntax, with or without specifying the optional keyword WORK, is compatible with SQL-92.

See Also

BEGIN DISTRIBUTED TRANSACTION (Transact-SQL)
BEGIN TRANSACTION (Transact-SQL)
COMMIT TRANSACTION (Transact-SQL)
ROLLBACK TRANSACTION (Transact-SQL)
ROLLBACK WORK (Transact-SQL)
SAVE TRANSACTION (Transact-SQL)
@@TRANCOUNT (Transact-SQL)

ROLLBACK TRANSACTION (Transact-SQL)

8/27/2018 • 4 minutes to read • Edit Online

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

Rolls back an explicit or implicit transaction to the beginning of the transaction, or to a savepoint inside the transaction. You can use ROLLBACK TRANSACTION to erase all data modifications made from the start of the transaction or to a savepoint. It also frees resources held by the transaction.

 Transact-SQL Syntax Conventions

Syntax

```
ROLLBACK { TRAN | TRANSACTION }
     [ transaction_name | @tran_name_variable
     | savepoint_name | @savepoint_variable ]
[ ; ]
```

Arguments

transaction_name
Is the name assigned to the transaction on BEGIN TRANSACTION. *transaction_name* must conform to the rules for identifiers, but only the first 32 characters of the transaction name are used. When nesting transactions, *transaction_name* must be the name from the outermost BEGIN TRANSACTION statement. *transaction_name* is always case-sensitive, even when the instance of SQL Server is not case-sensitive.

@ *tran_name_variable*
Is the name of a user-defined variable containing a valid transaction name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type.

savepoint_name
Is *savepoint_name* from a SAVE TRANSACTION statement. *savepoint_name* must conform to the rules for identifiers. Use *savepoint_name* when a conditional rollback should affect only part of the transaction.

@ *savepoint_variable*
Is name of a user-defined variable containing a valid savepoint name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type.

Error Handling

A ROLLBACK TRANSACTION statement does not produce any messages to the user. If warnings are needed in stored procedures or triggers, use the RAISERROR or PRINT statements. RAISERROR is the preferred statement for indicating errors.

General Remarks

ROLLBACK TRANSACTION without a *savepoint_name* or *transaction_name* rolls back to the beginning of the transaction. When nesting transactions, this same statement rolls back all inner transactions to the outermost BEGIN TRANSACTION statement. In both cases, ROLLBACK TRANSACTION decrements the @@TRANCOUNT system function to 0. ROLLBACK TRANSACTION *savepoint_name* does not decrement

@@TRANCOUNT.

ROLLBACK TRANSACTION cannot reference a *savepoint_name* in distributed transactions started either explicitly with BEGIN DISTRIBUTED TRANSACTION or escalated from a local transaction.

A transaction cannot be rolled back after a COMMIT TRANSACTION statement is executed, except when the COMMIT TRANSACTION is associated with a nested transaction that is contained within the transaction being rolled back. In this instance, the nested transaction is rolled back, even if you have issued a COMMIT TRANSACTION for it.

Within a transaction, duplicate savepoint names are allowed, but a ROLLBACK TRANSACTION using the duplicate savepoint name rolls back only to the most recent SAVE TRANSACTION using that savepoint name.

Interoperability

In stored procedures, ROLLBACK TRANSACTION statements without a *savepoint_name* or *transaction_name* roll back all statements to the outermost BEGIN TRANSACTION. A ROLLBACK TRANSACTION statement in a stored procedure that causes @@TRANCOUNT to have a different value when the stored procedure completes than the @@TRANCOUNT value when the stored procedure was called produces an informational message. This message does not affect subsequent processing.

If a ROLLBACK TRANSACTION is issued in a trigger:

- All data modifications made to that point in the current transaction are rolled back, including any made by the trigger.
- The trigger continues executing any remaining statements after the ROLLBACK statement. If any of these statements modify data, the modifications are not rolled back. No nested triggers are fired by the execution of these remaining statements.
- The statements in the batch after the statement that fired the trigger are not executed.

@@TRANCOUNT is incremented by one when entering a trigger, even when in autocommit mode. (The system treats a trigger as an implied nested transaction.)

ROLLBACK TRANSACTION statements in stored procedures do not affect subsequent statements in the batch that called the procedure; subsequent statements in the batch are executed. ROLLBACK TRANSACTION statements in triggers terminate the batch containing the statement that fired the trigger; subsequent statements in the batch are not executed.

The effect of a ROLLBACK on cursors is defined by these three rules:

1. With CURSOR_CLOSE_ON_COMMIT set ON, ROLLBACK closes, but does not deallocate all open cursors.
2. With CURSOR_CLOSE_ON_COMMIT set OFF, ROLLBACK does not affect any open synchronous STATIC or INSENSITIVE cursors or asynchronous STATIC cursors that have been fully populated. Open cursors of any other type are closed but not deallocated.
3. An error that terminates a batch and generates an internal rollback deallocates all cursors that were declared in the batch containing the error statement. All cursors are deallocated regardless of their type or the setting of CURSOR_CLOSE_ON_COMMIT. This includes cursors declared in stored procedures called by the error batch. Cursors declared in a batch before the error batch are subject to rules 1 and 2. A deadlock error is an example of this type of error. A ROLLBACK statement issued in a trigger also automatically generates this type of error.

Locking Behavior

A ROLLBACK TRANSACTION statement specifying a *savepoint_name* releases any locks that are acquired beyond the savepoint, with the exception of escalations and conversions. These locks are not released, and they are not converted back to their previous lock mode.

Permissions

Requires membership in the **public** role.

Examples

The following example shows the effect of rolling back a named transaction. After creating a table, the following statements start a named transaction, insert two rows, and then roll back the transaction named in the variable @TransactionName. Another statement outside of the named transaction inserts two rows. The query returns the results of the previous statements.

```
USE tempdb;
GO
CREATE TABLE ValueTable ([value] int);
GO

DECLARE @TransactionName varchar(20) = 'Transaction1';

BEGIN TRAN @TransactionName
       INSERT INTO ValueTable VALUES(1), (2);
ROLLBACK TRAN @TransactionName;

INSERT INTO ValueTable VALUES(3),(4);

SELECT [value] FROM ValueTable;

DROP TABLE ValueTable;
```

Here is the result set.

```
value
-----
3
4
```

See Also

BEGIN DISTRIBUTED TRANSACTION (Transact-SQL)
BEGIN TRANSACTION (Transact-SQL)
COMMIT TRANSACTION (Transact-SQL)
COMMIT WORK (Transact-SQL)
ROLLBACK WORK (Transact-SQL)
SAVE TRANSACTION (Transact-SQL)

ROLLBACK WORK (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Rolls back a user-specified transaction to the beginning of the transaction.

Transact-SQL Syntax Conventions

Syntax

```
ROLLBACK [ WORK ]
[ ; ]
```

Remarks

This statement functions identically to ROLLBACK TRANSACTION except that ROLLBACK TRANSACTION accepts a user-defined transaction name. With or without specifying the optional WORK keyword, this ROLLBACK syntax is ISO-compatible.

When nesting transactions, ROLLBACK WORK always rolls back to the outermost BEGIN TRANSACTION statement and decrements the @@TRANCOUNT system function to 0.

Permissions

ROLLBACK WORK permissions default to any valid user.

See Also

BEGIN DISTRIBUTED TRANSACTION (Transact-SQL)
BEGIN TRANSACTION (Transact-SQL)
COMMIT TRANSACTION (Transact-SQL)
COMMIT WORK (Transact-SQL)
ROLLBACK TRANSACTION (Transact-SQL)
SAVE TRANSACTION (Transact-SQL)

SAVE TRANSACTION (Transact-SQL)

6/20/2018 • 3 minutes to read • Edit Online

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

Sets a savepoint within a transaction.

Transact-SQL Syntax Conventions

.|

Syntax

```
SAVE { TRAN | TRANSACTION } { savepoint_name | @savepoint_variable }
[ ; ]
```

Arguments

savepoint_name
Is the name assigned to the savepoint. Savepoint names must conform to the rules for identifiers, but are limited to 32 characters. *savepoint_name* is always case sensitive, even when the instance of SQL Server is not case sensitive.

@savepoint_variable
Is the name of a user-defined variable containing a valid savepoint name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type. More than 32 characters can be passed to the variable, but only the first 32 characters will be used.

Remarks

A user can set a savepoint, or marker, within a transaction. The savepoint defines a location to which a transaction can return if part of the transaction is conditionally canceled. If a transaction is rolled back to a savepoint, it must proceed to completion with more Transact-SQL statements if needed and a COMMIT TRANSACTION statement, or it must be canceled altogether by rolling the transaction back to its beginning. To cancel an entire transaction, use the form ROLLBACK TRANSACTION *transaction_name*. All the statements or procedures of the transaction are undone.

Duplicate savepoint names are allowed in a transaction, but a ROLLBACK TRANSACTION statement that specifies the savepoint name will only roll the transaction back to the most recent SAVE TRANSACTION using that name.

SAVE TRANSACTION is not supported in distributed transactions started either explicitly with BEGIN DISTRIBUTED TRANSACTION or escalated from a local transaction.

> **IMPORTANT**
> A ROLLBACK TRANSACTION statement specifying a savepoint_name releases any locks that are acquired beyond the savepoint, with the exception of escalations and conversions. These locks are not released, and they are not converted back to their previous lock mode.

Permissions

Requires membership in the public role.

Examples

The following example shows how to use a transaction savepoint to roll back only the modifications made by a stored procedure if an active transaction is started before the stored procedure is executed.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.objects
           WHERE name = N'SaveTranExample')
    DROP PROCEDURE SaveTranExample;
GO
CREATE PROCEDURE SaveTranExample
    @InputCandidateID INT
AS
    -- Detect whether the procedure was called
    -- from an active transaction and save
    -- that for later use.
    -- In the procedure, @TranCounter = 0
    -- means there was no active transaction
    -- and the procedure started one.
    -- @TranCounter > 0 means an active
    -- transaction was started before the
    -- procedure was called.
    DECLARE @TranCounter INT;
    SET @TranCounter = @@TRANCOUNT;
    IF @TranCounter > 0
        -- Procedure called when there is
        -- an active transaction.
        -- Create a savepoint to be able
        -- to roll back only the work done
        -- in the procedure if there is an
        -- error.
        SAVE TRANSACTION ProcedureSave;
    ELSE
        -- Procedure must start its own
        -- transaction.
        BEGIN TRANSACTION;
    -- Modify database.
    BEGIN TRY
        DELETE HumanResources.JobCandidate
            WHERE JobCandidateID = @InputCandidateID;
        -- Get here if no errors; must commit
        -- any transaction started in the
        -- procedure, but not commit a transaction
        -- started before the transaction was called.
        IF @TranCounter = 0
            -- @TranCounter = 0 means no transaction was
            -- started before the procedure was called.
            -- The procedure must commit the transaction
            -- it started.
            COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        -- An error occurred; must determine
        -- which type of rollback will roll
        -- back only the work done in the
        -- procedure.
        IF @TranCounter = 0
            -- Transaction started in procedure.
            -- Roll back complete transaction.
            ROLLBACK TRANSACTION;
        ELSE
```

```
            -- Transaction started before procedure
            -- called, do not roll back modifications
            -- made before the procedure was called.
            IF XACT_STATE() <> -1
                -- If the transaction is still valid, just
                -- roll back to the savepoint set at the
                -- start of the stored procedure.
                ROLLBACK TRANSACTION ProcedureSave;
                -- If the transaction is uncommitable, a
                -- rollback to the savepoint is not allowed
                -- because the savepoint rollback writes to
                -- the log. Just return to the caller, which
                -- should roll back the outer transaction.

        -- After the appropriate rollback, echo error
        -- information to the caller.
        DECLARE @ErrorMessage NVARCHAR(4000);
        DECLARE @ErrorSeverity INT;
        DECLARE @ErrorState INT;

        SELECT @ErrorMessage = ERROR_MESSAGE();
        SELECT @ErrorSeverity = ERROR_SEVERITY();
        SELECT @ErrorState = ERROR_STATE();

        RAISERROR (@ErrorMessage, -- Message text.
                   @ErrorSeverity, -- Severity.
                   @ErrorState -- State.
                   );
    END CATCH
GO
```

See Also

BEGIN TRANSACTION (Transact-SQL)
COMMIT TRANSACTION (Transact-SQL)
COMMIT WORK (Transact-SQL)
ERROR_LINE (Transact-SQL)
ERROR_MESSAGE (Transact-SQL)
ERROR_NUMBER (Transact-SQL)
ERROR_PROCEDURE (Transact-SQL)
ERROR_SEVERITY (Transact-SQL)
ERROR_STATE (Transact-SQL)
RAISERROR (Transact-SQL)
ROLLBACK TRANSACTION (Transact-SQL)
ROLLBACK WORK (Transact-SQL)
TRY...CATCH (Transact-SQL)
XACT_STATE (Transact-SQL)

Variables (Transact-SQL)

8/27/2018 • 5 minutes to read • Edit Online

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

A Transact-SQL local variable is an object that can hold a single data value of a specific type. Variables in batches and scripts are typically used:

- As a counter either to count the number of times a loop is performed or to control how many times the loop is performed.
- To hold a data value to be tested by a control-of-flow statement.
- To save a data value to be returned by a stored procedure return code or function return value.

> **NOTE**
> The names of some Transact-SQL system functions begin with two *at* signs (@@). Although in earlier versions of SQL Server, the @@functions are referred to as global variables, they are not variables and do not have the same behaviors as variables. The @@functions are system functions, and their syntax usage follows the rules for functions.

The following script creates a small test table and populates it with 26 rows. The script uses a variable to do three things:

- Control how many rows are inserted by controlling how many times the loop is executed.
- Supply the value inserted into the integer column.
- Function as part of the expression that generates letters to be inserted into the character column.

```
-- Create the table.
CREATE TABLE TestTable (cola int, colb char(3));
GO
SET NOCOUNT ON;
GO
-- Declare the variable to be used.
DECLARE @MyCounter int;

-- Initialize the variable.
SET @MyCounter = 0;

-- Test the variable to see if the loop is finished.
WHILE (@MyCounter < 26)
BEGIN;
   -- Insert a row into the table.
   INSERT INTO TestTable VALUES
       -- Use the variable to provide the integer value
       -- for cola. Also use it to generate a unique letter
       -- for each row. Use the ASCII function to get the
       -- integer value of 'a'. Add @MyCounter. Use CHAR to
       -- convert the sum back to the character @MyCounter
       -- characters after 'a'.
       (@MyCounter,
        CHAR( ( @MyCounter + ASCII('a') ) )
       );
   -- Increment the variable to count this iteration
   -- of the loop.
   SET @MyCounter = @MyCounter + 1;
END;
GO
SET NOCOUNT OFF;
GO
-- View the data.
SELECT cola, colb
FROM TestTable;
GO
DROP TABLE TestTable;
GO
```

Declaring a Transact-SQL Variable

The DECLARE statement initializes a Transact-SQL variable by:

- Assigning a name. The name must have a single @ as the first character.
- Assigning a system-supplied or user-defined data type and a length. For numeric variables, a precision and scale are also assigned. For variables of type XML, an optional schema collection may be assigned.
- Setting the value to NULL.

For example, the following **DECLARE** statement creates a local variable named **@mycounter** with an int data type.

```
DECLARE @MyCounter int;
```

To declare more than one local variable, use a comma after the first local variable defined, and then specify the next local variable name and data type.

For example, the following **DECLARE** statement creates three local variables named **@LastName**, **@FirstName** and **@StateProvince**, and initializes each to NULL:

```
DECLARE @LastName nvarchar(30), @FirstName nvarchar(20), @StateProvince nchar(2);
```

The scope of a variable is the range of Transact-SQL statements that can reference the variable. The scope of a variable lasts from the point it is declared until the end of the batch or stored procedure in which it is declared. For example, the following script generates a syntax error because the variable is declared in one batch and referenced in another:

```
USE AdventureWorks2014;
GO
DECLARE @MyVariable int;
SET @MyVariable = 1;
-- Terminate the batch by using the GO keyword.
GO
-- @MyVariable has gone out of scope and no longer exists.

-- This SELECT statement generates a syntax error because it is
-- no longer legal to reference @MyVariable.
SELECT BusinessEntityID, NationalIDNumber, JobTitle
FROM HumanResources.Employee
WHERE BusinessEntityID = @MyVariable;
```

Variables have local scope and are only visible within the batch or procedure where they are defined. In the following example, the nested scope created for execution of sp_executesql does not have access to the variable declared in the higher scope and returns and error.

```
DECLARE @MyVariable int;
SET @MyVariable = 1;
EXECUTE sp_executesql N'SELECT @MyVariable'; -- this produces an error
```

Setting a Value in a Transact-SQL Variable

When a variable is first declared, its value is set to NULL. To assign a value to a variable, use the SET statement. This is the preferred method of assigning a value to a variable. A variable can also have a value assigned by being referenced in the select list of a SELECT statement.

To assign a variable a value by using the SET statement, include the variable name and the value to assign to the variable. This is the preferred method of assigning a value to a variable. The following batch, for example, declares two variables, assigns values to them, and then uses them in the `WHERE` clause of a `SELECT` statement:

```
USE AdventureWorks2014;
GO
-- Declare two variables.
DECLARE @FirstNameVariable nvarchar(50),
   @PostalCodeVariable nvarchar(15);

-- Set their values.
SET @FirstNameVariable = N'Amy';
SET @PostalCodeVariable = N'BA5 3HX';

-- Use them in the WHERE clause of a SELECT statement.
SELECT LastName, FirstName, JobTitle, City, StateProvinceName, CountryRegionName
FROM HumanResources.vEmployee
WHERE FirstName = @FirstNameVariable
   OR PostalCode = @PostalCodeVariable;
GO
```

A variable can also have a value assigned by being referenced in a select list. If a variable is referenced in a select

list, it should be assigned a scalar value or the SELECT statement should only return one row. For example:

```
USE AdventureWorks2014;
GO
DECLARE @EmpIDVariable int;

SELECT @EmpIDVariable = MAX(EmployeeID)
FROM HumanResources.Employee;
GO
```

WARNING

If there are multiple assignment clauses in a single SELECT statement, SQL Server does not guarantee the order of evaluation of the expressions. Note that effects are only visible if there are references among the assignments.

If a SELECT statement returns more than one row and the variable references a non-scalar expression, the variable is set to the value returned for the expression in the last row of the result set. For example, in the following batch **@EmpIDVariable** is set to the **BusinessEntityID** value of the last row returned, which is 1:

```
USE AdventureWorks2014;
GO
DECLARE @EmpIDVariable int;

SELECT @EmpIDVariable = BusinessEntityID
FROM HumanResources.Employee
ORDER BY BusinessEntityID DESC;

SELECT @EmpIDVariable;
GO
```

See Also

Declare @local_variable
SET @local_variable
SELECT @local_variable
Expressions (Transact-SQL)
Compound Operators (Transact-SQL)

SET (Transact-SQL)

8/27/2018 • 11 minutes to read • Edit Online

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

Sets the specified local variable, previously created by using the DECLARE @*local_variable* statement, to the specified value.

Transact-SQL Syntax Conventions

Syntax

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

SET
{ @local_variable
    [ . { property_name | field_name } ] = { expression | udt_name { . | :: } method_name }
}
|
{ @SQLCLR_local_variable.mutator_method
}
|
{ @local_variable
    {+= | -= | *= | /= | %= | &= | ^= | |= } expression
}
|
  { @cursor_variable =
    { @cursor_variable | cursor_name
    | { CURSOR [ FORWARD_ONLY | SCROLL ]
        [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
        [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
        [ TYPE_WARNING ]
    FOR select_statement
        [ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
      }
    }
}
```

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

SET @local_variable {+= | -= | *= | /= | %= | &= | ^= | |= } expression
```

Arguments

@ *local_variable*

Is the name of a variable of any type except **cursor**, **text**, **ntext**, **image**, or **table**. Variable names must start with one at sign (**@**). Variable names must comply with the rules for identifiers.

property_name

Is a property of a user-defined type.

field_name

Is a public field of a user-defined type.

udt_name
Is the name of a common language runtime (CLR) user-defined type.

{ **.** | **::** }
Specifies a method of a CLR user-define type. For an instance (non-static) method, use a period (**.**). For a static method, use two colons (**::**). To invoke a method, property, or field of a CLR user-defined type, you must have EXECUTE permission on the type.

method_name **(** *argument* [**,**... *n*] **)**
Is a method of a user-defined type that takes one or more arguments to modify the state of an instance of a type. Static methods must be public.

@ *SQLCLR_local_variable*
Is a variable whose type is located in an assembly. For more information, see Common Language Runtime (CLR) Integration Programming Concepts.

mutator_method
Is a method in the assembly that can change the state of the object. SQLMethodAttribute.IsMutator will be applied to this method.

{ **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **^=** | **|=** }
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

expression
Is any valid expression.

cursor_variable
Is the name of a cursor variable. If the target cursor variable previously referenced a different cursor, that previous reference is removed.

cursor_name
Is the name of a cursor declared by using the DECLARE CURSOR statement.

CURSOR
Specifies that the SET statement contains a declaration of a cursor.

SCROLL
Specifies that the cursor supports all fetch options: FIRST, LAST, NEXT, PRIOR, RELATIVE, and ABSOLUTE. SCROLL cannot be specified when FAST_FORWARD is also specified.

FORWARD_ONLY
Specifies that the cursor supports only the FETCH NEXT option. The cursor can be retrieved only in one direction, from the first to the last row. When FORWARD_ONLY is specified without the STATIC, KEYSET, or DYNAMIC keywords, the cursor is implemented as DYNAMIC. When neither FORWARD_ONLY nor SCROLL is specified,

FORWARD_ONLY is the default, unless the keywords STATIC, KEYSET, or DYNAMIC are specified. For STATIC, KEYSET, and DYNAMIC cursors, SCROLL is the default.

STATIC
Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in tempdb; therefore, modifications made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow for modifications.

KEYSET
Specifies that the membership and order of rows in the cursor are fixed when the cursor is opened. The set of keys that uniquely identify the rows is built into the keysettable in tempdb. Changes to nonkey values in the base tables, either made by the cursor owner or committed by other users, are visible as the cursor owner scrolls around the cursor. Inserts made by other users are not visible, and inserts cannot be made through a Transact-SQL server cursor.

If a row is deleted, an attempt to fetch the row returns an @@FETCH_STATUS of -2. Updates of key values from outside the cursor are similar to a delete of the old row followed by an insert of the new row. The row with the new values is not visible, and tries to fetch the row with the old values return an @@FETCH_STATUS of -2. The new values are visible if the update is performed through the cursor by specifying the WHERE CURRENT OF clause.

DYNAMIC
Defines a cursor that reflects all data changes made to the rows in its result set as the cursor owner scrolls around the cursor. The data values, order, and membership of the rows can change on each fetch. The absolute and relative fetch options are not supported with dynamic cursors.

FAST_FORWARD
Specifies a FORWARD_ONLY, READ_ONLY cursor with optimizations enabled. FAST_FORWARD cannot be specified when SCROLL is also specified.

READ_ONLY
Prevents updates from being made through this cursor. The cursor cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. This option overrides the default capability of a cursor to be updated.

SCROLL LOCKS
Specifies that positioned updates or deletes made through the cursor are guaranteed to succeed. SQL Server locks the rows as they are read into the cursor to guarantee their availability for later modifications. SCROLL_LOCKS cannot be specified when FAST_FORWARD is also specified.

OPTIMISTIC
Specifies that positioned updates or deletes made through the cursor do not succeed if the row has been updated since it was read into the cursor. SQL Server does not lock rows as they are read into the cursor. Instead, it uses comparisons of timestamp column values, or a checksum value if the table has no timestamp column, to determine whether the row was modified after it was read into the cursor. If the row was modified, the attempted positioned update or delete fails. OPTIMISTIC cannot be specified when FAST_FORWARD is also specified.

TYPE_WARNING
Specifies that a warning message is sent to the client when the cursor is implicitly converted from the requested type to another.

FOR *select_statement*
Is a standard SELECT statement that defines the result set of the cursor. The keywords FOR BROWSE, and INTO are not allowed within the *select_statement* of a cursor declaration.

If DISTINCT, UNION, GROUP BY, or HAVING are used, or an aggregate expression is included in the *select_list*, the cursor will be created as STATIC.

If each underlying tables does not have a unique index and an ISO SCROLL cursor or a Transact-SQL KEYSET cursor is requested, it will automatically be a STATIC cursor.

If *select_statement* contains an ORDER BY clause in which the columns are not unique row identifiers, a DYNAMIC cursor is converted to a KEYSET cursor, or to a STATIC cursor if a KEYSET cursor cannot be opened. This also occurs for a cursor defined by using ISO syntax but without the STATIC keyword.

READ ONLY
Prevents updates from being made through this cursor. The cursor cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. This option overrides the default capability of a cursor to be updated. This keyword varies from the earlier READ_ONLY by having a space instead of an underscore between READ and ONLY.

UPDATE [OF *column_name*[**,**... *n*]]
Defines updatable columns within the cursor. If OF *column_name* [**,**...*n*] is supplied, only the columns listed will allow modifications. If no list is supplied, all columns can be updated, unless the cursor has been defined as READ_ONLY.

Remarks

After a variable is declared, it is initialized to NULL. Use the SET statement to assign a value that is not NULL to a declared variable. The SET statement that assigns a value to the variable returns a single value. When you initialize multiple variables, use a separate SET statement for each local variable.

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

The syntax rules for SET **@***cursor_variable* do not include the LOCAL and GLOBAL keywords. When the SET **@***cursor_variable* = CURSOR... syntax is used, the cursor is created as GLOBAL or LOCAL, depending on the setting of the default to local cursor database option.

Cursor variables are always local, even if they reference a global cursor. When a cursor variable references a global cursor, the cursor has both a global and a local cursor reference. For more information, see Example C.

For more information, see DECLARE CURSOR (Transact-SQL).

The compound assignment operator can be used anywhere you have an assignment with an expression on the right hand side of the operator, including variables, and a SET in an UPDATE, SELECT and RECEIVE statement.

Do not use a variable in a SELECT statement to concatenate values (that is, to compute aggregate values). Unexpected query results may occur. This is because all expressions in the SELECT list (including assignments) are not guaranteed to be executed exactly once for each output row. For more information, see this KB article.

Permissions

Requires membership in the public role. All users can use SET **@***local_variable*.

Examples

A. Printing the value of a variable initialized by using SET

The following example creates the `@myvar` variable, puts a string value into the variable, and prints the value of the `@myvar` variable.

```
DECLARE @myvar char(20);
SET @myvar = 'This is a test';
SELECT @myvar;
GO
```

B. Using a local variable assigned a value by using SET in a SELECT statement

The following example creates a local variable named `@state` and uses this local variable in a `SELECT` statement to find the first and last names of all employees who reside in the state of `Oregon`.

```
USE AdventureWorks2012;
GO
DECLARE @state char(25);
SET @state = N'Oregon';
SELECT RTRIM(FirstName) + ' ' + RTRIM(LastName) AS Name, City
FROM HumanResources.vEmployee
WHERE StateProvinceName = @state;
```

C. Using a compound assignment for a local variable

The following two examples produce the same result. They create a local variable named `@NewBalance`, multiplies it by 10 and displays the new value of the local variable in a `SELECT` statement. The second example uses a compound assignment operator.

```
/* Example one */
DECLARE  @NewBalance  int ;
SET  @NewBalance  =  10;
SET  @NewBalance  =  @NewBalance  *  10;
SELECT  @NewBalance;

/* Example Two */
DECLARE @NewBalance int = 10;
SET @NewBalance *= 10;
SELECT @NewBalance;
```

D. Using SET with a global cursor

The following example creates a local variable and then sets the cursor variable to the global cursor name.

```
DECLARE my_cursor CURSOR GLOBAL
FOR SELECT * FROM Purchasing.ShipMethod
DECLARE @my_variable CURSOR ;
SET @my_variable = my_cursor ;
--There is a GLOBAL cursor declared(my_cursor) and a LOCAL variable
--(@my_variable) set to the my_cursor cursor.
DEALLOCATE my_cursor;
--There is now only a LOCAL variable reference
--(@my_variable) to the my_cursor cursor.
```

E. Defining a cursor by using SET

The following example uses the `SET` statement to define a cursor.

```
DECLARE @CursorVar CURSOR;

SET @CursorVar = CURSOR SCROLL DYNAMIC
FOR
SELECT LastName, FirstName
FROM AdventureWorks2012.HumanResources.vEmployee
WHERE LastName like 'B%';

OPEN @CursorVar;

FETCH NEXT FROM @CursorVar;
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM @CursorVar
END;

CLOSE @CursorVar;
DEALLOCATE @CursorVar;
```

F. Assigning a value from a query

The following example uses a query to assign a value to a variable.

```
USE AdventureWorks2012;
GO
DECLARE @rows int;
SET @rows = (SELECT COUNT(*) FROM Sales.Customer);
SELECT @rows;
```

G. Assigning a value to a user-defined type variable by modifying a property of the type

The following example sets a value for user-defined type `Point` by modifying the value of the property `X` of the type.

```
DECLARE @p Point;
SET @p.X = @p.X + 1.1;
SELECT @p;
GO
```

H. Assigning a value to a user-defined type variable by invoking a method of the type

The following example sets a value for user-defined type **point** by invoking method `SetXY` of the type.

```
DECLARE @p Point;
SET @p=point.SetXY(23.5, 23.5);
```

I. Creating a variable for a CLR type and calling a mutator method

The following example creates a variable for the type `Point`, and then executes a mutator method in `Point`.

```
CREATE ASSEMBLY mytest from 'c:\test.dll' WITH PERMISSION_SET = SAFE
CREATE TYPE Point EXTERNAL NAME mytest.Point
GO
DECLARE @p Point = CONVERT(Point, '')
SET @p.SetXY(22, 23);
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

J. Printing the value of a variable initialized by using SET

The following example creates the `@myvar` variable, puts a string value into the variable, and prints the value of the `@myvar` variable.

```
DECLARE @myvar char(20);
SET @myvar = 'This is a test';
SELECT top 1 @myvar FROM sys.databases;
```

K. Using a local variable assigned a value by using SET in a SELECT statement

The following example creates a local variable named `@dept` and uses this local variable in a `SELECT` statement to find the first and last names of all employees who work in the `Marketing` department.

```
-- Uses AdventureWorks

DECLARE @dept char(25);
SET @dept = N'Marketing';
SELECT RTRIM(FirstName) + ' ' + RTRIM(LastName) AS Name
FROM DimEmployee
WHERE DepartmentName = @dept;
```

L. Using a compound assignment for a local variable

The following two examples produce the same result. They create a local variable named `@NewBalance`, multiplies it by `10` and displays the new value of the local variable in a `SELECT` statement. The second example uses a compound assignment operator.

```
/* Example one */
DECLARE  @NewBalance  int ;
SET  @NewBalance  =  10;
SET  @NewBalance  =  @NewBalance  *  10;
SELECT  TOP 1 @NewBalance FROM sys.tables;

/* Example Two */
DECLARE @NewBalance int = 10;
SET @NewBalance *= 10;
SELECT TOP 1 @NewBalance FROM sys.tables;
```

M. Assigning a value from a query

The following example uses a query to assign a value to a variable.

```
-- Uses AdventureWorks

DECLARE @rows int;
SET @rows = (SELECT COUNT(*) FROM dbo.DimCustomer);
SELECT TOP 1 @rows FROM sys.tables;
```

See Also

Compound Operators (Transact-SQL)
DECLARE @local_variable (Transact-SQL)
EXECUTE (Transact-SQL)
SELECT (Transact-SQL)
SET Statements (Transact-SQL)

SELECT (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Sets a local variable to the value of an expression.

For assigning variables, we recommend that you use SET @local_variable instead of SELECT @*local_variable*.

Transact-SQL Syntax Conventions

Syntax

```
SELECT { @local_variable { = | += | -= | *= | /= | %= | &= | ^= | |= } expression }
    [ ,...n ] [ ; ]
```

Arguments

@local_variable
Is a declared variable for which a value is to be assigned.

{= | += | -= | *= | /= | %= | &= | ^= | |= }
Assign the value on the right to the variable on the left.

Compound assignment operator:

OPERATOR	ACTION
=	Assigns the expression that follows, to the variable.
+=	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

expression
Is any valid expression. This includes a scalar subquery.

Remarks

SELECT @*local_variable* is typically used to return a single value into the variable. However, when *expression* is the name of a column, it can return multiple values. If the SELECT statement returns more than one value, the variable is assigned the last value that is returned.

If the SELECT statement returns no rows, the variable retains its present value. If *expression* is a scalar subquery that returns no value, the variable is set to NULL.

One SELECT statement can initialize multiple local variables.

> **NOTE**
> A SELECT statement that contains a variable assignment cannot be used to also perform typical result set retrieval operations.

Examples

A. Use SELECT @local_variable to return a single value

In the following example, the variable `@var1` is assigned `Generic Name` as its value. The query against the `Store` table returns no rows because the value specified for `CustomerID` does not exist in the table. The variable retains the `Generic Name` value.

```
-- Uses AdventureWorks

DECLARE @var1 varchar(30);
SELECT @var1 = 'Generic Name';
SELECT @var1 = Name
FROM Sales.Store
WHERE CustomerID = 1000 ;
SELECT @var1 AS 'Company Name';
```

Here is the result set.

```
Company Name
------------------------------
Generic Name
```

B. Use SELECT @local_variable to return null

In the following example, a subquery is used to assign a value to `@var1`. Because the value requested for `CustomerID` does not exist, the subquery returns no value and the variable is set to `NULL`.

```
-- Uses AdventureWorks

DECLARE @var1 varchar(30)
SELECT @var1 = 'Generic Name'
SELECT @var1 = (SELECT Name
FROM Sales.Store
WHERE CustomerID = 1000)
SELECT @var1 AS 'Company Name' ;
```

Here is the result set.

```
Company Name
----------------------------
NULL
```

See Also

DECLARE @local_variable (Transact-SQL)
Expressions (Transact-SQL)
Compound Operators (Transact-SQL)
SELECT (Transact-SQL)

DECLARE (Transact-SQL)

8/27/2018 • 9 minutes to read • Edit Online

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

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

Transact-SQL Syntax Conventions

Syntax

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

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

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

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

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

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

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

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

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

Arguments

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

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

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

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

@cursor_variable_name
Is the name of a cursor variable. Cursor variable names must begin with an at (@) sign and conform to the rules for identifiers.

CURSOR
Specifies that the variable is a local cursor variable.

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

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

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

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

column_name
Is the name of the column in the table.

scalar_data_type
Specifies that the column is a scalar data type.

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

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

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

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

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

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

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

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

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

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

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

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

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

logical_expression
Is a logical expression that returns TRUE or FALSE.

Remarks

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

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

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

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

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

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

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

  A cursor variable:

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

Examples

A. Using DECLARE

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

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

Here is the result set.

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

(3 row(s) affected)
```

B. Using DECLARE with two variables

The following example retrieves the names of Adventure Works Cycles sales representatives who are located in the North American sales territory and have at least $2,000,000 in sales for the year.

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

C. Declaring a variable of type table

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

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

D. Declaring a variable of user-defined table type

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

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

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

E. Using DECLARE

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

```
-- Uses AdventureWorks

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

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

F. Using DECLARE with two variables

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

```
-- Uses AdventureWorks

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

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

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

See Also

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

EXECUTE (Transact-SQL)

8/27/2018 • 25 minutes to read • Edit Online

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

Executes a command string or character string within a Transact-SQL batch, or one of the following modules: system stored procedure, user-defined stored procedure, CLR stored procedure, scalar-valued user-defined function, or extended stored procedure. The EXECUTE statement can be used to send pass-through commands to linked servers. Additionally, the context in which a string or command is executed can be explicitly set. Metadata for the result set can be defined by using the WITH RESULT SETS options.

> **IMPORTANT**
> Before you call EXECUTE with a character string, validate the character string. Never execute a command constructed from user input that has not been validated.

 Transact-SQL Syntax Conventions

Syntax

```
-- Syntax for SQL Server

Execute a stored procedure or function
[ { EXEC | EXECUTE } ]
    {
      [ @return_status = ]
      { module_name [ ;number ] | @module_name_var }
        [ [ @parameter = ] { value
                           | @variable [ OUTPUT ]
                           | [ DEFAULT ]
                           }
        ]
      [ ,...n ]
      [ WITH <execute_option> [ ,...n ] ]
    }
[;]

Execute a character string
{ EXEC | EXECUTE }
    ( { @string_variable | [ N ]'tsql_string' } [ + ...n ] )
    [ AS { LOGIN | USER } = ' name ' ]
[;]

Execute a pass-through command against a linked server
{ EXEC | EXECUTE }
    ( { @string_variable | [ N ] 'command_string [ ? ]' } [ + ...n ]
        [ { , { value | @variable [ OUTPUT ] } } [ ...n ] ]
    )
    [ AS { LOGIN | USER } = ' name ' ]
    [ AT linked_server_name ]
[;]

<execute_option>::=
{
        RECOMPILE
    | { RESULT SETS UNDEFINED }
    | { RESULT SETS NONE }
    | { RESULT SETS ( <result_sets_definition> [,...n ] ) }
}

<result_sets_definition> ::=
{
    (
         { column_name
           data_type
         [ COLLATE collation_name ]
         [ NULL | NOT NULL ] }
         [,...n ]
    )
    | AS OBJECT
        [ db_name . [ schema_name ] . | schema_name . ]
        {table_name | view_name | table_valued_function_name }
    | AS TYPE [ schema_name.]table_type_name
    | AS FOR XML
}
```

```
-- In-Memory OLTP

Execute a natively compiled, scalar user-defined function
[ { EXEC | EXECUTE } ]
    {
      [ @return_status = ]
      { module_name | @module_name_var }
        [ [ @parameter = ] { value
                           | @variable
                           | [ DEFAULT ]
                           }
        ]
      [ ,...n ]
      [ WITH <execute_option> [ ,...n ] ]
    }
<execute_option>::=
{
    | { RESULT SETS UNDEFINED }
    | { RESULT SETS NONE }
    | { RESULT SETS ( <result_sets_definition> [,...n ] ) }
}
```

```
-- Syntax for Azure SQL Database

Execute a stored procedure or function
[ { EXEC | EXECUTE } ]
    {
      [ @return_status = ]
      { module_name  | @module_name_var }
        [ [ @parameter = ] { value
                           | @variable [ OUTPUT ]
                           | [ DEFAULT ]
                           }
        ]
      [ ,...n ]
      [ WITH RECOMPILE ]
    }
[;]

Execute a character string
{ EXEC | EXECUTE }
    ( { @string_variable | [ N ]'tsql_string' } [ + ...n ] )
    [ AS {  USER } = ' name ' ]
[;]

<execute_option>::=
{
        RECOMPILE
    | { RESULT SETS UNDEFINED }
    | { RESULT SETS NONE }
    | { RESULT SETS ( <result_sets_definition> [,...n ] ) }
}

<result_sets_definition> ::=
{
    (
         { column_name
           data_type
         [ COLLATE collation_name ]
         [ NULL | NOT NULL ] }
         [,...n ]
    )
    | AS OBJECT
        [ db_name . [ schema_name ] . | schema_name . ]
        {table_name | view_name | table_valued_function_name }
    | AS TYPE [ schema_name.]table_type_name
    | AS FOR XML
```

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

-- Execute a stored procedure
[ { EXEC | EXECUTE } ]
    procedure_name
        [ { value | @variable [ OUT | OUTPUT ] } ] [ ,...n ] }
[;]

-- Execute a SQL string
{ EXEC | EXECUTE }
    ( { @string_variable | [ N ] 'tsql_string' } [ +...n ] )
[;]
```

Arguments

@return_status

Is an optional integer variable that stores the return status of a module. This variable must be declared in the

batch, stored procedure, or function before it is used in an EXECUTE statement.

When used to invoke a scalar-valued user-defined function, the @*return_status* variable can be of any scalar data type.

module_name
Is the fully qualified or nonfully qualified name of the stored procedure or scalar-valued user-defined function to call. Module names must comply with the rules for identifiers. The names of extended stored procedures are always case-sensitive, regardless of the collation of the server.

A module that has been created in another database can be executed if the user running the module owns the module or has the appropriate permission to execute it in that database. A module can be executed on another server running SQL Server if the user running the module has the appropriate permission to use that server (remote access) and to execute the module in that database. If a server name is specified but no database name is specified, the SQL Server Database Engine looks for the module in the default database of the user.

;*number*
Applies to: SQL Server 2008 through SQL Server 2017

Is an optional integer that is used to group procedures of the same name. This parameter is not used for extended stored procedures.

> **NOTE**
> This feature is in maintenance mode and may 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.

For more information about procedure groups, see CREATE PROCEDURE (Transact-SQL).

@*module_name_var*
Is the name of a locally defined variable that represents a module name.

This can be a variable that holds the name of a natively compiled, scalar user-defined function.

@*parameter*
Is the parameter for *module_name*, as defined in the module. Parameter names must be preceded by the at sign (@). When used with the @*parameter_name*=*value* form, parameter names and constants do not have to be supplied in the order in which they are defined in the module. However, if the @*parameter_name*=*value* form is used for any parameter, it must be used for all subsequent parameters.

By default, parameters are nullable.

value
Is the value of the parameter to pass to the module or pass-through command. If parameter names are not specified, parameter values must be supplied in the order defined in the module.

When executing pass-through commands against linked servers, the order of the parameter values depends on the OLE DB provider of the linked server. Most OLE DB providers bind values to parameters from left to right.

If the value of a parameter is an object name, character string, or qualified by a database name or schema name, the whole name must be enclosed in single quotation marks. If the value of a parameter is a keyword, the keyword must be enclosed in double quotation marks.

If a default is defined in the module, a user can execute the module without specifying a parameter.

The default can also be NULL. Generally, the module definition specifies the action that should be taken if a parameter value is NULL.

@*variable*

Is the variable that stores a parameter or a return parameter.

OUTPUT
Specifies that the module or command string returns a parameter. The matching parameter in the module or command string must also have been created by using the keyword OUTPUT. Use this keyword when you use cursor variables as parameters.

If *value* is defined as OUTPUT of a module executed against a linked server, any changes to the corresponding @*parameter* performed by the OLE DB provider will be copied back to the variable at the end of the execution of module.

If OUTPUT parameters are being used and the intent is to use the return values in other statements within the calling batch or module, the value of the parameter must be passed as a variable, such as @*parameter* = @*variable*. You cannot execute a module by specifying OUTPUT for a parameter that is not defined as an OUTPUT parameter in the module. Constants cannot be passed to module by using OUTPUT; the return parameter requires a variable name. The data type of the variable must be declared and a value assigned before executing the procedure.

When EXECUTE is used against a remote stored procedure, or to execute a pass-through command against a linked server, OUTPUT parameters cannot be any one of the large object (LOB) data types.

Return parameters can be of any data type except the LOB data types.

DEFAULT
Supplies the default value of the parameter as defined in the module. When the module expects a value for a parameter that does not have a defined default and either a parameter is missing or the DEFAULT keyword is specified, an error occurs.

@*string_variable*
Is the name of a local variable. @*string_variable* can be any **char**, **varchar**, **nchar**, or **nvarchar** data type. These include the **(max)** data types.

[N] '*tsql_string*'
Is a constant string. *tsql_string* can be any **nvarchar** or **varchar** data type. If the N is included, the string is interpreted as **nvarchar** data type.

AS <context_specification>
Specifies the context in which the statement is executed.

LOGIN
Applies to: SQL Server 2008 through SQL Server 2017

Specifies the context to be impersonated is a login. The scope of impersonation is the server.

USER
Specifies the context to be impersonated is a user in the current database. The scope of impersonation is restricted to the current database. A context switch to a database user does not inherit the server-level permissions of that user.

> **IMPORTANT**
> While the context switch to the database user is active, any attempt to access resources outside the database will cause the statement to fail. This includes USE *database* statements, distributed queries, and queries that reference another database by using three- or four-part identifiers.

'*name*'
Is a valid user or login name. *name* must be a member of the sysadmin fixed server role or exist as a principal in

sys.database_principals or sys.server_principals, respectively.

name cannot be a built-in account, such as NT AUTHORITY\LocalService, NT AUTHORITY\NetworkService, or NT AUTHORITY\LocalSystem.

For more information, see Specifying a User or Login Name later in this topic.

[N] '*command_string*'
Is a constant string that contains the command to be passed through to the linked server. If the N is included, the string is interpreted as **nvarchar** data type.

[?]
Indicates parameters for which values are supplied in the <arg-list> of pass-through commands that are used in an EXEC('...', <arg-list>) AT <linkedsrv> statement.

AT *linked_server_name*
Applies to: SQL Server 2008 through SQL Server 2017

Specifies that *command_string* is executed against *linked_server_name* and results, if any, are returned to the client. *linked_server_name* must refer to an existing linked server definition in the local server. Linked servers are defined by using sp_addlinkedserver.

WITH <execute_option>
Possible execute options. The RESULT SETS options cannot be specified in an INSERT...EXEC statement.

TERM	DEFINITION
RECOMPILE	Forces a new plan to be compiled, used, and discarded after the module is executed. If there is an existing query plan for the module, this plan remains in the cache. Use this option if the parameter you are supplying is atypical or if the data has significantly changed. This option is not used for extended stored procedures. We recommend that you use this option sparingly because it is expensive. **Note:** You can not use WITH RECOMPILE when calling a stored procedure that uses OPENDATASOURCE syntax. The WITH RECOMPILE option is ignored when a four-part object name is specified. **Note:** RECOMPILE is not supported with natively compiled, scalar user-defined functions. If you need to recompile, use sp_recompile (Transact-SQL).
RESULT SETS UNDEFINED	**Applies to**: SQL Server 2012 (11.x) through SQL Server 2017, Azure SQL Database. This option provides no guarantee of what results, if any, will be returned, and no definition is provided. The statement executes without error if any results are returned or no results are returned. RESULT SETS UNDEFINED is the default behavior if a result_sets_option is not provided. For interpreted scalar user-defined functions, and natively compiled scalar user-defined functions, this option is not operational because the functions never return a result set.

TERM	DEFINITION
RESULT SETS NONE	**Applies to**: SQL Server 2012 (11.x) through SQL Server 2017, Azure SQL Database. Guarantees that the execute statement will not return any results. If any results are returned the batch is aborted. For interpreted scalar user-defined functions, and natively compiled scalar user-defined functions, this option is not operational because the functions never return a result set.
<result_sets_definition>	**Applies to**: SQL Server 2012 (11.x) through SQL Server 2017, Azure SQL Database. Provides a guarantee that the result will come back as specified in the result_sets_definition. For statements that return multiple result sets, provide multiple *result_sets_definition* sections. Enclose each *result_sets_definition* in parentheses, separated by commas. For more information, see <result_sets_definition> later in this topic. This option always results in an error for natively compiled, scalar user-defined functions because the functions never return a result set.

<result_sets_definition> **Applies to**: SQL Server 2012 (11.x) through SQL Server 2017, Azure SQL Database

Describes the result sets returned by the executed statements. The clauses of the result_sets_definition have the following meaning

TERM	DEFINITION

{
 	
column_name
 	
data_type
 	
[COLLATE collation_name]
 	
[NULL | NOT NULL]
 	
}
	See the table below.

db_name	The name of the database containing the table, view or table valued function.
schema_name	The name of the schema owning the table, view or table valued function.
table_name | view_name | table_valued_function_name	Specifies that the columns returned will be those specified in the table, view or table valued function named. Table variables, temporary tables, and synonyms are not supported in the AS object syntax.
AS TYPE [schema_name.]table_type_name	Specifies that the columns returned will be those specified in the table type.

TERM	DEFINITION
AS FOR XML	Specifies that the XML results from the statement or stored procedure called by the EXECUTE statement will be converted into the format as though they were produced by a SELECT ... FOR XML ... statement. All formatting from the type directives in the original statement are removed, and the results returned are as though no type directive was specified. AS FOR XML does not convert non-XML tabular results from the executed statement or stored procedure into XML.

TERM	DEFINITION
column_name	The names of each column. If the number of columns differs from the result set, an error occurs and the batch is aborted. If the name of a column differs from the result set, the column name returned will be set to the name defined.
data_type	The data types of each column. If the data types differ, an implicit conversion to the defined data type is performed. If the conversion fails the batch is aborted
COLLATE collation_name	The collation of each column. If there is a collation mismatch, an implicit collation is attempted. If that fails, the batch is aborted.
NULL | NOT NULL	The nullability of each column. If the defined nullability is NOT NULL and the data returned contains NULLs an error occurs and the batch is aborted. If not specified, the default value conforms to the setting of the ANSI_NULL_DFLT_ON and ANSI_NULL_DFLT_OFF options.

The actual result set being returned during execution can differ from the result defined using the WITH RESULT SETS clause in one of the following ways: number of result sets, number of columns, column name, nullability, and data type. If the number of result sets differs, an error occurs and the batch is aborted.

Remarks

Parameters can be supplied either by using *value* or by using @*parameter_name*=*value*. A parameter is not part of a transaction; therefore, if a parameter is changed in a transaction that is later rolled back, the value of the parameter does not revert to its previous value. The value returned to the caller is always the value at the time the module returns.

Nesting occurs when one module calls another or executes managed code by referencing a common language runtime (CLR) module, user-defined type, or aggregate. The nesting level is incremented when the called module or managed code reference starts execution, and it is decremented when the called module or managed code reference has finished. Exceeding the maximum of 32 nesting levels causes the complete calling chain to fail. The current nesting level is stored in the @@NESTLEVEL system function.

Because remote stored procedures and extended stored procedures are not within the scope of a transaction (unless issued within a BEGIN DISTRIBUTED TRANSACTION statement or when used with various configuration options), commands executed through calls to them cannot be rolled back. For more information, see System Stored Procedures (Transact-SQL) and BEGIN DISTRIBUTED TRANSACTION (Transact-SQL).

When you use cursor variables, if you execute a procedure that passes in a cursor variable with a cursor allocated to it an error occurs.

You do not have to specify the EXECUTE keyword when executing modules if the statement is the first one in a batch.

For additional information specific to CLR stored procedures, see CLR Stored Procedures.

Using EXECUTE with Stored Procedures

You do not have to specify the EXECUTE keyword when you execute stored procedures when the statement is the first one in a batch.

SQL Server system stored procedures start with the characters sp_. They are physically stored in the Resource database, but logically appear in the sys schema of every system and user-defined database. When you execute a system stored procedure, either in a batch or inside a module such as a user-defined stored procedure or function, we recommend that you qualify the stored procedure name with the sys schema name.

SQL Server system extended stored procedures start with the characters xp_, and these are contained in the dbo schema of the master database. When you execute a system extended stored procedure, either in a batch or inside a module such as a user-defined stored procedure or function, we recommend that you qualify the stored procedure name with master.dbo.

When you execute a user-defined stored procedure, either in a batch or inside a module such as a user-defined stored procedure or function, we recommend that you qualify the stored procedure name with a schema name. We do not recommend that you name a user-defined stored procedure with the same name as a system stored procedure. For more information about executing stored procedures, see Execute a Stored Procedure.

Using EXECUTE with a Character String

In earlier versions of SQL Server, character strings are limited to 8,000 bytes. This requires concatenating large strings for dynamic execution. In SQL Server, the **varchar(max)** and **nvarchar(max)** data types can be specified that allow for character strings to be up to 2 gigabytes of data.

Changes in database context last only until the end of the EXECUTE statement. For example, after the `EXEC` in this following statement is run, the database context is master.

```
USE master; EXEC ('USE AdventureWorks2012; SELECT BusinessEntityID, JobTitle FROM HumanResources.Employee;');
```

Context Switching

You can use the `AS { LOGIN | USER } = ' name '` clause to switch the execution context of a dynamic statement. When the context switch is specified as `EXECUTE ('string') AS <context_specification>`, the duration of the context switch is limited to the scope of the query being executed.

Specifying a User or Login Name

The user or login name specified in `AS { LOGIN | USER } = ' name '` must exist as a principal in sys.database_principals or sys.server_principals, respectively, or the statement will fail. Additionally, IMPERSONATE permissions must be granted on the principal. Unless the caller is the database owner or is a member of the sysadmin fixed server role, the principal must exist even when the user is accessing the database or instance of SQL Server through a Windows group membership. For example, assume the following conditions:

- CompanyDomain\SQLUsers group has access to the Sales database.
- CompanyDomain\SqlUser1 is a member of SQLUsers and, therefore, has implicit access to the Sales database.

  Although CompanyDomain\SqlUser1 has access to the database through membership in the SQLUsers

group, the statement `EXECUTE @string_variable AS USER = 'CompanyDomain\SqlUser1'` will fail because `CompanyDomain\SqlUser1` does not exist as a principal in the database.

Best Practices

Specify a login or user that has the least privileges required to perform the operations that are defined in the statement or module. For example, do not specify a login name, which has server-level permissions, if only database-level permissions are required; or do not specify a database owner account unless those permissions are required.

Permissions

Permissions are not required to run the EXECUTE statement. However, permissions are required on the securables that are referenced within the EXECUTE string. For example, if the string contains an INSERT statement, the caller of the EXECUTE statement must have INSERT permission on the target table. Permissions are checked at the time EXECUTE statement is encountered, even if the EXECUTE statement is included within a module.

EXECUTE permissions for a module default to the owner of the module, who can transfer them to other users. When a module is run that executes a string, permissions are checked in the context of the user who executes the module, not in the context of the user who created the module. However, if the same user owns the calling module and the module being called, EXECUTE permission checking is not performed for the second module.

If the module accesses other database objects, execution succeeds when you have EXECUTE permission on the module and one of the following is true:

- The module is marked EXECUTE AS USER or SELF, and the module owner has the corresponding permissions on the referenced object. For more information about impersonation within a module, see EXECUTE AS Clause (Transact-SQL).
- The module is marked EXECUTE AS CALLER, and you have the corresponding permissions on the object.
- The module is marked EXECUTE AS *user_name*, and *user_name* has the corresponding permissions on the object.

Context Switching Permissions

To specify EXECUTE AS on a login, the caller must have IMPERSONATE permissions on the specified login name. To specify EXECUTE AS on a database user, the caller must have IMPERSONATE permissions on the specified user name. When no execution context is specified, or EXECUTE AS CALLER is specified, IMPERSONATE permissions are not required.

Examples

A. Using EXECUTE to pass a single parameter

The `uspGetEmployeeManagers` stored procedure in the AdventureWorks2012 database expects one parameter (`@EmployeeID`). The following examples execute the `uspGetEmployeeManagers` stored procedure with `Employee ID 6` as its parameter value.

```
EXEC dbo.uspGetEmployeeManagers 6;
GO
```

The variable can be explicitly named in the execution:

```
EXEC dbo.uspGetEmployeeManagers @EmployeeID = 6;
GO
```

If the following is the first statement in a batch or an **osql** or **sqlcmd** script, EXEC is not required.

```
dbo.uspGetEmployeeManagers 6;
GO
--Or
dbo.uspGetEmployeeManagers @EmployeeID = 6;
GO
```

B. Using multiple parameters

The following example executes the `spGetWhereUsedProductID` stored procedure in the AdventureWorks2012 database. It passes two parameters: the first parameter is a product ID (`819`) and the second parameter, `@CheckDate,` is a `datetime` value.

```
DECLARE @CheckDate datetime;
SET @CheckDate = GETDATE();
EXEC dbo.uspGetWhereUsedProductID 819, @CheckDate;
GO
```

C. Using EXECUTE 'tsql_string' with a variable

The following example shows how `EXECUTE` handles dynamically built strings that contain variables. This example creates the `tables_cursor` cursor to hold a list of all user-defined tables in the **AdventureWorks2012** database, and then uses that list to rebuild all indexes on the tables.

```
DECLARE tables_cursor CURSOR
   FOR
   SELECT s.name, t.name
   FROM sys.objects AS t
   JOIN sys.schemas AS s ON s.schema_id = t.schema_id
   WHERE t.type = 'U';
OPEN tables_cursor;
DECLARE @schemaname sysname;
DECLARE @tablename sysname;
FETCH NEXT FROM tables_cursor INTO @schemaname, @tablename;
WHILE (@@FETCH_STATUS <> -1)
BEGIN;
   EXECUTE ('ALTER INDEX ALL ON ' + @schemaname + '.' + @tablename + ' REBUILD;');
   FETCH NEXT FROM tables_cursor INTO @schemaname, @tablename;
END;
PRINT 'The indexes on all tables have been rebuilt.';
CLOSE tables_cursor;
DEALLOCATE tables_cursor;
GO
```

D. Using EXECUTE with a remote stored procedure

The following example executes the `uspGetEmployeeManagers` stored procedure on the remote server `SQLSERVER1` and stores the return status that indicates success or failure in `@retstat` .

Applies to: SQL Server 2008 through SQL Server 2017

```
DECLARE @retstat int;
EXECUTE @retstat = SQLSERVER1.AdventureWorks2012.dbo.uspGetEmployeeManagers @BusinessEntityID = 6;
```

E. Using EXECUTE with a stored procedure variable

The following example creates a variable that represents a stored procedure name.

```
DECLARE @proc_name varchar(30);
SET @proc_name = 'sys.sp_who';
EXEC @proc_name;
```

F. Using EXECUTE with DEFAULT

The following example creates a stored procedure with default values for the first and third parameters. When the procedure is run, these defaults are inserted for the first and third parameters when no value is passed in the call or when the default is specified. Note the various ways the `DEFAULT` keyword can be used.

```
IF OBJECT_ID(N'dbo.ProcTestDefaults', N'P')IS NOT NULL
   DROP PROCEDURE dbo.ProcTestDefaults;
GO
-- Create the stored procedure.
CREATE PROCEDURE dbo.ProcTestDefaults (
@p1 smallint = 42,
@p2 char(1),
@p3 varchar(8) = 'CAR')
AS
   SET NOCOUNT ON;
   SELECT @p1, @p2, @p3
;
GO
```

The `Proc_Test_Defaults` stored procedure can be executed in many combinations.

```
-- Specifying a value only for one parameter (@p2).
EXECUTE dbo.ProcTestDefaults @p2 = 'A';
-- Specifying a value for the first two parameters.
EXECUTE dbo.ProcTestDefaults 68, 'B';
-- Specifying a value for all three parameters.
EXECUTE dbo.ProcTestDefaults 68, 'C', 'House';
-- Using the DEFAULT keyword for the first parameter.
EXECUTE dbo.ProcTestDefaults @p1 = DEFAULT, @p2 = 'D';
-- Specifying the parameters in an order different from the order defined in the procedure.
EXECUTE dbo.ProcTestDefaults DEFAULT, @p3 = 'Local', @p2 = 'E';
-- Using the DEFAULT keyword for the first and third parameters.
EXECUTE dbo.ProcTestDefaults DEFAULT, 'H', DEFAULT;
EXECUTE dbo.ProcTestDefaults DEFAULT, 'I', @p3 = DEFAULT;
```

G. Using EXECUTE with AT linked_server_name

The following example passes a command string to a remote server. It creates a linked server `SeattleSales` that points to another instance of SQL Server and executes a DDL statement (`CREATE TABLE`) against that linked server.

Applies to: SQL Server 2008 through SQL Server 2017

```
EXEC sp_addlinkedserver 'SeattleSales', 'SQL Server'
GO
EXECUTE ( 'CREATE TABLE AdventureWorks2012.dbo.SalesTbl
(SalesID int, SalesName varchar(10)) ; ' ) AT SeattleSales;
GO
```

H. Using EXECUTE WITH RECOMPILE

The following example executes the `Proc_Test_Defaults` stored procedure and forces a new query plan to be compiled, used, and discarded after the module is executed.

```
EXECUTE dbo.Proc_Test_Defaults @p2 = 'A' WITH RECOMPILE;
GO
```

I. Using EXECUTE with a user-defined function

The following example executes the `ufnGetSalesOrderStatusText` scalar user-defined function in the AdventureWorks2012 database. It uses the variable `@returnstatus` to store the value returned by the function. The function expects one input parameter, `@Status`. This is defined as a **tinyint** data type.

```
DECLARE @returnstatus nvarchar(15);
SET @returnstatus = NULL;
EXEC @returnstatus = dbo.ufnGetSalesOrderStatusText @Status = 2;
PRINT @returnstatus;
GO
```

J. Using EXECUTE to query an Oracle database on a linked server

The following example executes several `SELECT` statements at the remote Oracle server. The example begins by adding the Oracle server as a linked server and creating linked server login.

Applies to: SQL Server 2008 through SQL Server 2017

```
-- Setup the linked server.
EXEC sp_addlinkedserver
        @server='ORACLE',
        @srvproduct='Oracle',
        @provider='OraOLEDB.Oracle',
        @datasrc='ORACLE10';

EXEC sp_addlinkedsrvlogin
    @rmtsrvname='ORACLE',
    @useself='false',
    @locallogin=null,
    @rmtuser='scott',
    @rmtpassword='tiger';

EXEC sp_serveroption 'ORACLE', 'rpc out', true;
GO

-- Execute several statements on the linked Oracle server.
EXEC ( 'SELECT * FROM scott.emp') AT ORACLE;
GO
EXEC ( 'SELECT * FROM scott.emp WHERE MGR = ?', 7902) AT ORACLE;
GO
DECLARE @v INT;
SET @v = 7902;
EXEC ( 'SELECT * FROM scott.emp WHERE MGR = ?', @v) AT ORACLE;
GO
```

K. Using EXECUTE AS USER to switch context to another user

The following example executes a Transact-SQL string that creates a table and specifies the `AS USER` clause to switch the execution context of the statement from the caller to `User1`. The Database Engine will check the permissions of `User1` when the statement is run. `User1` must exist as a user in the database and must have permission to create tables in the `Sales` schema, or the statement fails.

```
EXECUTE ('CREATE TABLE Sales.SalesTable (SalesID int, SalesName varchar(10));')
AS USER = 'User1';
GO
```

L. Using a parameter with EXECUTE and AT linked_server_name

The following example passes a command string to a remote server by using a question mark (`?`) placeholder for a parameter. The example creates a linked server `SeattleSales` that points to another instance of SQL Server and executes a `SELECT` statement against that linked server. The `SELECT` statement uses the question mark as a place holder for the `ProductID` parameter (`952`), which is provided after the statement.

Applies to: SQL Server 2008 through SQL Server 2017

```
-- Setup the linked server.
EXEC sp_addlinkedserver 'SeattleSales', 'SQL Server'
GO
-- Execute the SELECT statement.
EXECUTE ('SELECT ProductID, Name
    FROM AdventureWorks2012.Production.Product
    WHERE ProductID = ? ', 952) AT SeattleSales;
GO
```

M. Using EXECUTE to redefine a single result set

Some of the previous examples executed `EXEC dbo.uspGetEmployeeManagers 6;` which returned 7 columns. The following example demonstrates using the `WITH RESULT SET` syntax to change the names and data types of the returning result set.

Applies to: SQL Server 2012 (11.x) through SQL Server 2017, Azure SQL Database

```
EXEC uspGetEmployeeManagers 16
WITH RESULT SETS
(
   ([Reporting Level] int NOT NULL,
    [ID of Employee] int NOT NULL,
    [Employee First Name] nvarchar(50) NOT NULL,
    [Employee Last Name] nvarchar(50) NOT NULL,
    [Employee ID of Manager] nvarchar(max) NOT NULL,
    [Manager First Name] nvarchar(50) NOT NULL,
    [Manager Last Name] nvarchar(50) NOT NULL )
);
```

N. Using EXECUTE to redefine a two result sets

When executing a statement that returns more than one result set, define each expected result set. The following example in **AdventureWorks2012** creates a procedure that returns two result sets. Then the procedure is executed using the **WITH RESULT SETS** clause, and specifying two result set definitions.

Applies to: SQL Server 2012 (11.x) through SQL Server 2017, Azure SQL Database

```
--Create the procedure
CREATE PROC Production.ProductList @ProdName nvarchar(50)
AS
-- First result set
SELECT ProductID, Name, ListPrice
    FROM Production.Product
    WHERE Name LIKE @ProdName;
-- Second result set
SELECT Name, COUNT(S.ProductID) AS NumberOfOrders
    FROM Production.Product AS P
    JOIN Sales.SalesOrderDetail AS S
        ON P.ProductID  = S.ProductID
    WHERE Name LIKE @ProdName
    GROUP BY Name;
GO

-- Execute the procedure
EXEC Production.ProductList '%tire%'
WITH RESULT SETS
(
    (ProductID int,   -- first result set definition starts here
    Name Name,
    ListPrice money)
    ,                 -- comma separates result set definitions
    (Name Name,       -- second result set definition starts here
    NumberOfOrders int)
);
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

Example O: Basic Procedure Execution

Executing a stored procedure:

```
EXEC proc1;
```

Calling a stored procedure with name determined at runtime:

```
EXEC ('EXEC ' + @var);
```

Calling a stored procedure from within a stored procedure:

```
CREATE sp_first AS EXEC sp_second; EXEC sp_third;
```

Example P: Executing Strings

Executing a SQL string:

```
EXEC ('SELECT * FROM sys.types');
```

Executing a nested string:

```
EXEC ('EXEC (''SELECT * FROM sys.types'')');
```

Executing a string variable:

```
DECLARE @stringVar nvarchar(100);
SET @stringVar = N'SELECT name FROM' + ' sys.sql_logins';
EXEC (@stringVar);
```

Example Q: Procedures with Parameters

The following example creates a procedure with parameters and demonstrates 3 ways to execute the procedure:

```
-- Uses AdventureWorks

CREATE PROC ProcWithParameters
    @name nvarchar(50),
@color nvarchar (15)
AS
SELECT ProductKey, EnglishProductName, Color FROM [dbo].[DimProduct]
WHERE EnglishProductName LIKE @name
AND Color = @color;
GO

-- Executing using positional parameters
EXEC ProcWithParameters N'%arm%', N'Black';
-- Executing using named parameters in order
EXEC ProcWithParameters @name = N'%arm%', @color = N'Black';
-- Executing using named parameters out of order
EXEC ProcWithParameters @color = N'Black', @name = N'%arm%';
GO
```

See Also

@@NESTLEVEL (Transact-SQL)
DECLARE @local_variable (Transact-SQL)
EXECUTE AS Clause (Transact-SQL)
osql Utility
Principals (Database Engine)
REVERT (Transact-SQL)
sp_addlinkedserver (Transact-SQL)
sqlcmd Utility
SUSER_NAME (Transact-SQL)
sys.database_principals (Transact-SQL)
sys.server_principals (Transact-SQL)
USER_NAME (Transact-SQL)
OPENDATASOURCE (Transact-SQL)
Scalar User-Defined Functions for In-Memory OLTP

PRINT (Transact-SQL)

8/27/2018 • 2 minutes to read • Edit Online

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

Returns a user-defined message to the client.

Transact-SQL Syntax Conventions

Syntax

```
PRINT msg_str | @local_variable | string_expr
```

Arguments

msg_str
Is a character string or Unicode string constant. For more information, see Constants (Transact-SQL).

@ *local_variable*
Is a variable of any valid character data type. **@***local_variable* must be **char**, **nchar**, **varchar**, or **nvarchar**, or it must be able to be implicitly converted to those data types.

string_expr
Is an expression that returns a string. Can include concatenated literal values, functions, and variables. For more information, see Expressions (Transact-SQL).

Remarks

A message string can be up to 8,000 characters long if it is a non-Unicode string, and 4,000 characters long if it is a Unicode string. Longer strings are truncated. The **varchar(max)** and **nvarchar(max)** data types are truncated to data types that are no larger than **varchar(8000)** and **nvarchar(4000)**.

RAISERROR can also be used to return messages. RAISERROR has these advantages over PRINT:

- RAISERROR supports substituting arguments into an error message string using a mechanism modeled on the printf function of the C language standard library.
- RAISERROR can specify a unique error number, a severity, and a state code in addition to the text message.
- RAISERROR can be used to return user-defined messages created using the sp_addmessage system stored procedure.

Examples

A. Conditionally executing print (IF EXISTS)

The following example uses the `PRINT` statement to conditionally return a message.

```
IF @@OPTIONS & 512 <> 0
    PRINT N'This user has SET NOCOUNT turned ON.';
ELSE
    PRINT N'This user has SET NOCOUNT turned OFF.';
GO
```

B. Building and displaying a string

The following example converts the results of the `GETDATE` function to a `nvarchar` data type and concatenates it with literal text to be returned by `PRINT`.

```
-- Build the message text by concatenating
-- strings and expressions.
PRINT N'This message was printed on '
    + RTRIM(CAST(GETDATE() AS nvarchar(30)))
    + N'.';
GO
-- This example shows building the message text
-- in a variable and then passing it to PRINT.
-- This was required in SQL Server 7.0 or earlier.
DECLARE @PrintMessage nvarchar(50);
SET @PrintMessage = N'This message was printed on '
    + RTRIM(CAST(GETDATE() AS nvarchar(30)))
    + N'.';
PRINT @PrintMessage;
GO
```

Examples: Azure SQL Data Warehouse and Parallel Data Warehouse

C. Conditionally executing print

The following example uses the `PRINT` statement to conditionally return a message.

```
IF DB_ID() = 1
    PRINT N'The current database is master.';
ELSE
    PRINT N'The current database is not master.';
GO
```

See Also

Data Types (Transact-SQL)
DECLARE @local_variable (Transact-SQL)
RAISERROR (Transact-SQL)

RAISERROR (Transact-SQL)

8/27/2018 • 11 minutes to read • Edit Online

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

Generates an error message and initiates error processing for the session. RAISERROR can either reference a user-defined message stored in the sys.messages catalog view or build a message dynamically. The message is returned as a server error message to the calling application or to an associated CATCH block of a TRY...CATCH construct. New applications should use THROW instead.

Transact-SQL Syntax Conventions

Syntax

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

RAISERROR ( { msg_id | msg_str | @local_variable }
    { ,severity ,state }
    [ ,argument [ ,...n ] ] )
    [ WITH option [ ,...n ] ]
```

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

RAISERROR ( { msg_str | @local_variable }
    { ,severity ,state }
    [ ,argument [ ,...n ] ] )
    [ WITH option [ ,...n ] ]
```

Arguments

msg_id
Is a user-defined error message number stored in the sys.messages catalog view using sp_addmessage. Error numbers for user-defined error messages should be greater than 50000. When *msg_id* is not specified, RAISERROR raises an error message with an error number of 50000.

msg_str
Is a user-defined message with formatting similar to the **printf** function in the C standard library. The error message can have a maximum of 2,047 characters. If the message contains 2,048 or more characters, only the first 2,044 are displayed and an ellipsis is added to indicate that the message has been truncated. Note that substitution parameters consume more characters than the output shows because of internal storage behavior. For example, the substitution parameter of *%d* with an assigned value of 2 actually produces one character in the message string but also internally takes up three additional characters of storage. This storage requirement decreases the number of available characters for message output.

When *msg_str* is specified, RAISERROR raises an error message with an error number of 50000.

msg_str is a string of characters with optional embedded conversion specifications. Each conversion specification defines how a value in the argument list is formatted and placed into a field at the location of the conversion specification in *msg_str*. Conversion specifications have this format:

% [[*flag*] [*width*] [. *precision*] [{h | l}]] *type*

The parameters that can be used in *msg_str* are:

flag

Is a code that determines the spacing and justification of the substituted value.

CODE	PREFIX OR JUSTIFICATION	DESCRIPTION
- (minus)	Left-justified	Left-justify the argument value within the given field width.
+ (plus)	Sign prefix	Preface the argument value with a plus (+) or minus (-) if the value is of a signed type.
0 (zero)	Zero padding	Preface the output with zeros until the minimum width is reached. When 0 and the minus sign (-) appear, 0 is ignored.
# (number)	0x prefix for hexadecimal type of x or X	When used with the o, x, or X format, the number sign (#) flag prefaces any nonzero value with 0, 0x, or 0X, respectively. When d, i, or u are prefaced by the number sign (#) flag, the flag is ignored.
' ' (blank)	Space padding	Preface the output value with blank spaces if the value is signed and positive. This is ignored when included with the plus sign (+) flag.

width

Is an integer that defines the minimum width for the field into which the argument value is placed. If the length of the argument value is equal to or longer than *width*, the value is printed with no padding. If the value is shorter than *width*, the value is padded to the length specified in *width*.

An asterisk (*) means that the width is specified by the associated argument in the argument list, which must be an integer value.

precision

Is the maximum number of characters taken from the argument value for string values. For example, if a string has five characters and precision is 3, only the first three characters of the string value are used.

For integer values, *precision* is the minimum number of digits printed.

An asterisk (*) means that the precision is specified by the associated argument in the argument list, which must be an integer value.

{h | l} *type*

Is used with character types d, i, o, s, x, X, or u, and creates **shortint** (h) or **longint** (l) values.

TYPE SPECIFICATION	REPRESENTS
d or i	Signed integer
o	Unsigned octal

TYPE SPECIFICATION	REPRESENTS
s	String
u	Unsigned integer
x or X	Unsigned hexadecimal

> **NOTE**
> These type specifications are based on the ones originally defined for the **printf** function in the C standard library. The type specifications used in RAISERROR message strings map to Transact-SQL data types, while the specifications used in **printf** map to C language data types. Type specifications used in **printf** are not supported by RAISERROR when Transact-SQL does not have a data type similar to the associated C data type. For example, the *%p* specification for pointers is not supported in RAISERROR because Transact-SQL does not have a pointer data type.

> **NOTE**
> To convert a value to the Transact-SQL **bigint** data type, specify **%I64d**.

@local_variable
Is a variable of any valid character data type that contains a string formatted in the same manner as *msg_str*. *@local_variable* must be **char** or **varchar**, or be able to be implicitly converted to these data types.

severity
Is the user-defined severity level associated with this message. When using *msg_id* to raise a user-defined message created using sp_addmessage, the severity specified on RAISERROR overrides the severity specified in sp_addmessage.

Severity levels from 0 through 18 can be specified by any user. Severity levels from 19 through 25 can only be specified by members of the sysadmin fixed server role or users with ALTER TRACE permissions. For severity levels from 19 through 25, the WITH LOG option is required. Severity levels less than 0 are interpreted as 0. Severity levels greater than 25 are interpreted as 25.

Caution
Severity levels from 20 through 25 are considered fatal. If a fatal severity level is encountered, the client connection is terminated after receiving the message, and the error is logged in the error and application logs.

You can specify -1 to return the severity value associated with the error as shown in the following example.

```
RAISERROR (15600,-1,-1, 'mysp_CreateCustomer');
```

Here is the result set.

```
Msg 15600, Level 15, State 1, Line 1
An invalid parameter or option was specified for procedure 'mysp_CreateCustomer'.
```

state
Is an integer from 0 through 255. Negative values default to 1. Values larger than 255 should not be used.

If the same user-defined error is raised at multiple locations, using a unique state number for each location can help find which section of code is raising the errors.

argument
Are the parameters used in the substitution for variables defined in *msg_str* or the message corresponding to *msg_id*. There can be 0 or more substitution parameters, but the total number of substitution parameters cannot exceed 20. Each substitution parameter can be a local variable or any of these data types: **tinyint**, **smallint**, **int**, **char**, **varchar**, **nchar**, **nvarchar**, **binary**, or **varbinary**. No other data types are supported.

option
Is a custom option for the error and can be one of the values in the following table.

VALUE	DESCRIPTION
LOG	Logs the error in the error log and the application log for the instance of the Microsoft SQL Server Database Engine. Errors logged in the error log are currently limited to a maximum of 440 bytes. Only a member of the sysadmin fixed server role or a user with ALTER TRACE permissions can specify WITH LOG. **Applies to:** SQL Server, SQL Database
NOWAIT	Sends messages immediately to the client. **Applies to:** SQL Server, SQL Database
SETERROR	Sets the @@ERROR and ERROR_NUMBER values to *msg_id* or 50000, regardless of the severity level. **Applies to:** SQL Server, SQL Database

Remarks

The errors generated by RAISERROR operate the same as errors generated by the Database Engine code. The values specified by RAISERROR are reported by the ERROR_LINE, ERROR_MESSAGE, ERROR_NUMBER, ERROR_PROCEDURE, ERROR_SEVERITY, ERROR_STATE, and @@ERROR system functions. When RAISERROR is run with a severity of 11 or higher in a TRY block, it transfers control to the associated CATCH block. The error is returned to the caller if RAISERROR is run:

- Outside the scope of any TRY block.
- With a severity of 10 or lower in a TRY block.
- With a severity of 20 or higher that terminates the database connection.

  CATCH blocks can use RAISERROR to rethrow the error that invoked the CATCH block by using system functions such as ERROR_NUMBER and ERROR_MESSAGE to retrieve the original error information. @@ERROR is set to 0 by default for messages with a severity from 1 through 10.

  When *msg_id* specifies a user-defined message available from the sys.messages catalog view, RAISERROR processes the message from the text column using the same rules as are applied to the text of a user-defined message specified using *msg_str*. The user-defined message text can contain conversion specifications, and RAISERROR will map argument values into the conversion specifications. Use sp_addmessage to add user-defined error messages and sp_dropmessage to delete user-defined error messages.

  RAISERROR can be used as an alternative to PRINT to return messages to calling applications. RAISERROR supports character substitution similar to the functionality of the **printf** function in the C standard library, while the Transact-SQL PRINT statement does not. The PRINT statement is not affected

by TRY blocks, while a RAISERROR run with a severity of 11 to 19 in a TRY block transfers control to the associated CATCH block. Specify a severity of 10 or lower to use RAISERROR to return a message from a TRY block without invoking the CATCH block.

Typically, successive arguments replace successive conversion specifications; the first argument replaces the first conversion specification, the second argument replaces the second conversion specification, and so on. For example, in the following `RAISERROR` statement, the first argument of `N'number'` replaces the first conversion specification of `%s`; and the second argument of `5` replaces the second conversion specification of `%d.`

```
RAISERROR (N'This is message %s %d.', -- Message text.
           10, -- Severity,
           1, -- State,
           N'number', -- First argument.
           5); -- Second argument.
-- The message text returned is: This is message number 5.
GO
```

If an asterisk (*) is specified for either the width or precision of a conversion specification, the value to be used for the width or precision is specified as an integer argument value. In this case, one conversion specification can use up to three arguments, one each for the width, precision, and substitution value.

For example, both of the following `RAISERROR` statements return the same string. One specifies the width and precision values in the argument list; the other specifies them in the conversion specification.

```
RAISERROR (N'<\<%*.*s>>', -- Message text.
           10, -- Severity,
           1, -- State,
           7, -- First argument used for width.
           3, -- Second argument used for precision.
           N'abcde'); -- Third argument supplies the string.
-- The message text returned is: <<    abc>>.
GO
RAISERROR (N'<\<%7.3s>>', -- Message text.
           10, -- Severity,
           1, -- State,
           N'abcde'); -- First argument supplies the string.
-- The message text returned is: <<    abc>>.
GO
```

Examples

A. Returning error information from a CATCH block

The following code example shows how to use `RAISERROR` inside a `TRY` block to cause execution to jump to the associated `CATCH` block. It also shows how to use `RAISERROR` to return information about the error that invoked the `CATCH` block.

> **NOTE**
>
> RAISERROR only generates errors with state from 1 through 127. Because the Database Engine may raise errors with state 0, we recommend that you check the error state returned by ERROR_STATE before passing it as a value to the state parameter of RAISERROR.

```
BEGIN TRY
    -- RAISERROR with severity 11-19 will cause execution to
    -- jump to the CATCH block.
    RAISERROR ('Error raised in TRY block.', -- Message text.
               16, -- Severity.
               1 -- State.
               );
END TRY
BEGIN CATCH
    DECLARE @ErrorMessage NVARCHAR(4000);
    DECLARE @ErrorSeverity INT;
    DECLARE @ErrorState INT;

    SELECT
        @ErrorMessage = ERROR_MESSAGE(),
        @ErrorSeverity = ERROR_SEVERITY(),
        @ErrorState = ERROR_STATE();

    -- Use RAISERROR inside the CATCH block to return error
    -- information about the original error that caused
    -- execution to jump to the CATCH block.
    RAISERROR (@ErrorMessage, -- Message text.
               @ErrorSeverity, -- Severity.
               @ErrorState -- State.
               );
END CATCH;
```

B. Creating an ad hoc message in sys.messages

The following example shows how to raise a message stored in the sys.messages catalog view. The message was added to the sys.messages catalog view by using the `sp_addmessage` system stored procedure as message number `50005`.

```
sp_addmessage @msgnum = 50005,
              @severity = 10,
              @msgtext = N'<\<%7.3s>>';
GO
RAISERROR (50005, -- Message id.
           10, -- Severity,
           1, -- State,
           N'abcde'); -- First argument supplies the string.
-- The message text returned is: <<    abc>>.
GO
sp_dropmessage @msgnum = 50005;
GO
```

C. Using a local variable to supply the message text

The following code example shows how to use a local variable to supply the message text for a `RAISERROR` statement.

```
DECLARE @StringVariable NVARCHAR(50);
SET @StringVariable = N'<\<%7.3s>>';

RAISERROR (@StringVariable, -- Message text.
           10, -- Severity,
           1, -- State,
           N'abcde'); -- First argument supplies the string.
-- The message text returned is: <<    abc>>.
GO
```

See Also

Built-in Functions (Transact-SQL)
DECLARE @local_variable (Transact-SQL)
PRINT (Transact-SQL)
sp_addmessage (Transact-SQL)
sp_dropmessage (Transact-SQL)
sys.messages (Transact-SQL)
xp_logevent (Transact-SQL)
@@ERROR (Transact-SQL)
ERROR_LINE (Transact-SQL)
ERROR_MESSAGE (Transact-SQL)
ERROR_NUMBER (Transact-SQL)
ERROR_PROCEDURE (Transact-SQL)
ERROR_SEVERITY (Transact-SQL)
ERROR_STATE (Transact-SQL)
TRY...CATCH (Transact-SQL)

CHECKPOINT (Transact-SQL)

7/4/2018 • 2 minutes to read • Edit Online

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

Generates a manual checkpoint in the SQL Server database to which you are currently connected.

> **NOTE**
> For information about different types of database checkpoints and checkpoint operation in general, see Database Checkpoints (SQL Server).

Transact-SQL Syntax Conventions

Syntax

```
CHECKPOINT [ checkpoint_duration ]
```

Arguments

checkpoint_duration
Specifies the requested amount of time, in seconds, for the manual checkpoint to complete. When *checkpoint_duration* is specified, the SQL Server Database Engine attempts to perform the checkpoint within the requested duration. The *checkpoint_duration* must be an expression of type **int** and must be greater than zero. When this parameter is omitted, the Database Engine adjusts the checkpoint duration to minimize the performance impact on database applications. *checkpoint_duration* is an advanced option.

Factors Affecting the Duration of Checkpoint Operations

In general, the amount time required for a checkpoint operation increases with the number of dirty pages that the operation must write. By default, to minimize the performance impact on other applications, SQL Server adjusts the frequency of writes that a checkpoint operation performs. Decreasing the write frequency increases the time the checkpoint operation requires to complete. SQL Server uses this strategy for a manual checkpoint unless a *checkpoint_duration* value is specified in the CHECKPOINT command.

The performance impact of using *checkpoint_duration* depends on the number of dirty pages, the activity on the system, and the actual duration specified. For example, if the checkpoint would normally complete in 120 seconds, specifying a *checkpoint_duration* of 45 seconds causes SQL Server to devote more resources to the checkpoint than would be assigned by default. In contrast, specifying a *checkpoint_duration* of 180 seconds would cause SQL Server to assign fewer resources than would be assigned by default. In general, a short *checkpoint_duration* will increase the resources devoted to the checkpoint, while a long *checkpoint_duration* will reduce the resources devoted to the checkpoint. SQL Server always completes a checkpoint if possible, and the CHECKPOINT statement returns immediately when a checkpoint completes. Therefore, in some cases, a checkpoint may complete sooner than the specified duration or may run longer than the specified duration.

Security

Permissions

CHECKPOINT permissions default to members of the **sysadmin** fixed server role and the **db_owner** and **db_backupoperator** fixed database roles, and are not transferable.

See Also

ALTER DATABASE (Transact-SQL)
Database Checkpoints (SQL Server)
Configure the recovery interval Server Configuration Option
SHUTDOWN (Transact-SQL)

KILL (Transact-SQL)

8/27/2018 • 5 minutes to read • Edit Online

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

Terminates a user process that is based on the session ID or unit of work (UOW). If the specified session ID or UOW has much work to undo, the KILL statement may take some time to complete, particularly when it involves rolling back a long transaction.

KILL can be used to terminate a normal connection, which internally terminates the transactions that are associated with the specified session ID. The statement can also be used to terminate orphaned and in-doubt distributed transactions when Microsoft Distributed Transaction Coordinator (MS DTC) is in use.

Transact-SQL Syntax Conventions

Syntax

```
-- Syntax for SQL Server

KILL { session ID | UOW } [ WITH STATUSONLY ]
```

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

KILL 'session_id'
[;]
```

Arguments

session ID
Is the session ID of the process to terminate. *session ID* is a unique integer (**int**) that is assigned to each user connection when the connection is made. The session ID value is tied to the connection for the duration of the connection. When the connection ends, the integer value is released and can be reassigned to a new connection. The following query can help you identify the `session_id` that you want to kill:

```
SELECT conn.session_id, host_name, program_name,
    nt_domain, login_name, connect_time, last_request_end_time
FROM sys.dm_exec_sessions AS sess
JOIN sys.dm_exec_connections AS conn
   ON sess.session_id = conn.session_id;
```

UOW
Applies to: (SQL Server 2008 through SQL Server 2017

Identifies the Unit of Work ID (UOW) of distributed transactions. *UOW* is a GUID that may be obtained from the request_owner_guid column of the sys.dm_tran_locks dynamic management view. *UOW* also can be obtained from the error log or through the MS DTC monitor. For more information about monitoring distributed transactions, see the MS DTC documentation.

Use KILL *UOW* to terminate orphaned distributed transactions. These transactions are not associated with any real session ID, but instead are associated artificially with session ID = '-2'. This session ID makes it easier to identify

orphaned transactions by querying the session ID column in sys.dm_tran_locks, sys.dm_exec_sessions, or sys.dm_exec_requests dynamic management views.

WITH STATUSONLY
Generates a progress report about a specified *session ID* or *UOW* that is being rolled back due to an earlier KILL statement. KILL WITH STATUSONLY does not terminate or roll back the *session ID* or *UOW*; the command only displays the current progress of the rollback.

Remarks

KILL is commonly used to terminate a process that is blocking other important processes with locks, or a process that is executing a query that is using necessary system resources. System processes and processes running an extended stored procedure cannot be terminated.

Use KILL carefully, especially when critical processes are running. You cannot kill your own process. Other processes you should not kill include the following:

- AWAITING COMMAND
- CHECKPOINT SLEEP
- LAZY WRITER
- LOCK MONITOR
- SIGNAL HANDLER

Use @@SPID to display the session ID value for the current session.

To obtain a report of active session ID values, you can query the session_id column of the sys.dm_tran_locks, sys.dm_exec_sessions, and sys.dm_exec_requests dynamic management views. You can also view the SPID column that is returned by the sp_who system stored procedure. If a rollback is in progress for a specific SPID, the cmd column in the sp_who result set for that SPID indicates KILLED/ROLLBACK.

When a particular connection has a lock on a database resource and blocks the progress of another connection, the session ID of the blocking connection shows up in the `blocking_session_id` column of `sys.dm_exec_requests` or the `blk` column returned by `sp_who`.

The KILL command can be used to resolve in-doubt distributed transactions. These transactions are unresolved distributed transactions that occur because of unplanned restarts of the database server or MS DTC coordinator. For more information about in-doubt transactions, see the "Two-Phase Commit" section in Use Marked Transactions to Recover Related Databases Consistently (Full Recovery Model).

Using WITH STATUSONLY

KILL WITH STATUSONLY generates a report only if the session ID or UOW is currently being rolled back because of a previous KILL *session ID|UOW* statement. The progress report states the amount of rollback completed (in percent) and the estimated length of time left (in seconds), in the following form:

```
Spid|UOW <xxx>: Transaction rollback in progress. Estimated rollback completion: <yy>% Estimated time left: <zz>
seconds
```

If the rollback of the session ID or UOW has finished when the KILL *session ID|UOW* WITH STATUSONLY statement is executed, or if no session ID or UOW is being rolled back, KILL *session ID|UOW* WITH STATUSONLY returns the following error:

```
"Msg 6120, Level 16, State 1, Line 1"
"Status report cannot be obtained. Rollback operation for Process ID <session ID> is not in progress."
```

The same status report can be obtained by repeating the same KILL *session ID|UOW* statement without using the

WITH STATUSONLY option; however, we do not recommend doing this. Repeating a KILL *session ID* statement might terminate a new process if the rollback had finished and the session ID was reassigned to a new task before the new KILL statement is run. Specifying WITH STATUSONLY prevents this from happening.

Permissions

SQL Server: Requires the ALTER ANY CONNECTION permission. ALTER ANY CONNECTION is included with membership in the sysadmin or processadmin fixed server roles.

SQL Database: Requires the KILL DATABASE CONNECTION permission. The server-level principal login has the KILL DATABASE CONNECTION.

Examples

A. Using KILL to terminate a session

The following example shows how to terminate session ID `53`.

```
KILL 53;
GO
```

B. Using KILL session ID WITH STATUSONLY to obtain a progress report

The following example generates a status of the rollback process for the specific session ID.

```
KILL 54;
KILL 54 WITH STATUSONLY;
GO

--This is the progress report.
spid 54: Transaction rollback in progress. Estimated rollback completion: 80% Estimated time left: 10 seconds.
```

C. Using KILL to terminate an orphaned distributed transaction

The following example shows how to terminate an orphaned distributed transaction (session ID = -2) with a *UOW* of `D5499C66-E398-45CA-BF7E-DC9C194B48CF`.

```
KILL 'D5499C66-E398-45CA-BF7E-DC9C194B48CF';
```

See Also

KILL STATS JOB (Transact-SQL)
KILL QUERY NOTIFICATION SUBSCRIPTION (Transact-SQL)
Built-in Functions (Transact-SQL)
SHUTDOWN (Transact-SQL)
@@SPID (Transact-SQL)
sys.dm_exec_requests (Transact-SQL)
sys.dm_exec_sessions (Transact-SQL)
sys.dm_tran_locks (Transact-SQL)
sp_lock (Transact-SQL)
sp_who (Transact-SQL)

KILL QUERY NOTIFICATION SUBSCRIPTION (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Removes query notification subscriptions from the instance. This statement can remove a specific subscription or all subscriptions.

Transact-SQL Syntax Conventions

Syntax

```
KILL QUERY NOTIFICATION SUBSCRIPTION
   { ALL | subscription_id }
```

Arguments

ALL
Removes all subscriptions in the instance.

subscription_id
Removes the subscription with the subscription id *subscription_id*.

Remarks

The KILL QUERY NOTIFICATION SUBSCRIPTION statement removes query notification subscriptions without producing a notification message.

subscription_id is the id for the subscription as shown in the dynamic management view sys.dm_qn_subscriptions (Transact-SQL).

If the specified subscription id does not exist, the statement produces an error.

Permissions

Permission to execute this statement is restricted to members of the **sysadmin** fixed server role.

Examples

A. Removing all query notification subscriptions in the instance

The following example removes all query notification subscriptions in the instance.

```
KILL QUERY NOTIFICATION SUBSCRIPTION ALL ;
```

B. Removing a single query notification subscription

The following example removes the query notification subscription with the id `73`.

```
KILL QUERY NOTIFICATION SUBSCRIPTION 73 ;
```

See Also

sys.dm_qn_subscriptions (Transact-SQL)

KILL STATS JOB (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Terminates an asynchronous statistics update job in SQL Server.

Transact-SQL Syntax Conventions

Syntax

```
KILL STATS JOB job_id
```

Arguments

job_id
Is the job_id field returned by the sys.dm_exec_background_job_queue dynamic management view for the job.

Remarks

The job_id is unrelated to session_id or unit of work used in other forms of the KILL statement.

Permissions

User must have VIEW SERVER STATE permission to access information from the sys.dm_exec_background_job_queue dynamic management view.

KILL STATS JOB permissions default to the members of the sysadmin and processadmin fixed database roles and are not transferable.

Examples

The following example shows how to terminate the statistics update associated with a job where the *job_id* = `53`.

```
KILL STATS JOB 53;
GO
```

See Also

KILL (Transact-SQL)
KILL QUERY NOTIFICATION SUBSCRIPTION (Transact-SQL)
sys.dm_exec_background_job_queue (Transact-SQL)
Statistics

RECONFIGURE (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Updates the currently configured value (the **config_value** column in the **sp_configure** result set) of a configuration option changed with the **sp_configure** system stored procedure. Because some configuration options require a server stop and restart to update the currently running value, RECONFIGURE does not always update the currently running value (the **run_value** column in the **sp_configure** result set) for a changed configuration value.

Transact-SQL Syntax Conventions

Syntax

```
RECONFIGURE [ WITH OVERRIDE ]
```

Arguments

RECONFIGURE

Specifies that if the configuration setting does not require a server stop and restart, the currently running value should be updated. RECONFIGURE also checks the new configuration values for either values that are not valid (for example, a sort order value that does not exist in **syscharsets**) or nonrecommended values. With those configuration options not requiring a server stop and restart, the currently running value and the currently configured values for the configuration option should be the same value after RECONFIGURE is specified.

WITH OVERRIDE

Disables the configuration value checking (for values that are not valid or for nonrecommended values) for the **recovery interval** advanced configuration option.

Almost any configuration option can be reconfigured by using the WITH OVERRIDE option, however some fatal errors are still prevented. For example, the **min server memory** configuration option cannot be configured with a value greater than the value specified in the **max server memory** configuration option.

Remarks

sp_configure does not accept new configuration option values out of the documented valid ranges for each configuration option.

RECONFIGURE is not allowed in an explicit or implicit transaction. When you reconfigure several options at the same time, if any of the reconfigure operations fail, none of the reconfigure operations will take effect.

When reconfiguring the resource governor, see the RECONFIGURE option of ALTER RESOURCE GOVERNOR (Transact-SQL).

Permissions

RECONFIGURE permissions default to grantees of the ALTER SETTINGS permission. The **sysadmin** and **serveradmin** fixed server roles implicitly hold this permission.

Examples

The following example sets the upper limit for the `recovery interval` configuration option to `75` minutes and uses `RECONFIGURE WITH OVERRIDE` to install it. Recovery intervals greater than 60 minutes are not recommended and disallowed by default. However, because the `WITH OVERRIDE` option is specified, SQL Server does not check whether the value specified (`90`) is a valid value for the `recovery interval` configuration option.

```
EXEC sp_configure 'recovery interval', 75'
RECONFIGURE WITH OVERRIDE;
GO
```

See Also

Server Configuration Options (SQL Server)
sp_configure (Transact-SQL)

SHUTDOWN (Transact-SQL)

6/20/2018 • 2 minutes to read • Edit Online

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

Immediately stops SQL Server.

Transact-SQL Syntax Conventions

Syntax

```
SHUTDOWN [ WITH NOWAIT ]
```

Arguments

WITH NOWAIT
Optional. Shuts down SQL Server without performing checkpoints in every database. SQL Server exits after attempting to terminate all user processes. When the server restarts, a rollback operation occurs for uncompleted transactions.

Remarks

Unless the WITHNOWAIT option is used, SHUTDOWN shuts down SQL Server by:

1. Disabling logins (except for members of the **sysadmin** and **serveradmin** fixed server roles).

   > **NOTE**
   > To display a list of all current users, run **sp_who**.

2. Waiting for currently running Transact-SQL statements or stored procedures to finish. To display a list of all active processes and locks, run **sp_who** and **sp_lock**, respectively.

3. Inserting a checkpoint in every database.

   Using the SHUTDOWN statement minimizes the amount of automatic recovery work needed when members of the **sysadmin** fixed server role restart SQL Server.

   Other tools and methods can also be used to stop SQL Server. Each of these issues a checkpoint in all databases. You can flush committed data from the data cache and stop the server:

- By using SQL Server Configuration Manager.
- By running **net stop mssqlserver** from a command prompt for a default instance, or by running **net stop mssql$***instancename* from a command prompt for a named instance.
- By using Services in Control Panel.

  If **sqlservr.exe** was started from the command prompt, pressing CTRL+C shuts down SQL Server. However, pressing CTRL+C does not insert a checkpoint.

NOTE

Using any of these methods to stop SQL Server sends the `SERVICE_CONTROL_STOP` message to SQL Server.

Permissions

SHUTDOWN permissions are assigned to members of the **sysadmin** and **serveradmin** fixed server roles, and they are not transferable.

See Also

CHECKPOINT (Transact-SQL)
sp_lock (Transact-SQL)
sp_who (Transact-SQL)
sqlservr Application
Start, Stop, Pause, Resume, Restart the Database Engine, SQL Server Agent, or SQL Server Browser Service

Reserved Keywords (Transact-SQL)

8/27/2018 • 5 minutes to read • Edit Online

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

Microsoft SQL Server uses reserved keywords for defining, manipulating, and accessing databases. Reserved keywords are part of the grammar of the Transact-SQL language that is used by SQL Server to parse and understand Transact-SQL statements and batches. Although it is syntactically possible to use SQL Server reserved keywords as identifiers and object names in Transact-SQL scripts, you can do this only by using delimited identifiers.

The following table lists SQL Server and SQL Data Warehouse reserved keywords.

ADD	EXTERNAL	PROCEDURE
ALL	FETCH	PUBLIC
ALTER	FILE	RAISERROR
AND	FILLFACTOR	READ
ANY	FOR	READTEXT
AS	FOREIGN	RECONFIGURE
ASC	FREETEXT	REFERENCES
AUTHORIZATION	FREETEXTTABLE	REPLICATION
BACKUP	FROM	RESTORE
BEGIN	FULL	RESTRICT
BETWEEN	FUNCTION	RETURN
BREAK	GOTO	REVERT
BROWSE	GRANT	REVOKE
BULK	GROUP	RIGHT
BY	HAVING	ROLLBACK
CASCADE	HOLDLOCK	ROWCOUNT
CASE	IDENTITY	ROWGUIDCOL
CHECK	IDENTITY_INSERT	RULE

CHECKPOINT	IDENTITYCOL	SAVE
CLOSE	IF	SCHEMA
CLUSTERED	IN	SECURITYAUDIT
COALESCE	INDEX	SELECT
COLLATE	INNER	SEMANTICKEYPHRASETABLE
COLUMN	INSERT	SEMANTICSIMILARITYDETAILSTABLE
COMMIT	INTERSECT	SEMANTICSIMILARITYTABLE
COMPUTE	INTO	SESSION_USER
CONSTRAINT	IS	SET
CONTAINS	JOIN	SETUSER
CONTAINSTABLE	KEY	SHUTDOWN
CONTINUE	KILL	SOME
CONVERT	LEFT	STATISTICS
CREATE	LIKE	SYSTEM_USER
CROSS	LINENO	TABLE
CURRENT	LOAD	TABLESAMPLE
CURRENT_DATE	MERGE	TEXTSIZE
CURRENT_TIME	NATIONAL	THEN
CURRENT_TIMESTAMP	NOCHECK	TO
CURRENT_USER	NONCLUSTERED	TOP
CURSOR	NOT	TRAN
DATABASE	NULL	TRANSACTION
DBCC	NULLIF	TRIGGER
DEALLOCATE	OF	TRUNCATE
DECLARE	OFF	TRY_CONVERT
DEFAULT	OFFSETS	TSEQUAL

DELETE	ON	UNION
DENY	OPEN	UNIQUE
DESC	OPENDATASOURCE	UNPIVOT
DISK	OPENQUERY	UPDATE
DISTINCT	OPENROWSET	UPDATETEXT
DISTRIBUTED	OPENXML	USE
DOUBLE	OPTION	USER
DROP	OR	VALUES
DUMP	ORDER	VARYING
ELSE	OUTER	VIEW
END	OVER	WAITFOR
ERRLVL	PERCENT	WHEN
ESCAPE	PIVOT	WHERE
EXCEPT	PLAN	WHILE
EXEC	PRECISION	WITH
EXECUTE	PRIMARY	WITHIN GROUP
EXISTS	PRINT	WRITETEXT
EXIT	PROC	

The following table lists reserved keywords that are exclusive to **SQL Data Warehouse**.

Label	

Additionally, the ISO standard defines a list of reserved keywords. Avoid using ISO reserved keywords for object names and identifiers. The ODBC reserved keyword list, shown in the following table, is the same as the ISO reserved keyword list.

> **NOTE**
> The ISO standards reserved keywords list sometimes can be more restrictive than SQL Server and at other times less restrictive. For example, the ISO reserved keywords list contains **INT**. SQL Server does not have to distinguish this as a reserved keyword.

Transact-SQL reserved keywords can be used as identifiers or names of databases or database objects, such as tables, columns, views, and so on. Use either quoted identifiers or delimited identifiers. Using reserved keywords as the names of variables and stored procedure parameters is not restricted.

ODBC Reserved Keywords

The following words are reserved for use in ODBC function calls. These words do not constrain the minimum SQL grammar; however, to ensure compatibility with drivers that support the core SQL grammar, applications should avoid using these keywords.

This is the current list of ODBC reserved keywords.

ABSOLUTE	**EXEC**	**OVERLAPS**
ACTION	**EXECUTE**	**PAD**
ADA	**EXISTS**	**PARTIAL**
ADD	**EXTERNAL**	**PASCAL**
ALL	**EXTRACT**	**POSITION**
ALLOCATE	**FALSE**	**PRECISION**
ALTER	**FETCH**	**PREPARE**
AND	**FIRST**	**PRESERVE**
ANY	**FLOAT**	**PRIMARY**
ARE	**FOR**	**PRIOR**
AS	**FOREIGN**	**PRIVILEGES**
ASC	**FORTRAN**	**PROCEDURE**
ASSERTION	**FOUND**	**PUBLIC**
AT	**FROM**	**READ**
AUTHORIZATION	**FULL**	**REAL**
AVG	**GET**	**REFERENCES**
BEGIN	**GLOBAL**	**RELATIVE**
BETWEEN	**GO**	**RESTRICT**
BIT	**GOTO**	**REVOKE**
BIT_LENGTH	**GRANT**	**RIGHT**

BOTH	**GROUP**	**ROLLBACK**
BY	**HAVING**	**ROWS**
CASCADE	**HOUR**	**SCHEMA**
CASCADED	**IDENTITY**	**SCROLL**
CASE	**IMMEDIATE**	**SECOND**
CAST	**IN**	**SECTION**
CATALOG	**INCLUDE**	**SELECT**
CHAR	**INDEX**	**SESSION**
CHAR_LENGTH	**INDICATOR**	**SESSION_USER**
CHARACTER	**INITIALLY**	**SET**
CHARACTER_LENGTH	**INNER**	**SIZE**
CHECK	**INPUT**	**SMALLINT**
CLOSE	**INSENSITIVE**	**SOME**
COALESCE	**INSERT**	**SPACE**
COLLATE	**INT**	**SQL**
COLLATION	**INTEGER**	**SQLCA**
COLUMN	**INTERSECT**	**SQLCODE**
COMMIT	**INTERVAL**	**SQLERROR**
CONNECT	**INTO**	**SQLSTATE**
CONNECTION	**IS**	**SQLWARNING**
CONSTRAINT	**ISOLATION**	**SUBSTRING**
CONSTRAINTS	**JOIN**	**SUM**
CONTINUE	**KEY**	**SYSTEM_USER**
CONVERT	**LANGUAGE**	**TABLE**
CORRESPONDING	**LAST**	**TEMPORARY**
COUNT	**LEADING**	**THEN**

CREATE	**LEFT**	**TIME**
CROSS	**LEVEL**	**TIMESTAMP**
CURRENT	**LIKE**	**TIMEZONE_HOUR**
CURRENT_DATE	**LOCAL**	**TIMEZONE_MINUTE**
CURRENT_TIME	**LOWER**	**TO**
CURRENT_TIMESTAMP	**MATCH**	**TRAILING**
CURRENT_USER	**MAX**	**TRANSACTION**
CURSOR	**MIN**	**TRANSLATE**
DATE	**MINUTE**	**TRANSLATION**
DAY	**MODULE**	**TRIM**
DEALLOCATE	**MONTH**	**TRUE**
DEC	**NAMES**	**UNION**
DECIMAL	**NATIONAL**	**UNIQUE**
DECLARE	**NATURAL**	**UNKNOWN**
DEFAULT	**NCHAR**	**UPDATE**
DEFERRABLE	**NEXT**	**UPPER**
DEFERRED	**NO**	**USAGE**
DELETE	**NONE**	**USER**
DESC	**NOT**	**USING**
DESCRIBE	**NULL**	**VALUE**
DESCRIPTOR	**NULLIF**	**VALUES**
DIAGNOSTICS	**NUMERIC**	**VARCHAR**
DISCONNECT	**OCTET_LENGTH**	**VARYING**
DISTINCT	**OF**	**VIEW**
DOMAIN	**ON**	**WHEN**
DOUBLE	**ONLY**	**WHENEVER**

DROP	**OPEN**	**WHERE**
ELSE	**OPTION**	**WITH**
END	**OR**	**WORK**
END-EXEC	**ORDER**	**WRITE**
ESCAPE	**OUTER**	**YEAR**
EXCEPT	**OUTPUT**	**ZONE**
EXCEPTION		

Future Keywords

The following keywords could be reserved in future releases of SQL Server as new features are implemented. Consider avoiding the use of these words as identifiers.

ABSOLUTE	HOST	RELATIVE
ACTION	HOUR	RELEASE
ADMIN	IGNORE	RESULT
AFTER	IMMEDIATE	RETURNS
AGGREGATE	INDICATOR	ROLE
ALIAS	INITIALIZE	ROLLUP
ALLOCATE	INITIALLY	ROUTINE
ARE	INOUT	ROW
ARRAY	INPUT	ROWS
ASENSITIVE	INT	SAVEPOINT
ASSERTION	INTEGER	SCROLL
ASYMMETRIC	INTERSECTION	SCOPE
AT	INTERVAL	SEARCH
ATOMIC	ISOLATION	SECOND
BEFORE	ITERATE	SECTION

BINARY	LANGUAGE	SENSITIVE
BIT	LARGE	SEQUENCE
BLOB	LAST	SESSION
BOOLEAN	LATERAL	SETS
BOTH	LEADING	SIMILAR
BREADTH	LESS	SIZE
CALL	LEVEL	SMALLINT
CALLED	LIKE_REGEX	SPACE
CARDINALITY	LIMIT	SPECIFIC
CASCADED	LN	SPECIFICTYPE
CAST	LOCAL	SQL
CATALOG	LOCALTIME	SQLEXCEPTION
CHAR	LOCALTIMESTAMP	SQLSTATE
CHARACTER	LOCATOR	SQLWARNING
CLASS	MAP	START
CLOB	MATCH	STATE
COLLATION	MEMBER	STATEMENT
COLLECT	METHOD	STATIC
COMPLETION	MINUTE	STDDEV_POP
CONDITION	MOD	STDDEV_SAMP
CONNECT	MODIFIES	STRUCTURE
CONNECTION	MODIFY	SUBMULTISET
CONSTRAINTS	MODULE	SUBSTRING_REGEX
CONSTRUCTOR	MONTH	SYMMETRIC
CORR	MULTISET	SYSTEM
CORRESPONDING	NAMES	TEMPORARY

COVAR_POP	NATURAL	TERMINATE
COVAR_SAMP	NCHAR	THAN
CUBE	NCLOB	TIME
CUME_DIST	NEW	TIMESTAMP
CURRENT_CATALOG	NEXT	TIMEZONE_HOUR
CURRENT_DEFAULT_TRANSFORM_GROUP	NO	TIMEZONE_MINUTE
CURRENT_PATH	NONE	TRAILING
CURRENT_ROLE	NORMALIZE	TRANSLATE_REGEX
CURRENT_SCHEMA	NUMERIC	TRANSLATION
CURRENT_TRANSFORM_GROUP_FOR_TYPE	OBJECT	TREAT
CYCLE	OCCURRENCES_REGEX	TRUE
DATA	OLD	UESCAPE
DATE	ONLY	UNDER
DAY	OPERATION	UNKNOWN
DEC	ORDINALITY	UNNEST
DECIMAL	OUT	USAGE
DEFERRABLE	OVERLAY	USING
DEFERRED	OUTPUT	VALUE
DEPTH	PAD	VAR_POP
DEREF	PARAMETER	VAR_SAMP
DESCRIBE	PARAMETERS	VARCHAR
DESCRIPTOR	PARTIAL	VARIABLE
DESTROY	PARTITION	WHENEVER
DESTRUCTOR	PATH	WIDTH_BUCKET
DETERMINISTIC	POSTFIX	WITHOUT

DICTIONARY	PREFIX	WINDOW
DIAGNOSTICS	PREORDER	WITHIN
DISCONNECT	PREPARE	WORK
DOMAIN	PERCENT_RANK	WRITE
DYNAMIC	PERCENTILE_CONT	XMLAGG
EACH	PERCENTILE_DISC	XMLATTRIBUTES
ELEMENT	POSITION_REGEX	XMLBINARY
END-EXEC	PRESERVE	XMLCAST
EQUALS	PRIOR	XMLCOMMENT
EVERY	PRIVILEGES	XMLCONCAT
EXCEPTION	RANGE	XMLDOCUMENT
FALSE	READS	XMLELEMENT
FILTER	REAL	XMLEXISTS
FIRST	RECURSIVE	XMLFOREST
FLOAT	REF	XMLITERATE
FOUND	REFERENCING	XMLNAMESPACES
FREE	REGR_AVGX	XMLPARSE
FULLTEXTTABLE	REGR_AVGY	XMLPI
FUSION	REGR_COUNT	XMLQUERY
GENERAL	REGR_INTERCEPT	XMLSERIALIZE
GET	REGR_R2	XMLTABLE
GLOBAL	REGR_SLOPE	XMLTEXT
GO	REGR_SXX	XMLVALIDATE
GROUPING	REGR_SXY	YEAR
HOLD	REGR_SYY	ZONE

See Also

SET QUOTED_IDENTIFIER (Transact-SQL)
ALTER DATABASE Compatibility Level (Transact-SQL)

Transact-SQL Syntax Conventions (Transact-SQL)

8/27/2018 • 4 minutes to read • Edit Online

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

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.
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.

CONVENTION	USED FOR
<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 schema 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	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.

"Applies to" References

The Transact-SQL reference includes articles related to SQL Server (SQL Server 2008 through SQL Server 2017), Azure SQL Database, and Azure SQL Data Warehouse.

Near the top of each article is a section indicating which products support the subject of the article. If a product is omitted, then the feature described by the article is not available in that product. For example, availability groups were introduced in SQL Server 2012 (11.x). The **CREATE AVAILABILITY GROUP** article indicates it applies to SQL Server (SQL Server 2012 (11.x) through SQL Server 2017) because it does not apply to SQL Server 2008, SQL Server 2008 R2, or Azure SQL Database.

In some cases, the general subject of the article can be used in a product, but all of the arguments are not supported. For example, contained database users were introduced in SQL Server 2012 (11.x). The **CREATE USER** statement can be used in any SQL Server

product, however the **WITH PASSWORD** syntax cannot be used with older versions. In this case, additional **Applies to** sections are inserted into the appropriate argument descriptions in the body of the article.

See Also

Transact-SQL Reference (Database Engine)
Reserved Keywords (Transact SQL)
Transact-SQL Design Issues
Transact-SQL Naming Issues
Transact-SQL Performance Issues