Programming in Transact-SQL

1. Reserved Keywords

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 reserved keywords.

ADD	EXISTS	PRECISION
ALL	EXIT	PRIMARY
ALTER	EXTERNAL	PRINT
AND	FETCH	PROC
ANY	FILE	PROCEDURE
AS	FILLFACTOR	PUBLIC
ASC	FOR	RAISERROR
AUTHORIZATION	FOREIGN	READ
BACKUP	FREETEXT	READTEXT
BEGIN	FREETEXTTABLE	RECONFIGURE
BETWEEN	FROM	REFERENCES
BREAK	FULL	REPLICATION
BROWSE	FUNCTION	RESTORE
BULK	GOTO	RESTRICT
BY	GRANT	RETURN
CASCADE	GROUP	REVERT
CASE	HAVING	REVOKE
CHECK	HOLDLOCK	RIGHT
CHECKPOINT	IDENTITY	ROLLBACK

CLOSE	IDENTITY_INSERT	ROWCOUNT
CLUSTERED	IDENTITYCOL	ROWGUIDCOL
COALESCE	IF	RULE
COLLATE	IN	SAVE
COLUMN	INDEX	SCHEMA
COMMIT	INNER	SECURITYAUDIT
COMPUTE	INSERT	SELECT
CONSTRAINT	INTERSECT	SESSION_USER
CONTAINS	INTO	SET
CONTAINSTABLE	IS	SETUSER
CONTINUE	JOIN	SHUTDOWN
CONVERT	KEY	SOME
CREATE	KILL	STATISTICS
CROSS	LEFT	SYSTEM_USER
CURRENT	LIKE	TABLE
CURRENT_DATE	LINENO	TABLESAMPLE
CURRENT_TIME	LOAD	TEXTSIZE
CURRENT_TIMESTAMP	MERGE	THEN
CURRENT_USER	NATIONAL	TO
CURSOR	NOCHECK	TOP
DATABASE	NONCLUSTERED	TRAN
DBCC	NOT	TRANSACTION
DEALLOCATE	NULL	TRIGGER
DECLARE	NULLIF	TRUNCATE
DEFAULT	OF	TSEQUAL
DELETE	OFF	UNION

DENY	OFFSETS	UNIQUE
DESC	ON	UNPIVOT
DISK	OPEN	UPDATE
DISTINCT	OPENDATASOURCE	UPDATETEXT
DISTRIBUTED	OPENQUERY	USE
DOUBLE	OPENROWSET	USER
DROP	OPENXML	VALUES
DUMP	OPTION	VARYING
ELSE	OR	VIEW
END	ORDER	WAITFOR
ERRLVL	OUTER	WHEN
ESCAPE	OVER	WHERE
EXCEPT	PERCENT	WHILE
EXEC	PIVOT	WITH
EXECUTE	PLAN	WRITETEXT

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. For more information, see Using Identifiers As Object Names.

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

2. Transact-SQL Syntax Conventions

The following table lists and describes conventions that are used in the syntax diagrams in the Transact-SQL Reference.

Convention	Used for
UPPERCASE	Transact-SQL keywords.
italic	User-supplied parameters of Transact-SQL syntax.
bold	Database names, table names, column names, index names, stored procedures, utilities, data type names, and text that must be typed exactly as shown.
underline	Indicates the default value applied when the clause that contains the underlined value is omitted from the statement.
| (vertical bar)	Separates syntax items enclosed in brackets or braces. You can use only one of the items.
[ ] (brackets)	Optional syntax items. Do not type the brackets.
{ } (braces)	Required syntax items. Do not type the braces.
[,...*n*]	Indicates the preceding item can be repeated *n* number of times. The occurrences are separated by commas.
[...*n*]	Indicates the preceding item can be repeated *n* number of times. The occurrences are separated by blanks.
;	Transact-SQL statement terminator.Although the semicolon is not required for most statements in this version of SQL Server, it will be required in a future version.
<label> ::=	The name for a block of syntax. This convention is used to group and label sections of lengthy syntax or a unit of syntax that can be used in more than one location within a statement. Each location in which the block of syntax can be used is indicated with the label enclosed in chevrons: <label>. A set is a collection of expressions, for example <grouping set>; and a list is a collection of sets, for example <composite element list>.

Multipart Names

Unless specified otherwise, all Transact-SQL references to the name of a database object can be a four-part name in the following form:

server_name.[database_name].[schema_name].object_name

| database_name.[schema_name].object_name

| schema_name.object_name

| object_name

server_name

Specifies a linked server name or remote server name.

database_name

Specifies the name of a SQL Server database when the object resides in a local instance of SQL Server. When the object is in a linked server, *database_name* specifies an OLE DB catalog.

schema_name

Specifies the name of the schema that contains the object if the object is in a SQL Server database. When the object is in a linked server, *schema_name* specifies an OLE DB schema name. For more information about schemas, see User-Schema Separation.

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. For more information, see Server-Side Programming with Unicode.

3. Built-in Functions

SQL Server provides many built-in functions that you can use in queries to return data or perform operations on data.

Types of Functions

Function	Description
Rowset Functions	Return an object that can be used like table references in an SQL statement.
Aggregate Functions	Operate on a collection of values but return a single, summarizing value.
Ranking Functions	Return a ranking value for each row in a partition.
Scalar Functions (Described below)	Operate on a single value and then return a single value. Scalar functions can be used wherever an expression is valid.

Scalar Functions

Function category	Description
Configuration Functions	Return information about the current configuration.
Cryptographic Functions	Support encryption, decryption, digital signing, and the validation of digital signatures.
Cursor Functions	Return information about cursors.
Data Type Functions	Return information about identity values and other data type values.
Date and Time Data Types and Functions	Perform operations on a date and time input values and return string, numeric, or date and time values.
Mathematical Functions	Perform calculations based on input values provided as parameters to the functions, and return numeric values.
Metadata Functions	Return information about the database and database objects.
ODBC Scalar Functions	Return information about scalar ODBC functions in a Transact-SQL statement.

Replication Functions	Return information that is used to administer, monitor, and maintain a replication topology
Security Functions	Return information about users and roles.
String Functions	Perform operations on a string (char or varchar) input value and return a string or numeric value.
System Functions	Perform operations and return information about values, objects, and settings in an instance of SQL Server.
System Statistical Functions	Return statistical information about the system.
Text and Image Functions	Perform operations on text or image input values or columns, and return information about the value.
Trigger Functions	Return information about triggers.

Function Determinism

SQL Server built-in functions are either deterministic or nondeterministic. Functions are deterministic when they always return the same result any time they are called by using a specific set of input values. Functions are nondeterministic when they could return different results every time they are called, even with the same specific set of input values. For more information, see Deterministic and Nondeterministic Functions

Function Collation

Functions that take a character string input and return a character string output use the collation of the input string for the output.

Functions that take non-character inputs and return a character string use the default collation of the current database for the output.

Functions that take multiple character string inputs and return a character string use the rules of collation precedence to set the collation of the output string. For more information, see Collation Precedence.

3.1. Aggregate Functions

Aggregate functions perform a calculation on a set of values and return a single value. Except for COUNT, aggregate functions ignore null values. Aggregate functions are frequently used with the GROUP BY clause of the SELECT statement.

All aggregate functions are deterministic. This means aggregate functions return the same value any time that they are called by using a specific set of input values. For more information about function

determinism, see Deterministic and Nondeterministic Functions. The OVER clause may follow all aggregate functions except CHECKSUM.

Aggregate functions can be used as expressions only in the following:

- The select list of a SELECT statement (either a subquery or an outer query).
- A COMPUTE or COMPUTE BY clause.
- A HAVING clause.

Transact-SQL provides the following aggregate functions:

AVG	MIN
CHECKSUM_AGG	OVER Clause
COUNT	ROWCOUNT_BIG
COUNT_BIG	STDEV
GROUPING	STDEVP
GROUPING_ID	SUM
MAX	VAR
	VARP

3.1.1. AVG

Returns the average of the values in a group. Null values are ignored. May be followed by the OVER clause.

Syntax

```
AVG ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the aggregate function to all values. ALL is the default.

DISTINCT

Specifies that AVG be performed only on each unique instance of a value, regardless of how many times the value occurs.

expression

Is an expression of the exact numeric or approximate numeric data type category, except for the bit data type. Aggregate functions and subqueries are not permitted.

Return Types

The return type is determined by the type of the evaluated result of *expression*.

Expression result	Return type
tinyint	int
smallint	int
int	int
bigint	bigint
decimal category (p, s)	decimal(38, s) divided by decimal(10, 0)
money and smallmoney category	money
float and real category	float

Remarks

If the data type of *expression* is an alias data type, the return type is also of the alias data type. However, if the base data type of the alias data type is promoted, for example from tinyint to int, the return value is of the promoted data type and not the alias data type.

AVG () computes the average of a set of values by dividing the sum of those values by the count of nonnull values. If the sum exceeds the maximum value for the data type of the return value an error will be returned.

Examples

A. Using the SUM and AVG functions for calculations

The following example calculates the average vacation hours and the sum of sick leave hours that the vice presidents of Adventure Works Cycles have used. Each of these aggregate functions produces a single summary value for all the retrieved rows.

```
USE AdventureWorks2008R2;
GO
SELECT AVG(VacationHours)AS 'Average vacation hours',
    SUM  (SickLeaveHours) AS 'Total sick leave hours'
FROM HumanResources.Employee
WHERE JobTitle LIKE 'Vice President%';
```

Here is the result set.

```
Average vacation hours Total sick leave hours
---------------------- ----------------------
25 97
(1 row(s) affected)
```

B. Using the SUM and AVG functions with a GROUP BY clause

When used with a GROUP BY clause, each aggregate function produces a single value for each group, instead of for the whole table. The following example produces summary values for each sales territory. The summary lists the average bonus received by the sales people in each territory and the sum of year-to-date sales for each territory.

```
USE AdventureWorks2008R2;
GO
SELECT TerritoryID, AVG(Bonus)as 'Average bonus', SUM(SalesYTD) as 'YTD sales
'
FROM Sales.SalesPerson
GROUP BY TerritoryID;
GO
```

C. Using AVG with DISTINCT

The following statement returns the average list price of products.

```
USE AdventureWorks2008R2;
GO
SELECT AVG(DISTINCT ListPrice)
FROM Production.Product;
```

Here is the result set.

```
-----------------------------
437.4042
(1 row(s) affected)
```

D. Using AVG without DISTINCT

Without DISTINCT, the AVG function finds the average list price of all products in the Product table.

```
USE AdventureWorks2008R2;
GO
SELECT AVG(ListPrice)
FROM Production.Product;
```

Here is the result set.

```
-----------------------------
438.6662
(1 row(s) affected)
```

3.1.2. CHECKSUM_AGG

Returns the checksum of the values in a group. Null values are ignored. Can be followed by the OVER clause.

Syntax

```
CHECKSUM_AGG ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the aggregate function to all values. ALL is the default.

DISTINCT

Specifies that CHECKSUM_AGG returns the checksum of unique values.

expression

Is an integer expression. Aggregate functions and subqueries are not allowed.

Return Types

Returns the checksum of all *expression* values as int.

Remarks

CHECKSUM_AGG can be used to detect changes in a table.

The order of the rows in the table does not affect the result of CHECKSUM_AGG. Also, CHECKSUM_AGG functions may be used with the DISTINCT keyword and the GROUP BY clause.

If one of the values in the expression list changes, the checksum of the list also generally changes. However, there is a small chance that the checksum will not change.

CHECKSUM_AGG has similar functionality with other aggregate functions. For more information, see Aggregate Functions.

Examples

The following example uses CHECKSUM_AGG to detect changes in the Quantity column of the ProductInventory table in the AdventureWorks2008R2 database.

```
--Get the checksum value before the column value is changed.
USE AdventureWorks2008R2;
GO
SELECT CHECKSUM_AGG(CAST(Quantity AS int))
FROM Production.ProductInventory;
GO
```

Here is the result set.

262

```
UPDATE Production.ProductInventory
SET Quantity=125
WHERE Quantity=100;
GO
--Get the checksum of the modified column.
SELECT CHECKSUM_AGG(CAST(Quantity AS int))
FROM Production.ProductInventory;
```

Here is the result set.

287

3.1.3. COUNT_BIG

Returns the number of items in a group. COUNT_BIG works like the COUNT function. The only difference between the two functions is their return values. COUNT_BIG always returns a bigint data type value. COUNT always returns an int data type value. May be followed by the OVER Clause.

Syntax

```
COUNT_BIG ( { [ ALL | DISTINCT ] expression } | * )
```

Arguments

ALL

Applies the aggregate function to all values. ALL is the default.

DISTINCT

Specifies that COUNT_BIG returns the number of unique nonnull values.

expression

Is an expression of any type. Aggregate functions and subqueries are not permitted.

*

Specifies that all rows should be counted to return the total number of rows in a table. COUNT_BIG(*) takes no parameters and cannot be used with DISTINCT. COUNT_BIG(*) does not require an *expression* parameter because, by definition, it does not use information about any particular column. COUNT_BIG(*) returns the number of rows in a specified table without getting rid of duplicates. It counts each row separately. This includes rows that contain null values.

Return Types

bigint

Remarks

COUNT_BIG(*) returns the number of items in a group. This includes NULL values and duplicates.

COUNT_BIG(ALL *expression*) evaluates *expression* for each row in a group and returns the number of nonnull values.

COUNT_BIG(DISTINCT *expression*) evaluates *expression* for each row in a group and returns the number of unique, nonnull values.

3.1.4. COUNT

Returns the number of items in a group. COUNT works like the COUNT_BIG function. The only difference between the two functions is their return values. COUNT always returns an int data type value. COUNT_BIG always returns a bigint data type value. May be followed by the OVER clause.

Syntax

```
COUNT ( { [ [ ALL | DISTINCT ] expression ] | * } )
```

Arguments

ALL

Applies the aggregate function to all values. ALL is the default.

DISTINCT

Specifies that COUNT returns the number of unique nonnull values.

expression

Is an expression of any type except text, image, or ntext. Aggregate functions and subqueries are not permitted.

*

Specifies that all rows should be counted to return the total number of rows in a table. COUNT(*) takes no parameters and cannot be used with DISTINCT. COUNT(*) does not require an *expression* parameter because, by definition, it does not use information about any particular column. COUNT(*) returns the number of rows in a specified table without getting rid of duplicates. It counts each row separately. This includes rows that contain null values.

Return Types

int

Remarks

COUNT(*) returns the number of items in a group. This includes NULL values and duplicates.

COUNT(ALL *expression*) evaluates *expression* for each row in a group and returns the number of nonnull values.

COUNT(DISTINCT *expression*) evaluates *expression* for each row in a group and returns the number of unique, nonnull values.

For return values greater than 2^31-1, COUNT produces an error. Use COUNT_BIG instead.

Examples

A. Using COUNT and DISTINCT

The following example lists the number of different titles that an employee who works at Adventure Works Cycles can hold.

```
USE AdventureWorks2008R2;
GO
SELECT COUNT(DISTINCT JobTitle)
FROM HumanResources.Employee;
GO
```

Here is the result set.

```
-----------
67
(1 row(s) affected)
```

B. Using COUNT(*)

The following example finds the total number of employees who work at Adventure Works Cycles.

```
USE AdventureWorks2008R2;
GO
SELECT COUNT(*)
FROM HumanResources.Employee;
GO
```

Here is the result set.

```
-----------
290
(1 row(s) affected)
```

C. Using COUNT(*) with other aggregates

The following example shows that COUNT(*) can be combined with other aggregate functions in the select list.

```
USE AdventureWorks2008R2;
GO
SELECT COUNT(*), AVG(Bonus)
FROM Sales.SalesPerson
WHERE SalesQuota > 25000;
GO
```

Here is the result set.

```
----------- ---------------------
14 3472.1428
(1 row(s) affected)
```

3.1.5. GROUPING

Indicates whether a specified column expression in a GROUP BY list is aggregated or not. GROUPING returns 1 for aggregated or 0 for not aggregated in the result set. GROUPING can be used only in the SELECT <select> list, HAVING, and ORDER BY clauses when GROUP BY is specified.

Syntax

```
GROUPING ( <column_expression> )
```

Arguments

<column_expression>

Is a column or an expression that contains a column in a GROUP BY clause.

Return Types

tinyint

Remarks

GROUPING is used to distinguish the null values that are returned by ROLLUP, CUBE or GROUPING SETS from standard null values. The NULL returned as the result of a ROLLUP, CUBE or GROUPING SETS operation is a special use of NULL. This acts as a column placeholder in the result set and means all.

Examples

The following example groups SalesQuota and aggregates SaleYTD amounts. The GROUPING function is applied to the SalesQuota column.

```
USE AdventureWorks2008R2;
GO
SELECT SalesQuota, SUM(SalesYTD) 'TotalSalesYTD', GROUPING(SalesQuota) AS 'Gr
ouping'
FROM Sales.SalesPerson
GROUP BY SalesQuota WITH ROLLUP;
GO
```

The result set shows two null values under SalesQuota. The first NULL represents the group of null values from this column in the table. The second NULL is in the summary row added by the ROLLUP operation. The summary row shows the TotalSalesYTD amounts for all SalesQuota groups and is indicated by 1 in the Grouping column.

Here is the result set.

```
SalesQuota TotalSalesYTD Grouping
--------- ------------ --------
NULL 1533087.5999 0
250000.00 33461260.59 0
300000.00 9299677.9445 0
NULL 44294026.1344 1
(4 row(s) affected)
```

3.1.6. GROUPING_ID

Is a function that computes the level of grouping. GROUPING_ID can be used only in the SELECT <select> list, HAVING, or ORDER BY clauses when GROUP BY is specified.

Syntax

```
GROUPING_ID ( <column_expression>[ ,...n ] )
```

Arguments

<column_expression>

Is a *column_expression* in a GROUP BY clause.

Return Type

int

Remarks

The GROUPING_ID <column_expression> must exactly match the expression in the GROUP BY list. For example, if you are grouping by DATEPART (yyyy, <*column name*>), use GROUPING_ID (DATEPART (yyyy, <*column name*>)); or if you are grouping by <*column name*>, use GROUPING_ID (<*column name*>).

Comparing GROUPING_ID () to GROUPING ()

GROUPING_ID (<column_expression> [**,**...*n*]) inputs the equivalent of the GROUPING (<column_expression>) return for each column in its column list in each output row as a string of ones and zeros. GROUPING_ID interprets that string as a base-2 number and returns the equivalent integer. For example consider the following statement: SELECT a, b, c, SUM(d), GROUPING_ID(a,b,c) FROM T GROUP BY <group by list>. The following table shows the GROUPING_ID () input and output values.

Columns aggregated	GROUPING_ID (a, b, c) input = GROUPING(a) + GROUPING(b) + GROUPING(c)	GROUPING_ID () output
a	100	4
b	010	2
c	001	1
ab	110	6
ac	101	5

bc	011	3
abc	111	7

Technical Definition of GROUPING_ID ()

Each GROUPING_ID argument must be an element of the GROUP BY list. GROUPING_ID () returns an integer bitmap whose lowest N bits may be lit. A lit bit indicates the corresponding argument is not a grouping column for the given output row. The lowest-order bit corresponds to argument N, and the N-1th lowest-order bit corresponds to argument 1.

GROUPING_ID () Equivalents

For a single grouping query, GROUPING (<column_expression>) is equivalent to GROUPING_ID (<column_expression>), and both return 0.

For example, the following statements are equivalent:

```
SELECT GROUPING_ID(A,B)
FROM T
GROUP BY CUBE(A,B)
```

```
SELECT 3 FROM T GROUP BY ()
UNION ALL
SELECT 1 FROM T GROUP BY A
UNION ALL
SELECT 2 FROM T GROUP BY B
UNION ALL
SELECT 0 FROM T GROUP BY A,B
```

Examples

A. Using GROUPING_ID to identify grouping levels

The following example returns the count of employees by Name and Title, Name, and company total. GROUPING_ID() is used to create a value for each row in the Title column that identifies its level of aggregation.

```
USE AdventureWorks2008R2;
GO
SELECT D.Name
    ,CASE
    WHEN GROUPING_ID(D.Name, E.JobTitle) = 0 THEN E.JobTitle
    WHEN GROUPING_ID(D.Name, E.JobTitle) = 1 THEN N'Total: ' + D.Name
    WHEN GROUPING_ID(D.Name, E.JobTitle) = 3 THEN N'Company Total:'
        ELSE N'Unknown'
    END AS N'Job Title'
    ,COUNT(E.BusinessEntityID) AS N'Employee Count'
FROM HumanResources.Employee E
    INNER JOIN HumanResources.EmployeeDepartmentHistory DH
        ON E.BusinessEntityID = DH.BusinessEntityID
    INNER JOIN HumanResources.Department D
        ON D.DepartmentID = DH.DepartmentID
WHERE DH.EndDate IS NULL
```

```
    AND D.DepartmentID IN (12,14)
GROUP BY ROLLUP(D.Name, E.JobTitle);
```

B. Using GROUPING_ID to filter a result set

Simple Example

In the following code, to return only the rows that have a count of employees by title, remove the comment characters from HAVING GROUPING_ID(D.Name, E.JobTitle); = 0. To return only rows with a count of employees by department, remove the comment characters from HAVING GROUPING_ID(D.Name, E.JobTitle) = 1;.

```
USE AdventureWorks2008R2;
GO
SELECT D.Name
    ,E.JobTitle
    ,GROUPING_ID(D.Name, E.JobTitle) AS 'Grouping Level'
    ,COUNT(E.BusinessEntityID) AS N'Employee Count'
FROM HumanResources.Employee AS E
    INNER JOIN HumanResources.EmployeeDepartmentHistory AS DH
        ON E.BusinessEntityID = DH.BusinessEntityID
    INNER JOIN HumanResources.Department AS D
        ON D.DepartmentID = DH.DepartmentID
WHERE DH.EndDate IS NULL
    AND D.DepartmentID IN (12,14)
GROUP BY ROLLUP(D.Name, E.JobTitle)
--HAVING GROUPING_ID(D.Name, E.JobTitle) = 0; --All titles
--HAVING GROUPING_ID(D.Name, E.JobTitle) = 1; --Group by Name;
```

Here is the unfiltered result set.

Name	Title	Grouping Level	Employee Count	Name
Document Control	Control Specialist	0	2	Document Control
Document Control	Document Control Assistant	0	2	Document Control
Document Control	Document Control Manager	0	1	Document Control
Document Control	NULL	1	5	Document Control
Facilities and Maintenance	Facilities Administrative Assistant	0	1	Facilities and Maintenance
Facilities and	Facilities Manager	0	1	Facilities and

Maintenance				Maintenance
Facilities and Maintenance	Janitor	0	4	Facilities and Maintenance
Facilities and Maintenance	Maintenance Supervisor	0	1	Facilities and Maintenance
Facilities and Maintenance	NULL	1	7	Facilities and Maintenance
NULL	NULL	3	12	NULL

Complex Example

In the following example, GROUPING_ID() is used to filter a result set that contains multiple grouping levels by grouping level. Similar code can be used to create a view that has several grouping levels and a stored procedure that calls the view by passing a parameter that filters the view by grouping level.

```
USE AdventureWorks2008R2;
GO
DECLARE @Grouping nvarchar(50);
DECLARE @GroupingLevel smallint;
SET @Grouping = N'CountryRegionCode Total';

SELECT @GroupingLevel = (
    CASE @Grouping
        WHEN N'Grand Total'              THEN 15
        WHEN N'SalesPerson Total'        THEN 14
        WHEN N'Store Total'              THEN 13
        WHEN N'Store SalesPerson Total' THEN 12
        WHEN N'CountryRegionCode Total' THEN 11
        WHEN N'Group Total'              THEN 7
        ELSE N'Unknown'
    END);

SELECT
    T.[Group]
    ,T.CountryRegionCode
    ,S.Name AS N'Store'
    ,(SELECT P.FirstName + ' ' + P.LastName
        FROM Person.Person AS P
        WHERE P.BusinessEntityID = H.SalesPersonID)
        AS N'Sales Person'
    ,SUM(TotalDue)AS N'TotalSold'
    ,CAST(GROUPING(T.[Group])AS char(1)) +
        CAST(GROUPING(T.CountryRegionCode)AS char(1)) +
        CAST(GROUPING(S.Name)AS char(1)) +
        CAST(GROUPING(H.SalesPersonID)AS char(1))
```

```
        AS N'GROUPING base-2'
    ,GROUPING_ID((T.[Group])
        ,(T.CountryRegionCode),(S.Name),(H.SalesPersonID)
        ) AS N'GROUPING_ID'
    ,CASE
        WHEN GROUPING_ID(
            (T.[Group]),(T.CountryRegionCode)
            ,(S.Name),(H.SalesPersonID)
            ) = 15 THEN N'Grand Total'
        WHEN GROUPING_ID(
            (T.[Group]),(T.CountryRegionCode)
            ,(S.Name),(H.SalesPersonID)
            ) = 14 THEN N'SalesPerson Total'
        WHEN GROUPING_ID(
            (T.[Group]),(T.CountryRegionCode)
            ,(S.Name),(H.SalesPersonID)
            ) = 13 THEN N'Store Total'
        WHEN GROUPING_ID(
            (T.[Group]),(T.CountryRegionCode)
            ,(S.Name),(H.SalesPersonID)
            ) = 12 THEN N'Store SalesPerson Total'
        WHEN GROUPING_ID(
            (T.[Group]),(T.CountryRegionCode)
            ,(S.Name),(H.SalesPersonID)
            ) = 11 THEN N'CountryRegionCode Total'
        WHEN GROUPING_ID(
            (T.[Group]),(T.CountryRegionCode)
            ,(S.Name),(H.SalesPersonID)
            ) =  7 THEN N'Group Total'
        ELSE N'Error'
        END AS N'Level'
FROM Sales.Customer AS C
    INNER JOIN Sales.Store AS S
        ON C.StoreID  = S.BusinessEntityID
    INNER JOIN Sales.SalesTerritory AS T
        ON C.TerritoryID  = T.TerritoryID
    INNER JOIN Sales.SalesOrderHeader AS H
        ON C.CustomerID = H.CustomerID
GROUP BY GROUPING SETS ((S.Name,H.SalesPersonID)
    ,(H.SalesPersonID),(S.Name)
    ,(T.[Group]),(T.CountryRegionCode),()
    )
HAVING GROUPING_ID(
    (T.[Group]),(T.CountryRegionCode),(S.Name),(H.SalesPersonID)
    ) = @GroupingLevel
ORDER BY
    GROUPING_ID(S.Name,H.SalesPersonID),GROUPING_ID((T.[Group])
    ,(T.CountryRegionCode)
    ,(S.Name)
    ,(H.SalesPersonID))ASC;
```

C. Using GROUPING_ID () with ROLLUP and CUBE to identify grouping levels

The code in the following examples show using GROUPING() to compute the Bit Vector(base-2) column. GROUPING_ID() is used to compute the corresponding Integer Equivalent column. The column order in the GROUPING_ID() function is the opposite of the column order of the columns that are concatenated by the GROUPING() function.

In these examples, GROUPING_ID() is used to create a value for each row in the Grouping Level column to identify the level of grouping. Grouping levels are not always a consecutive list of integers that start with 1 (0, 1, 2,...*n*).

Note
GROUPING and GROUPING_ID can be used n a HAVING clause to filter a result set.

ROLLUP Example

In this example, all grouping levels do not appear as they do in the following CUBE example. If the order of the columns in the ROLLUP list is changed, the level values in the Grouping Level column will also have to be changed.

```
USE AdventureWorks2008R2;
GO
SELECT DATEPART(yyyy,OrderDate) AS N'Year'
    ,DATEPART(mm,OrderDate) AS N'Month'
    ,DATEPART(dd,OrderDate) AS N'Day'
    ,SUM(TotalDue) AS N'Total Due'
    ,CAST(GROUPING(DATEPART(dd,OrderDate))AS char(1)) +
        CAST(GROUPING(DATEPART(mm,OrderDate))AS char(1)) +
        CAST(GROUPING(DATEPART(yyyy,OrderDate))AS char(1))
     AS N'Bit Vector(base-2)'
    ,GROUPING_ID(DATEPART(yyyy,OrderDate)
        ,DATEPART(mm,OrderDate)
        ,DATEPART(dd,OrderDate))
        AS N'Integer Equivalent'
    ,CASE
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 0 THEN N'Year Month Day'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 1 THEN N'Year Month'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 2 THEN N'not used'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 3 THEN N'Year'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 4 THEN N'not used'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 5 THEN N'not used'
```

```
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 6 THEN N'not used'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 7 THEN N'Grand Total'
    ELSE N'Error'
    END AS N'Grouping Level'
FROM Sales.SalesOrderHeader
WHERE DATEPART(yyyy,OrderDate) IN(N'2007',N'2008')
    AND DATEPART(mm,OrderDate) IN(1,2)
    AND DATEPART(dd,OrderDate) IN(1,2)
GROUP BY ROLLUP(DATEPART(yyyy,OrderDate)
        ,DATEPART(mm,OrderDate)
        ,DATEPART(dd,OrderDate))
ORDER BY GROUPING_ID(DATEPART(mm,OrderDate)
    ,DATEPART(yyyy,OrderDate)
    ,DATEPART(dd,OrderDate)
    )
    ,DATEPART(yyyy,OrderDate)
    ,DATEPART(mm,OrderDate)
    ,DATEPART(dd,OrderDate);
```

Here is a partial result set.

Year	Month	Day	Total Due	Bit Vector (base-2)	Integer Equivalent	Grouping Level
2007	1	1	1497452.6066	000	0	Year Month Day
2007	1	2	21772.3494	000	0	Year Month Day
2007	2	1	2705653.5913	000	0	Year Month Day
2007	2	2	21684.4068	000	0	Year Month Day
2008	1	1	1908122.0967	000	0	Year Month Day
2008	1	2	46458.0691	000	0	Year Month Day
2008	2	1	3108771.9729	000	0	Year Month Day
2008	2	2	54598.5488	000	0	Year Month Day
2007	1	NULL	1519224.956	100	1	Year Month
2007	2	NULL	2727337.9981	100	1	Year Month

2008	1	NULL	1954580.1658	100	1	Year Month
2008	2	NULL	3163370.5217	100	1	Year Month
2007	NULL	NULL	4246562.9541	110	3	Year
2008	NULL	NULL	5117950.6875	110	3	Year
NULL	NULL	NULL	9364513.6416	111	7	Grand Total

CUBE Example

In this example, the GROUPING_ID() function is used to create a value for each row in the Grouping Level column to identify the level of grouping.

Unlike ROLLUP in the previous example, CUBE outputs all grouping levels. If the order of the columns in the CUBE list is changed, the level values in the Grouping Level column will also have to be changed.

```
USE AdventureWorks2008R2;
GO
SELECT DATEPART(yyyy,OrderDate) AS N'Year'
    ,DATEPART(mm,OrderDate) AS N'Month'
    ,DATEPART(dd,OrderDate) AS N'Day'
    ,SUM(TotalDue) AS N'Total Due'
    ,CAST(GROUPING(DATEPART(dd,OrderDate))AS char(1)) +
        CAST(GROUPING(DATEPART(mm,OrderDate))AS char(1)) +
        CAST(GROUPING(DATEPART(yyyy,OrderDate))AS char(1))
        AS N'Bit Vector(base-2)'
    ,GROUPING_ID(DATEPART(yyyy,OrderDate)
        ,DATEPART(mm,OrderDate)
        ,DATEPART(dd,OrderDate))
        AS N'Integer Equivalent'
    ,CASE
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 0 THEN N'Year Month Day'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 1 THEN N'Year Month'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 2 THEN N'Year Day'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 3 THEN N'Year'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 4 THEN N'Month Day'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
```

```
            ) = 5 THEN N'Month'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 6 THEN N'Day'
        WHEN GROUPING_ID(DATEPART(yyyy,OrderDate)
            ,DATEPART(mm,OrderDate),DATEPART(dd,OrderDate)
            ) = 7 THEN N'Grand Total'
    ELSE N'Error'
    END AS N'Grouping Level'
FROM Sales.SalesOrderHeader
WHERE DATEPART(yyyy,OrderDate) IN(N'2007',N'2008')
    AND DATEPART(mm,OrderDate) IN(1,2)
    AND DATEPART(dd,OrderDate) IN(1,2)
GROUP BY CUBE(DATEPART(yyyy,OrderDate)
    ,DATEPART(mm,OrderDate)
    ,DATEPART(dd,OrderDate))
ORDER BY GROUPING_ID(DATEPART(yyyy,OrderDate)
    ,DATEPART(mm,OrderDate)
    ,DATEPART(dd,OrderDate)
    )
    ,DATEPART(yyyy,OrderDate)
    ,DATEPART(mm,OrderDate)
    ,DATEPART(dd,OrderDate);
```

Here is a partial result set.

Year	Month	Day	Total Due	Bit Vector (base-2)	Integer Equivalent	Grouping Level
2007	1	1	1497452.6066	000	0	Year Month Day
2007	1	2	21772.3494	000	0	Year Month Day
2007	2	1	2705653.5913	000	0	Year Month Day
2007	2	2	21684.4068	000	0	Year Month Day
2008	1	1	1908122.0967	000	0	Year Month Day
2008	1	2	46458.0691	000	0	Year Month Day
2008	2	1	3108771.9729	000	0	Year Month Day
2008	2	2	54598.5488	000	0	Year Month Day
2007	1	NULL	1519224.956	100	1	Year Month
2007	2	NULL	2727337.9981	100	1	Year Month

2008	1	NULL	1954580.1658	100	1	Year Month
2008	2	NULL	3163370.5217	100	1	Year Month
2007	NULL	1	4203106.1979	010	2	Year Day
2007	NULL	2	43456.7562	010	2	Year Day
2008	NULL	1	5016894.0696	010	2	Year Day
2008	NULL	2	101056.6179	010	2	Year Day
2007	NULL	NULL	4246562.9541	110	3	Year
2008	NULL	NULL	5117950.6875	110	3	Year
NULL	1	1	3405574.7033	001	4	Month Day
NULL	1	2	68230.4185	001	4	Month Day
NULL	2	1	5814425.5642	001	4	Month Day
NULL	2	2	76282.9556	001	4	Month Day
NULL	1	NULL	3473805.1218	101	5	Month
NULL	2	NULL	5890708.5198	101	5	Month
NULL	NULL	1	9220000.2675	011	6	Day
NULL	NULL	2	144513.3741	011	6	Day
NULL	NULL	NULL	9364513.6416	111	7	Grand Total

3.1.7. MAX

Returns the maximum value in the expression. May be followed by the OVER clause.

Syntax

```
MAX ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the aggregate function to all values. ALL is the default.

DISTINCT

Specifies that each unique value is considered. DISTINCT is not meaningful with MAX and is available for ISO compatibility only.

expression

Is a constant, column name, or function, and any combination of arithmetic, bitwise, and string operators. MAX can be used with numeric, character, and datetime columns, but not with bit columns. Aggregate functions and subqueries are not permitted.

For more information, see Expressions.

Return Types

Returns a value same as *expression*.

Remarks

MAX ignores any null values.

For character columns, MAX finds the highest value in the collating sequence.

Examples

The following example returns the highest (maximum) tax rate.

```
USE AdventureWorks2008R2;
GO
SELECT MAX(TaxRate)
FROM Sales.SalesTaxRate;
GO
```

Here is the result set.

```
------
19.60
(1 row(s) affected)
Warning, null value eliminated from aggregate.
```

3.1.8. MIN

Returns the minimum value in the expression. May be followed by the OVER clause.

Syntax

```
MIN ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the aggregate function to all values. ALL is the default.

DISTINCT

Specifies that each unique value is considered. DISTINCT is not meaningful with MIN and is available for ISO compatibility only.

expression

Is a constant, column name, or function, and any combination of arithmetic, bitwise, and string operators. MIN can be used with numeric, char, varchar, or datetime columns, but not with bit columns. Aggregate functions and subqueries are not permitted.

For more information, see Expressions.

Return Types

Returns a value same as *expression*.

Remarks

MIN ignores any null values.

With character data columns, MIN finds the value that is lowest in the sort sequence.

Examples

The following example returns the lowest (minimum) tax rate.

```
USE AdventureWorks2008R2;
GO
SELECT MIN(TaxRate)
FROM Sales.SalesTaxRate;
GO
```

Here is the result set.

```
------------------
5.00
(1 row(s) affected)
```

3.1.9. ROWCOUNT_BIG

Returns the number of rows affected by the last statement executed. This function operates like @@ROWCOUNT, except the return type of ROWCOUNT_BIG is bigint.

Syntax

```
ROWCOUNT_BIG ( )
```

Return Types

bigint

Remarks

Following a SELECT statement, this function returns the number of rows returned by the SELECT statement.

Following an INSERT, UPDATE, or DELETE statement, this function returns the number of rows affected by the data modification statement.

Following statements that do not return rows, such as an IF statement, this function returns 0.

3.1.10. STDEV

Returns the statistical standard deviation of all values in the specified expression. May be followed by the OVER clause.

Syntax

```
STDEV ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the function to all values. ALL is the default.

DISTINCT

Specifies that each unique value is considered.

expression

Is a numeric expression. Aggregate functions and subqueries are not permitted. *expression* is an expression of the exact numeric or approximate numeric data type category, except for the bit data type.

Return Types

float

Remarks

If STDEV is used on all items in a SELECT statement, each value in the result set is included in the calculation. STDEV can be used with numeric columns only. Null values are ignored.

Examples

The following example returns the standard deviation for all bonus values in the SalesPerson table.

```
USE AdventureWorks2008R2;
GO
SELECT STDEV(Bonus)
FROM Sales.SalesPerson;
GO
```

3.1.11. STDEVP

Returns the statistical standard deviation for the population for all values in the specified expression. May be followed by the OVER clause.

Syntax

```
STDEVP ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the function to all values. ALL is the default.

DISTINCT

Specifies that each unique value is considered.

expression

Is a numeric expression. Aggregate functions and subqueries are not permitted. *expression* is an expression of the exact numeric or approximate numeric data type category, except for the bit data type.

Return Types

float

Remarks

If STDEVP is used on all items in a SELECT statement, each value in the result set is included in the calculation. STDEVP can be used with numeric columns only. Null values are ignored.

Examples

The following example returns the standard deviation for the population for all bonus values in the SalesPerson table.

```
USE AdventureWorks2008R2;
GO
SELECT STDEVP(Bonus)
FROM Sales.SalesPerson;
GO
```

3.1.12. SUM

Returns the sum of all the values, or only the DISTINCT values, in the expression. SUM can be used with numeric columns only. Null values are ignored. May be followed by the OVER Clause.

Syntax

```
SUM ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the aggregate function to all values. ALL is the default.

DISTINCT

Specifies that SUM return the sum of unique values.

expression

Is a constant, column, or function, and any combination of arithmetic, bitwise, and string operators. *expression* is an expression of the exact numeric or approximate numeric data type category, except for the bit data type. Aggregate functions and subqueries are not permitted. For more information, see Expressions.

Return Types

Returns the summation of all *expression* values in the most precise *expression* data type.

Expression result	Return type
tinyint	int
smallint	int
int	int
bigint	bigint
decimal category (p, s)	decimal(38, s)
money and smallmoney category	money
float and real category	float

Examples

A. Using SUM for aggregates and row aggregates

The following examples show the differences between aggregate functions and row aggregate functions. The first shows aggregate functions giving only the summary data, and the second shows row aggregate functions giving both the detail and summary data.

```
USE AdventureWorks2008R2;
GO
SELECT Color, SUM(ListPrice), SUM(StandardCost)
FROM Production.Product
WHERE Color IS NOT NULL
    AND ListPrice != 0.00
    AND Name LIKE 'Mountain%'
GROUP BY Color
ORDER BY Color;
GO
```

Here is the result set.

```
Color
--------------- --------------------- ---------------------
Black 27404.84 15214.9616
Silver 26462.84 14665.6792
White 19.00 6.7926
(3 row(s) affected)
```

```
USE AdventureWorks2008R2;
GO
SELECT Color, ListPrice, StandardCost
FROM Production.Product
WHERE Color IS NOT NULL
    AND ListPrice != 0.00
    AND Name LIKE 'Mountain%'
ORDER BY Color
COMPUTE SUM(ListPrice), SUM(StandardCost) BY Color;
GO
```

Here is the result set.

```
Color ListPrice StandardCost
--------------- --------------------- ---------------------
Black 2294.99 1251.9813
Black 2294.99 1251.9813
Black 2294.99 1251.9813
Black 1079.99 598.4354
Black 1079.99 598.4354
Black 1079.99 598.4354
Black 1079.99 598.4354
Black 3374.99 1898.0944
Black 3374.99 1898.0944
Black 3374.99 1898.0944
Black 3374.99 1898.0944
```

```
Black 539.99 294.5797
Black 539.99 294.5797
Black 539.99 294.5797
Black 539.99 294.5797
Black 539.99 294.5797
sum sum
--------------------- ---------------------
27404.84 15214.9616
Color ListPrice StandardCost
--------------- --------------------- ---------------------
Silver 2319.99 1265.6195
Silver 2319.99 1265.6195
Silver 2319.99 1265.6195
Silver 3399.99 1912.1544
Silver 3399.99 1912.1544
Silver 3399.99 1912.1544
Silver 3399.99 1912.1544
Silver 769.49 419.7784
Silver 769.49 419.7784
Silver 769.49 419.7784
Silver 769.49 419.7784
Silver 564.99 308.2179
Silver 564.99 308.2179
Silver 564.99 308.2179
Silver 564.99 308.2179
Silver 564.99 308.2179
sum sum
--------------------- ---------------------
26462.84 14665.6792
Color ListPrice StandardCost
--------------- --------------------- ---------------------
White 9.50 3.3963
White 9.50 3.3963
sum sum
--------------------- ---------------------
19.00 6.7926
(37 row(s) affected)
```

B. Calculating group totals with more than one column

The following example calculates the sum of the ListPrice and StandardCost for each color listed in the Product table.

```
USE AdventureWorks2008R2;
GO
SELECT Color, SUM(ListPrice), SUM(StandardCost)
FROM Production.Product
GROUP BY Color
ORDER BY Color;
GO
```

Here is the result set.
Color
--- --- ---
NULL 4182.32 2238.4755
Black 67436.26 38636.5002
Blue 24015.66 14746.1464
Grey 125.00 51.5625
Multi 478.92 272.2542
Red 53274.10 32610.7661
Silver 36563.13 20060.0483
Silver/Black 448.13 198.97
White 36.98 13.5172
Yellow 34527.29 21507.6521
(10 row(s) affected)

3.1.13. VAR

Returns the statistical variance of all values in the specified expression. May be followed by the OVER clause.

Syntax

```
VAR ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the function to all values. ALL is the default.

DISTINCT

Specifies that each unique value is considered.

expression

Is an expression of the exact numeric or approximate numeric data type category, except for the bit data type. Aggregate functions and subqueries are not permitted.

Return Types

float

Remarks

If VAR is used on all items in a SELECT statement, each value in the result set is included in the calculation. VAR can be used with numeric columns only. Null values are ignored.

Examples

The following example returns the variance for all bonus values in the SalesPerson table.

```
USE AdventureWorks2008R2;
GO
SELECT VAR(Bonus)
FROM Sales.SalesPerson;
GO
```

3.1.14. VARP

Returns the statistical variance for the population for all values in the specified expression. May be followed by the OVER clause.

Syntax

```
VARP ( [ ALL | DISTINCT ] expression )
```

Arguments

ALL

Applies the function to all values. ALL is the default.

DISTINCT

Specifies that each unique value is considered.

expression

Is an expression of the exact numeric or approximate numeric data type category, except for the bit data type. Aggregate functions and subqueries are not permitted.

Return Types

float

Remarks

If VARP is used on all items in a SELECT statement, each value in the result set is included in the calculation. VARP can be used with numeric columns only. Null values are ignored.

Examples

The following example returns the variance for the population for all bonus values in the SalesPerson table.

```
USE AdventureWorks2008R2;
GO
SELECT VARP(Bonus)
FROM Sales.SalesPerson;
GO
```

3.2. Collation Functions

The following scalar functions return information about collations.

3.2.1. COLLATIONPROPERTY

Returns the property of a specified collation.

Syntax

```
COLLATIONPROPERTY( collation_name , property )
```

Arguments

collation_name

Is the name of the collation. *collation_name* is nvarchar(128), and has no default.

property

Is the property of the collation. *property* is varchar(128), and can be any one of the following values:

Property name	Description
CodePage	Non-Unicode code page of the collation.
LCID	Windows LCID of the collation.
ComparisonStyle	Windows comparison style of the collation. Returns 0 for all binary collations .
Version	The version of the collation, derived from the version field of the collation ID. Returns 2, 1, or 0. All new collation versions introduced in SQL Server 2008 (collations with "100" in the name) return 2. All collation versions introduced in SQL Server 2005 (collations with "90" in the name) return 1. All other collations, introduced in earlier versions of SQL Server return 0.

Return Types

sql_variant

Examples

```
SELECT COLLATIONPROPERTY('Traditional_Spanish_CS_AS_KS_WS', 'CodePage')
```

Here is the result set.

1252

3.2.2. TERTIARY_WEIGHTS

Returns a binary string of weights for each character in a non-Unicode string expression defined with an SQL tertiary collation.

Syntax

```
TERTIARY_WEIGHTS( non_Unicode_character_string_expression )
```

Arguments

non_Unicode_character_string_expression

Is a string expression of type char, varchar, or varchar(max) defined on a tertiary SQL collation. For a list of these collations, see Remarks.

Return Types

TERTIARY_WEIGHTS returns varbinary when *non_Unicode_character_string_expression* is char or varchar, and returns varbinary(max) when *non_Unicode_character_string_expression* is varchar(max).

Remarks

TERTIARY_WEIGHTS returns NULL when *non_Unicode_character_string_expression* is not defined with an SQL tertiary collation. The following table shows the SQL tertiary collations.

Sort order ID	SQL collation
33	SQL_Latin1_General_Pref_CP437_CI_AS
34	SQL_Latin1_General_CP437_CI_AI
43	SQL_Latin1_General_Pref_CP850_CI_AS
44	SQL_Latin1_General_CP850_CI_AI
49	SQL_1xCompat_CP850_CI_AS
53	SQL_Latin1_General_Pref_CP1_CI_AS
54	SQL_Latin1_General_CP1_CI_AI

56	SQL_AltDiction_Pref_CP850_CI_AS
57	SQL_AltDiction_CP850_CI_AI
58	SQL_Scandinavian_Pref_CP850_CI_AS
82	SQL_Latin1_General_CP1250_CI_AS
84	SQL_Czech_CP1250_CI_AS
86	SQL_Hungarian_CP1250_CI_AS
88	SQL_Polish_CP1250_CI_AS
90	SQL_Romanian_CP1250_CI_AS
92	SQL_Croatian_CP1250_CI_AS
94	SQL_Slovak_CP1250_CI_AS
96	SQL_Slovenian_CP1250_CI_AS
106	SQL_Latin1_General_CP1251_CI_AS
108	SQL_Ukrainian_CP1251_CI_AS
113	SQL_Latin1_General_CP1253_CS_AS
114	SQL_Latin1_General_CP1253_CI_AS
130	SQL_Latin1_General_CP1254_CI_AS
146	SQL_Latin1_General_CP1256_CI_AS
154	SQL_Latin1_General_CP1257_CI_AS
156	SQL_Estonian_CP1257_CI_AS
158	SQL_Latvian_CP1257_CI_AS
160	SQL_Lithuanian_CP1257_CI_AS

183	SQL_Danish_Pref_CP1_CI_AS
184	SQL_SwedishPhone_Pref_CP1_CI_AS
185	SQL_SwedishStd_Pref_CP1_CI_AS
186	SQL_Icelandic_Pref_CP1_CI_AS

TERTIARY_WEIGHTS is intended for use in the definition of a computed column that is defined on the values of a char, varchar, or varchar(max) column. Defining an index on both the computed column and the char, varchar, or varchar(max) column can improve performance when the char, varchar, or varchar(max) column is specified in the ORDER BY clause of a query.

Examples

The following example creates a computed column in a table that applies the TERTIARY_WEIGHTS function to the values of a char column.

```
CREATE TABLE TertColTable
(Col1 char(15) COLLATE SQL_Latin1_General_Pref_CP437_CI_AS,
Col2 AS TERTIARY_WEIGHTS(Col1));
GO
```

3.3. Configuration Functions

The following scalar functions return information about current configuration option settings:

@@DATEFIRST	Returns the current value, for a session, of SET DATEFIRST.
@@DBTS	Returns the value of the current timestamp data type for the current database. This timestamp is guaranteed to be unique in the database.
@@LANGID	Returns the local language identifier (ID) of the language that is currently being used.
@@LANGUAGE	Returns the name of the language currently being used.
@@LOCK_TIMEOUT	Returns the current lock time-out setting in milliseconds for the current session.
@@MAX_CONNECTIONS	Returns the maximum number of simultaneous user

	connections allowed on an instance of SQL Server. The number returned is not necessarily the number currently configured.
@@MAX_PRECISION	Returns the precision level used by decimal and numeric data types as currently set in the server.
@@NESTLEVEL	Returns the nesting level of the current stored procedure execution (initially 0) on the local server. For information about nesting levels, see Nesting Stored Procedures.
@@OPTIONS	Returns information about the current SET options.
@@REMSERVER	Returns the name of the remote SQL Server database server as it appears in the login record.
@@SERVERNAME	Returns the name of the local server that is running SQL Server.
@@SERVICENAME	Returns the name of the registry key under which SQL Server is running. @@SERVICENAME returns 'MSSQLSERVER' if the current instance is the default instance; this function returns the instance name if the current instance is a named instance
@@SPID	Returns the session ID of the current user process.
@@TEXTSIZE	Returns the current value of the TEXTSIZE option.
@@VERSION	Returns version, processor architecture, build date, and operating system for the current installation of SQL Server.

All configuration functions are nondeterministic. This means these functions do not always return the same results every time they are called, even with the same set of input values. For more information about function determinism, see Deterministic and Nondeterministic Functions.

3.4. Cursor Functions

The following scalar functions return information about cursors:

@@CURSOR_ROWS	**CURSOR_STATUS**
@@FETCH_STATUS	

All cursor functions are nondeterministic. This means these functions do not always return the same results every time they are called, even with the same set of input values. For more information about function determinism, see Deterministic and Nondeterministic Functions.

3.4.1. @@CURSOR_ROWS

Returns the number of qualifying rows currently in the last cursor opened on the connection. To improve performance, SQL Server can populate large keyset and static cursors asynchronously. @@CURSOR_ROWS can be called to determine that the number of the rows that qualify for a cursor are retrieved at the time @@CURSOR_ROWS is called.

Syntax

@@CURSOR_ROWS

Return Types

integer

Return Value

Return value	Description
-m	The cursor is populated asynchronously. The value returned (*-m*) is the number of rows currently in the keyset.
-1	The cursor is dynamic. Because dynamic cursors reflect all changes, the number of rows that qualify for the cursor is constantly changing. It can never be definitely stated that all qualified rows have been retrieved.
0	No cursors have been opened, no rows qualified for the last opened cursor, or the last-opened cursor is closed or deallocated.
n	The cursor is fully populated. The value returned (*n*) is the total number of rows in the cursor.

Remarks

The number returned by @@CURSOR_ROWS is negative if the last cursor was opened asynchronously. Keyset-driver or static cursors are opened asynchronously if the value for sp_configure cursor threshold is greater than 0 and the number of rows in the cursor result set is greater than the cursor threshold.

Examples

The following example declares a cursor and uses SELECT to display the value of @@CURSOR_ROWS. The setting has a value of 0 before the cursor is opened and a value of -1 to indicate that the cursor keyset is populated asynchronously.

```
USE AdventureWorks2008R2;
GO
SELECT @@CURSOR_ROWS;
DECLARE Name_Cursor CURSOR FOR
SELECT LastName ,@@CURSOR_ROWS FROM Person.Person;
OPEN Name_Cursor;
FETCH NEXT FROM Name_Cursor;
SELECT @@CURSOR_ROWS;
CLOSE Name_Cursor;
DEALLOCATE Name_Cursor;
GO
```

Here are the result sets.

```
-----------
0
LastName
---------------
Sanchez
-----------
-1
```

3.4.2. @@FETCH_STATUS

Returns the status of the last cursor FETCH statement issued against any cursor currently opened by the connection.

Syntax

```
@@FETCH_STATUS
```

Return Type

integer

Return Value

Return value	Description
0	The FETCH statement was successful.
-1	The FETCH statement failed or the row was beyond the result set.
-2	The row fetched is missing.

Remarks

Because @@FETCH_STATUS is global to all cursors on a connection, use @@FETCH_STATUS carefully. After a FETCH statement is executed, the test for @@FETCH_STATUS must occur before any other FETCH statement is executed against another cursor. The value of @@FETCH_STATUS is undefined before any fetches have occurred on the connection.

For example, a user executes a FETCH statement from one cursor, and then calls a stored procedure that opens and processes the results from another cursor. When control is returned from the called stored procedure, @@FETCH_STATUS reflects the last FETCH executed in the stored procedure, not the FETCH statement executed before the stored procedure is called.

To retrieve the last fetch status of a specific cursor, query the fetch_status column of the sys.dm_exec_cursors dynamic management function.

Examples

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

```
DECLARE Employee_Cursor CURSOR FOR
SELECT BusinessEntityID, JobTitle
FROM AdventureWorks2008R2.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
```

3.4.3. CURSOR_STATUS

A scalar function that allows the caller of a stored procedure to determine whether or not the procedure has returned a cursor and result set for a given parameter.

Syntax

```
CURSOR_STATUS
     (
          { 'local' , 'cursor_name' }
          | { 'global' , 'cursor_name' }
          | { 'variable' , 'cursor_variable' }
     )
```

Arguments

'local'

Specifies a constant that indicates the source of the cursor is a local cursor name.

'*cursor_name*'

Is the name of the cursor. A cursor name must conform to the rules for identifiers.

'global'

Specifies a constant that indicates the source of the cursor is a global cursor name.

'variable'

Specifies a constant that indicates the source of the cursor is a local variable.

'*cursor_variable*'

Is the name of the cursor variable. A cursor variable must be defined using the cursor data type.

Return Types

smallint

Return value	Cursor name	Cursor variable
1	The result set of the cursor has at least one row. For insensitive and keyset cursors, the result set has at least one row. For dynamic cursors, the result set can have zero, one, or more rows.	The cursor allocated to this variable is open. For insensitive and keyset cursors, the result set has at least one row. For dynamic cursors, the result set can have zero, one, or more rows.
0	The result set of the cursor is empty.*	The cursor allocated to this variable is open, but the result set is definitely empty.*
-1	The cursor is closed.	The cursor allocated to this variable is closed.
-2	Not applicable.	Can be: No cursor was assigned to this OUTPUT variable by the previously called procedure. A cursor was assigned to this OUTPUT variable by the previously called procedure, but it was in a closed state upon completion of the procedure. Therefore, the cursor is deallocated and not returned to the calling

		procedure. There is no cursor assigned to a declared cursor variable.
-3	A cursor with the specified name does not exist.	A cursor variable with the specified name does not exist, or if one exists it has not yet had a cursor allocated to it.

*Dynamic cursors never return this result.

Examples

The following example uses the CURSOR_STATUS function to show the status of a cursor before and after it is opened and closed.

```
CREATE TABLE #TMP
(
   ii int
)
GO

INSERT INTO #TMP(ii) VALUES(1)
INSERT INTO #TMP(ii) VALUES(2)
INSERT INTO #TMP(ii) VALUES(3)

GO

--Create a cursor.
DECLARE cur CURSOR
FOR SELECT * FROM #TMP

--Display the status of the cursor before and after opening
--closing the cursor.

SELECT CURSOR_STATUS('global','cur') AS 'After declare'
OPEN cur
SELECT CURSOR_STATUS('global','cur') AS 'After Open'
CLOSE cur
SELECT CURSOR_STATUS('global','cur') AS 'After Close'

--Remove the cursor.
DEALLOCATE cur

--Drop the table.
DROP TABLE #TMP
```

Here is the result set.

```
After declare
--------------
```

```
-1
After Open
----------
1
After Close
-----------
-1
```

3.5. Data Type Functions

The following scalar functions return information about various data type values.

DATALENGTH	Returns the number of bytes used to represent any expression.
IDENT_CURRENT	Returns the last identity value generated for a specified table or view. The last identity value generated can be for any session and any scope.
IDENT_INCR	Returns the increment value (returned as numeric (**@@**MAXPRECISION,0)) specified during the creation of an identity column in a table or view that has an identity column.
IDENT_SEED	Returns the original seed value (returned as numeric(**@@**MAXPRECISION,0)) that was specified when an identity column in a table or a view was created. Changing the current value of an identity column by using DBCC CHECKIDENT does not change the value returned by this function.
IDENTITY (Function)	Is used only in a SELECT statement with an INTO *table* clause to insert an identity column into a new table. Although similar, the IDENTITY function is not the IDENTITY property that is used with CREATE TABLE and ALTER TABLE.
SQL_VARIANT_PROPERTY	Returns the base data type and other information about a sql_variant value.

3.6. Date and Time Functions

The following sections in this topic provide an overview of all Transact-SQL date and time data types and functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Date and Time Data Types

The Transact-SQL date and time data types are listed in the following table.

Data type	Format	Range	Accuracy	Storage size (bytes)	User-defined fractional second precision	Time zone offset
time	hh:mm:ss[.nnnnnnn]	00:00:00.0000000 through 23:59:59.9999999	100 nanoseconds	3 to 5	Yes	No
date	YYYY-MM-DD	0001-01-01 through 9999-12-31	1 day	3	No	No
smalldatetime	YYYY-MM-DD hh:mm:ss	1900-01-01 through 2079-06-06	1 minute	4	No	No
datetime	YYYY-MM-DD hh:mm:ss[.nnn]	1753-01-01 through 9999-12-31	0.00333 second	8	No	No
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999	100 nanoseconds	6 to 8	Yes	No
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999 (in UTC)	100 nanoseconds	8 to 10	Yes	Yes

Note

The Transact-SQL rowversion data type is not a date or time data type. timestamp is a deprecated synonym for rowversion.

Date and Time Functions

The Transact-SQL date and time functions are listed in the following tables. For more information about determinism, see Deterministic and Nondeterministic Functions.

Functions That Get System Date and Time Values

All system date and time values are derived from the operating system of the computer on which the instance of SQL Server is running.

Higher-Precision System Date and Time Functions

SQL Server 2008 R2 obtains the date and time values by using the **GetSystemTimeAsFileTime()** Windows API. The accuracy depends on the computer hardware and version of Windows on which the instance of SQL Server is running. The precision of this API is fixed at 100 nanoseconds. The accuracy can be determined by using the **GetSystemTimeAdjustment()** Windows API.

Function	Syntax	Return value	Return data type	Determinism
SYSDATETIME	SYSDATETIME ()	Returns a datetime2(7) value that contains the date and time of the computer on which the instance of SQL Server is running. The time zone offset is not included.	datetime2(7)	Nondeterministic
SYSDATETIMEOFFSET	SYSDATETIMEOFFSET ( )	Returns a datetimeoffset(7) value that contains the date and time of the computer on which the instance of SQL Server is running. The time zone offset is	datetimeoffset(7)	Nondeterministic

		included.		
SYSUTCDATETIME	SYSUTCDATETIME ( )	Returns a datetime2(7) value that contains the date and time of the computer on which the instance of SQL Server is running. The date and time is returned as UTC time (Coordinated Universal Time).	datetime2(7)	Nondeterministic

Lower-Precision System Date and Time Functions

Function	Syntax	Return value	Return data type	Determinism
CURRENT_TIMESTAMP	CURRENT_TIMESTAMP	Returns a datetime2(7) value that contains the date and time of the computer on which the instance of SQL Server is running. The time zone offset is not included.	datetime	Nondeterministic
GETDATE	GETDATE ( )	Returns a datetime2(7) value that contains the date and time of the computer on which the instance of SQL Server is running. The time zone offset is not included.	datetime	Nondeterministic
GETUTCDATE	GETUTCDATE ( )	Returns a datetime2(7) value that contains the date and time of the	datetime	Nondeterministic

		computer on which the instance of SQL Server is running. The date and time is returned as UTC time (Coordinated Universal Time).		

Functions That Get Date and Time Parts

Function	Syntax	Return value	Return data type	Determinism
DATENAME	DATENAME (*datepart* , *date*)	Returns a character string that represents the specified *datepart* of the specified date.	nvarchar	Nondeterministic
DATEPART	DATEPART (*datepart* , *date*)	Returns an integer that represents the specified *datepart* of the specified *date*.	int	Nondeterministic
DAY	DAY (*date*)	Returns an integer that represents the day day part of the specified *date*.	int	Deterministic
MONTH	MONTH (*date*)	Returns an integer that represents the month part of a specified *date*.	int	Deterministic
YEAR	YEAR (*date*)	Returns an integer that represents the year part of a specified *date*.	int	Deterministic

Functions That Get Date and Time Difference

Function	Syntax	Return value	Return data type	Determinism
DATEDIFF	DATEDIFF (*datepart* , *startdate* , *enddate*)	Returns the number of date or time *datepart* boundaries that are crossed between two specified dates.	int	Deterministic

Functions That Modify Date and Time Values

Function	Syntax	Return value	Return data type	Determinism
DATEADD	DATEADD (*datepart* , *number* , *date*)	Returns a new datetime value by adding an interval to the specified *datepart* of the	The data type of the *date* argument	Deterministic

		specified *date*.		
SWITCHOFFSET	SWITCH*OFFSET* (*DATETIMEOFFSET* , *time_zone*)	SWITCH*OFFSET* changes the time zone offset of a DATETIMEOFFSET value and preserves the UTC value.	datetimeoffset with the fractional precision of the *DATETIMEOFFSET*	Deterministic
TODATETIMEOFFSET	TODATETIMEOFFSET (*expression* , *time_zone*)	TODATETIMEOFFSET transforms a datetime2 value into a datetimeoffset value. The datetime2 value is interpreted in local time for the specified time_zone.	datetimeoffset with the fractional precision of the *datetime* argument	Deterministic

Functions That Set or Get Session Format

Function	Syntax	Return value	Return data type	Determinism
@@DATEFIRST	@@DATEFIRST	Returns the current value, for the session, of SET DATEFIRST.	tinyint	Nondeterministic
SET DATEFIRST	SET DATEFIRST { *number* | **@***number_var* }	Sets the first day of the week to a number from 1 through 7.	Not applicable	Not applicable
SET DATEFORMAT	SET DATEFORMAT { *format* | **@***format_var* }	Sets the order of the dateparts (month/day/year) for entering datetime or smalldatetime data.	Not applicable	Not applicable
@@LANGUAGE	@@LANGUAGE	Returns the name of the language that is currently being used. @@LANGUAGE is not a date or time function.	Not applicable	Not applicable

		However, the language setting can affect the output of date functions.		
SET LANGUAGE	SET LANGUAGE { [N] '*language*' | **@***language_var* }	Sets the language environment for the session and system messages. SET LANGUAGE is not a date or time function. However, the language setting affects the output of date functions.	Not applicable	Not applicable
sp_helplanguage	**sp_helplanguage** [[**@language** =] '*language*']	Returns information about date formats of all supported languages. **sp_helplanguage** is not a date or time stored procedure. However, the language setting affects the output of date functions.	Not applicable	Not applicable

Functions That Validate Date and Time Values

Function	Syntax	Return value	Return data type	Determinism
ISDATE	ISDATE (*expression*)	Determines whether a datetime or smalldatetime input expression is a valid date or time value.	int	ISDATE is deterministic only if you use it with the CONVERT function, when the CONVERT style parameter is specified, and when style is not equal to 0, 100, 9, or 109.

Date and Time–Related Topics

Topic	Description
Using Date and Time Data	Provides information and examples that are common to date and time data types and functions.
CAST and **CONVERT**	Provides information about the conversion of date and time values to and from

	string literals and other date and time formats.
Writing International Transact-SQL Statements	Provides guidelines for portability of databases and database applications that use Transact-SQL statements from one language to another, or that support multiple languages.
ODBC Scalar Functions	Provides information about ODBC scalar functions that can be used in Transact-SQL statements. This includes ODBC date and time functions.
Data Type Mapping with Distributed Queries	Provides information about how date and time data types affect distributed queries between servers that have different versions of SQL Server or different providers.

3.6.1. DATEADD

Returns a specified *date* with the specified *number* interval (signed integer) added to a specified *datepart* of that *date*.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
DATEADD (datepart , number , date )
```

Arguments

datepart

Is the part of *date* to which an integer *number* is added. The following table lists all valid *datepart* arguments. User-defined variable equivalents are not valid.

datepart	Abbreviations
`year`	`yy, yyyy`
`quarter`	`qq, q`
`month`	`mm, m`
`dayofyear`	`dy, y`
`day`	`dd, d`

week	wk, ww
weekday	dw, w
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns

number

Is an expression that can be resolved to an int that is added to a *datepart* of *date*. User-defined variables are valid.

If you specify a value with a decimal fraction, the fraction is truncated and not rounded.

date

Is an expression that can be resolved to a time, date, smalldatetime, datetime, datetime2, or datetimeoffset value. *date* can be an expression, column expression, user-defined variable, or string literal. If the expression is a string literal, it must resolve to a datetime. To avoid ambiguity, use four-digit years. For information about two-digit years, see two digit year cutoff Option.

Return Types

The return data type is the data type of the *date* argument, except for string literals.

The return data type for a string literal is datetime. An error will be raised if the string literal seconds scale is more than three positions (.nnn) or contains the time zone offset part.

Note

If string literals are not explicitly cast for the *date* parameter then locals that use a day-month-year (dmy) date format may get incorrect results when DATEADD is used in conjunction with other date/time functions.

Returning a datetime2 type

DATEADD returns a datetime2 type when the *date* parameter is a datetime2 type. When using string literals for the *date* parameter, then you must explicitly cast them to a datetime2 type for DATEADD to return a datetime2 type.

datepart Argument

dayofyear, day, and weekday return the same value.

Each *datepart* and its abbreviations return the same value.

If *datepart* is month and the *date* month has more days than the return month and the *date* day does not exist in the return month, the last day of the return month is returned. For example, September has 30 days; therefore, the two following statements return 2006-09-30 00:00:00.000:

```
SELECT DATEADD(month, 1, '2006-08-30')
SELECT DATEADD(month, 1, '2006-08-31')
```

The *number* argument cannot exceed the range of int. In the following statements, the argument for *number* exceeds the range of int by 1. The following error message is returned: "Arithmetic overflow error converting expression to data type int."

```
SELECT DATEADD(year,2147483648, '2006-07-31');
SELECT DATEADD(year,-2147483649, '2006-07-31');
```

The *date* argument cannot be incremented to a value outside the range of its data type. In the following statements, the *number* value that is added to the *date* value exceeds the range of the *date* data type. The following error message is returned: "Adding a value to a 'datetime' column caused overflow."

```
SELECT DATEADD(year,2147483647, '2006-07-31');
SELECT DATEADD(year,-2147483647, '2006-07-31');
```

The seconds part of a smalldatetime value is always 00. If *date* is smalldatetime, the following apply:

- If *datepart* is second and *number* is between -30 and +29, no addition is performed.
- If *datepart* is second and *number* is less than-30 or more than +29, addition is performed beginning at one minute.
- If *datepart* is millisecond and *number* is between -30001 and +29998, no addition is performed.
- If *datepart* is millisecond and *number* is less than -30001 or more than +29998, addition is performed beginning at one minute.

Remarks

DATEADD can be used in the SELECT <list>, WHERE, HAVING, GROUP BY and ORDER BY clauses.

Fractional Seconds Precision

Addition for a *datepart* of microsecond or nanosecond for *date* data types smalldatetime, date, and datetime is not allowed.

Milliseconds have a scale of 3 (.123). microseconds have a scale of 6 (.123456). nanoseconds have a scale of 9 (.123456789). The time, datetime2, and datetimeoffset data types have a maximum scale of 7 (.1234567). If *datepart* is nanosecond, *number* must be 100 before the fractional seconds of *date* increase. A *number* between 1 and 49 is rounded down to 0 and a number from 50 to 99 is rounded up to 100.

The following statements add a *datepart* of millisecond, microsecond, or nanosecond.

```
DECLARE @datetime2 datetime2 = '2007-01-01 13:10:10.1111111'
SELECT '1 millisecond' ,DATEADD(millisecond,1,@datetime2)
```

```
UNION ALL
SELECT '2 milliseconds', DATEADD(millisecond,2,@datetime2)
UNION ALL
SELECT '1 microsecond', DATEADD(microsecond,1,@datetime2)
UNION ALL
SELECT '2 microseconds', DATEADD(microsecond,2,@datetime2)
UNION ALL
SELECT '49 nanoseconds', DATEADD(nanosecond,49,@datetime2)
UNION ALL
SELECT '50 nanoseconds', DATEADD(nanosecond,50,@datetime2)
UNION ALL
SELECT '150 nanoseconds', DATEADD(nanosecond,150,@datetime2);
/*
Returns:
1 millisecond     2007-01-01 13:10:10.1121111
2 milliseconds    2007-01-01 13:10:10.1131111
1 microsecond     2007-01-01 13:10:10.1111121
2 microseconds    2007-01-01 13:10:10.1111131
49 nanoseconds    2007-01-01 13:10:10.1111111
50 nanoseconds    2007-01-01 13:10:10.1111112
150 nanoseconds   2007-01-01 13:10:10.1111113
*/
```

Time Zone Offset

Addition is not allowed for time zone offset.

Examples

A. Incrementing datepart by an interval of 1

Each of the following statements increments *datepart* by an interval of 1.

```
DECLARE @datetime2 datetime2 = '2007-01-01 13:10:10.1111111'
SELECT 'year', DATEADD(year,1,@datetime2)
UNION ALL
SELECT 'quarter',DATEADD(quarter,1,@datetime2)
UNION ALL
SELECT 'month',DATEADD(month,1,@datetime2)
UNION ALL
SELECT 'dayofyear',DATEADD(dayofyear,1,@datetime2)
UNION ALL
SELECT 'day',DATEADD(day,1,@datetime2)
UNION ALL
SELECT 'week',DATEADD(week,1,@datetime2)
UNION ALL
SELECT 'weekday',DATEADD(weekday,1,@datetime2)
UNION ALL
SELECT 'hour',DATEADD(hour,1,@datetime2)
UNION ALL
SELECT 'minute',DATEADD(minute,1,@datetime2)
UNION ALL
SELECT 'second',DATEADD(second,1,@datetime2)
UNION ALL
```

```
SELECT 'millisecond',DATEADD(millisecond,1,@datetime2)
UNION ALL
SELECT 'microsecond',DATEADD(microsecond,1,@datetime2)
UNION ALL
SELECT 'nanosecond',DATEADD(nanosecond,1,@datetime2);
/*
Year          2008-01-01 13:10:10.1111111
quarter       2007-04-01 13:10:10.1111111
month         2007-02-01 13:10:10.1111111
dayofyear     2007-01-02 13:10:10.1111111
day           2007-01-02 13:10:10.1111111
week          2007-01-08 13:10:10.1111111
weekday       2007-01-02 13:10:10.1111111
hour          2007-01-01 14:10:10.1111111
minute        2007-01-01 13:11:10.1111111
second        2007-01-01 13:10:11.1111111
millisecond   2007-01-01 13:10:10.1121111
microsecond   2007-01-01 13:10:10.1111121
nanosecond    2007-01-01 13:10:10.1111111
*/
```

B. Incrementing more than one level of datepart in one statement

Each of the following statements increments *datepart* by a *number* large enough to also increment the next higher *datepart* of *date*.

```
DECLARE @datetime2 datetime2;
SET @datetime2 = '2007-01-01 01:01:01.1111111';
--Statement                                    Result
-----------------------------------------------------------------
SELECT DATEADD(quarter,4,@datetime2);     --2008-01-01 01:01:01.110
SELECT DATEADD(month,13,@datetime2);      --2008-02-01 01:01:01.110
SELECT DATEADD(dayofyear,365,@datetime2); --2008-01-01 01:01:01.110
SELECT DATEADD(day,365,@datetime2);       --2008-01-01 01:01:01.110
SELECT DATEADD(week,5,@datetime2);        --2007-02-05 01:01:01.110
SELECT DATEADD(weekday,31,@datetime2);    --2007-02-01 01:01:01.110
SELECT DATEADD(hour,23,@datetime2);       --2007-01-02 00:01:01.110
SELECT DATEADD(minute,59,@datetime2);     --2007-01-01 02:00:01.110
SELECT DATEADD(second,59,@datetime2);     --2007-01-01 01:02:00.110
SELECT DATEADD(millisecond,1,@datetime2); --2007-01-01 01:01:01.110
```

C. Using expressions as arguments for the number and date parameters

The following examples use different types of expressions as arguments for the *number* and *date* parameters.

Specifying column as date

The following example adds 2 days to each OrderDate to calculate a new PromisedShipDate.

```
USE AdventureWorks2008R2;
```

```
GO
SELECT SalesOrderID
    ,OrderDate
    ,DATEADD(day,2,OrderDate) AS PromisedShipDate
FROM Sales.SalesOrderHeader;
```

Specifying user-defined variables as number and date

The following example specifies user-defined variables as arguments for *number* and *date*.

```
DECLARE @days int;
DECLARE @datetime datetime;
SET @days = 365;
SET @datetime = '2000-01-01 01:01:01.111'; /* 2000 was a leap year */
SELECT DATEADD(day, @days, @datetime);
```

Specifying scalar system function as date

The following example specifies SYSDATETIME for *date*.

```
SELECT DATEADD(month, 1, SYSDATETIME());
```

Specifying scalar subqueries and scalar functions as number and date

The following example uses scalar subqueries and scalar functions, MAX(ModifiedDate), as arguments for *number* and *date*. (SELECT TOP 1 BusinessEntityID FROM Person.Person) is an artificial argument for the number parameter to show how to select a *number* argument from a value list.

```
USE AdventureWorks2008R2;
GO
SELECT DATEADD(month,(SELECT TOP 1 BusinessEntityID FROM Person.Person),
    (SELECT MAX(ModifiedDate) FROM Person.Person));
```

Specifying constants as number and date

The following example uses numeric and character constants as arguments for *number* and *date*.

```
SELECT DATEADD(minute, 1, ' 2007-05-07 09:53:01.0376635');
```

Specifying numeric expressions and scalar system functions as number and date

The following example uses a numeric expressions (-(10/2)), unary operators (-), an arithmetic operator (/), and scalar system functions (SYSDATETIME) as arguments for *number* and *date*.

```
SELECT DATEADD(month,-(10/2), SYSDATETIME());
```

Specifying ranking functions as number

The following example uses a ranking function as arguments for *number*.

```
USE AdventureWorks2008R2;
GO
SELECT p.FirstName, p.LastName
    ,DATEADD(day,ROW_NUMBER() OVER (ORDER BY
        a.PostalCode),SYSDATETIME()) AS 'Row Number'
```

```
FROM Sales.SalesPerson AS s
    INNER JOIN Person.Person AS p
        ON s.BusinessEntityID = p.BusinessEntityID
    INNER JOIN Person.Address AS a
        ON a.AddressID = p.BusinessEntityID
WHERE TerritoryID IS NOT NULL
    AND SalesYTD <> 0;
```

Specifying an aggregate window function as number

The following example uses an aggregate window function as an argument for *number*.

```
USE AdventureWorks2008R2;
GO
SELECT SalesOrderID, ProductID, OrderQty
    ,DATEADD(day,SUM(OrderQty)
        OVER(PARTITION BY SalesOrderID),SYSDATETIME()) AS 'Total'
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN(43659,43664);
GO
```

D. Using DATEADD for Locales that use the dmy date format

The following examples show how to use string literals with DATEADD for some locales.

Demonstrating the pitfalls of using an implicit cast of a string literal

The following example shows what happens when a string literal is not explicitly cast.

```
SET LANGUAGE Español;
GO
SELECT DATENAME(m, DATEADD(d, 0,'1987-03-07'));
SELECT DATENAME(m, '1987-03-07');
GO
```

The first select statement returns julio (July) for the month and the second select statement return marzo (March) for the month.

Avoiding erroneous results by explicitly casting the string literal

The following example shows how to explicitly cast the *date* parameter to avoid erroneous results.

```
SET LANGUAGE Español;
GO
SELECT DATENAME(m, DATEADD(d, 0, CAST('1987-03-07' AS datetime2)));
SELECT DATENAME(m, '1987-03-07');
GO
```

Both select statements return marzo (March) for the month.

Using a datetime2 variable in the place of a string literal

The following example avoids the direct use of a string literal.

```
SET LANGUAGE Español;
GO
DECLARE @d datetime2 = '1987-03-07';
SELECT DATENAME(m, DATEADD(d, 0, @d));
SELECT DATENAME(m, @d);
GO
```

3.6.2. DATEDIFF

Returns the count (signed integer) of the specified *datepart* boundaries crossed between the specified *startdate* and *enddate*.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
DATEDIFF ( datepart , startdate , enddate )
```

Arguments

datepart

Is the part of *startdate* and *enddate* that specifies the type of boundary crossed. The following table lists all valid *datepart* arguments. User-defined variable equivalents are not valid.

datepart	Abbreviations
`year`	`yy, yyyy`
`quarter`	`qq, q`
`month`	`mm, m`
`dayofyear`	`dy, y`
`day`	`dd, d`
`week`	`wk, ww`
`hour`	`hh`
`minute`	`mi, n`
`second`	`ss, s`
`millisecond`	`ms`
`microsecond`	`mcs`

nanosecond	**ns**

startdate

Is an expression that can be resolved to a time, date, smalldatetime, datetime, datetime2, or datetimeoffset value. *date* can be an expression, column expression, user-defined variable or string literal. *startdate* is subtracted from *enddate*.

To avoid ambiguity, use four-digit years. For information about two digits years, see two digit year cutoff Option.

enddate

See *startdate*.

Return Type

int

Return Value

- Each *datepart* and its abbreviations return the same value.

If the return value is out of range for int (-2,147,483,648 to +2,147,483,647), an error is returned. For millisecond, the maximum difference between *startdate* and *enddate* is 24 days, 20 hours, 31 minutes and 23.647 seconds. For second, the maximum difference is 68 years.

If *startdate* and *enddate* are both assigned only a time value and the *datepart* is not a time *datepart*, 0 is returned.

A time zone offset component of *startdate* or *endate* is not used in calculating the return value.

Because smalldatetime is accurate only to the minute, when a smalldatetime value is used for *startdate* or *enddate*, seconds and milliseconds are always set to 0 in the return value.

If only a time value is assigned to a variable of a date data type, the value of the missing date part is set to the default value: 1900-01-01. If only a date value is assigned to a variable of a time or date data type, the value of the missing time part is set to the default value: 00:00:00. If either *startdate* or *enddate* have only a time part and the other only a date part, the missing time and date parts are set to the default values.

If *startdate* and *enddate* are of different date data types and one has more time parts or fractional seconds precision than the other, the missing parts of the other are set to 0.

datepart Boundaries

The following statements have the same *startdate* and the same *endate*. Those dates are adjacent and differ in time by .0000001 second. The difference between the *startdate* and *endate* in each statement crosses one calendar or time boundary of its *datepart*. Each statement returns 1. If different years are used for this example and if both *startdate* and *endate* are in the same calendar week, the return value for week would be 0.

```
SELECT DATEDIFF(year, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(quarter, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(month, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(dayofyear, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(day, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(week, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(hour, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(minute, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(second, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
SELECT DATEDIFF(millisecond, '2005-12-31 23:59:59.9999999'
, '2006-01-01 00:00:00.0000000');
```

Remarks

DATEDIFF can be used in the select list, WHERE, HAVING, GROUP BY and ORDER BY clauses. In SQL Server 2008, DATEDIFF implicitly casts string literals as datetime2 types. When using DATEDIFF with DATEADD, avoid implicit casts of string literals. For more information, see DATEADD

Examples

The following examples use different types of expressions as arguments for the *startdate* and *enddate* parameters.

A. Specifying columns for startdate and enddate

The following example calculates the number of day boundaries that are crossed between dates in two columns in a table.

```
CREATE TABLE dbo.Duration
    (
    startDate datetime2
    ,endDate datetime2
    )
INSERT INTO dbo.Duration(startDate,endDate)
    VALUES('2007-05-06 12:10:09','2007-05-07 12:10:09')
SELECT DATEDIFF(day,startDate,endDate) AS 'Duration'
FROM dbo.Duration;
-- Returns: 1
```

B. Specifying user-defined variables for startdate and enddate

The following example uses user-defined variables as arguments for *startdate* and *enddate*.

```
DECLARE @startdate datetime2 = '2007-05-05 12:10:09.3312722';
DECLARE @enddate datetime2 = '2007-05-04 12:10:09.3312722';
SELECT DATEDIFF(day, @startdate, @enddate);
```

C. Specifying scalar system functions for startdate and enddate

The following example uses scalar system functions as arguments for *startdate* and *enddate*.

```
SELECT DATEDIFF(millisecond, GETDATE(), SYSDATETIME());
```

D. Specifying scalar subqueries and scalar functions for startdate and enddate

The following example uses scalar subqueries and scalar functions as arguments for *startdate* and *enddate*.

```
USE AdventureWorks2008R2;
GO
SELECT DATEDIFF(day,(SELECT MIN(OrderDate) FROM Sales.SalesOrderHeader),
    (SELECT MAX(OrderDate) FROM Sales.SalesOrderHeader));
```

E. Specifying constants for startdate and enddate

The following example uses character constants as arguments for *startdate* and *enddate*.

```
SELECT DATEDIFF(day, '2007-05-07 09:53:01.0376635'
    , '2007-05-08 09:53:01.0376635');
```

F. Specifying numeric expressions and scalar system functions for enddate

The following example uses a numeric expression, (GETDATE ()+ 1), and scalar system functions, GETDATE and SYSDATETIME, as arguments for *enddate*.

Note

SYSDATETIME, SYSUTCDATETIME, and SYSDATETIMEOFFSET cannot be part of an arithmetic expression.

```
USE AdventureWorks2008R2;
GO
SELECT DATEDIFF(day, '2007-05-07 09:53:01.0376635', GETDATE()+ 1)
    AS NumberOfDays
FROM Sales.SalesOrderHeader;
GO
USE AdventureWorks2008R2;
GO
SELECT DATEDIFF(day, '2007-05-07 09:53:01.0376635',
DATEADD(day,1,SYSDATETIME())) AS NumberOfDays
FROM Sales.SalesOrderHeader;
GO
```

G. Specifying ranking functions for startdate

The following example uses a ranking function as an argument for *startdate*.

```
USE AdventureWorks2008R2;
GO
SELECT p.FirstName, p.LastName
    ,DATEDIFF(day,ROW_NUMBER() OVER (ORDER BY
        a.PostalCode),SYSDATETIME()) AS 'Row Number'
FROM Sales.SalesPerson s
    INNER JOIN Person.Person p
        ON s.BusinessEntityID = p.BusinessEntityID
    INNER JOIN Person.Address a
        ON a.AddressID = p.BusinessEntityID
WHERE TerritoryID IS NOT NULL
    AND SalesYTD <> 0;
```

H. Specifying an aggregate window function for startdate

The following example uses an aggregate window function as an argument for *startdate*.

```
USE AdventureWorks2008R2;
GO
SELECT soh.SalesOrderID, sod.ProductID, sod.OrderQty,soh.OrderDate
    ,DATEDIFF(day,MIN(soh.OrderDate)
        OVER(PARTITION BY soh.SalesOrderID),SYSDATETIME() ) AS 'Total'
FROM Sales.SalesOrderDetail sod
    INNER JOIN Sales.SalesOrderHeader soh
        ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.SalesOrderID IN(43659,58918);
GO
```

3.6.3. DATENAME

Returns a character string that represents the specified *datepart* of the specified *date*

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
DATENAME ( datepart , date )
```

Arguments

datepart

Is the part of the *date* to return. The following table lists all valid *datepart* arguments. User-defined variable equivalents are not valid.

datepart	Abbreviations
`year`	`yy, yyyy`
`quarter`	`qq, q`
`month`	`mm, m`
`dayofyear`	`dy, y`
`day`	`dd, d`
`week`	`wk, ww`
`weekday`	`dw`
`hour`	`hh`
`minute`	`mi, n`
`second`	`ss, s`
`millisecond`	`ms`
`microsecond`	`mcs`
`nanosecond`	`ns`
`TZoffset`	`tz`

date

Is an expression that can be resolved to a time, date, smalldatetime, datetime, datetime2, or datetimeoffset value. *date* can be an expression, column expression, user-defined variable, or string literal.

To avoid ambiguity, use four-digit years. For information about two-digit years, see two digit year cutoff Option.

Return Type

nvarchar

Return Value

- Each *datepart* and its abbreviations return the same value.

The return value depends on the language environment set by using SET LANGUAGE and by the default language of the login. The return value is dependant on SET DATEFORMAT if *date* is a string literal of some formats. SET DATEFORMAT does not affect the return value when the date is a column expression of a date or time data type.

For versions of SQL Server later than SQL Server 2000, when the *date* parameter has a date data type argument, the return value depends on the setting specified by using SET DATEFIRST.

TZoffset datepart Argument

If *datepart* argument is TZoffset (tz) and the *date* argument has no time zone offset, 0 is returned.

smalldatetime date Argument

When *date* is smalldatetime, seconds are returned as 00.

Default Returned for a datepart That Is Not in the date Argument

If the data type of the *date* argument does not have the specified *datepart*, the default for that *datepart* will be returned.

For example, the default year-month-day for any date data type is 1900-01-01. The following statement has date part arguments for *datepart*, a time argument for *date*, and returns 1900, January, 1, 1, Monday.

```
SELECT DATENAME(year, '12:10:30.123')
    ,DATENAME(month, '12:10:30.123')
    ,DATENAME(day, '12:10:30.123')
    ,DATENAME(dayofyear, '12:10:30.123')
    ,DATENAME(weekday, '12:10:30.123');
```

The default hour-minute-second for the time data type is 00:00:00. The following statement has time part arguments for *datepart*, a date argument for *date*, and returns 0, 0, 0.

```
SELECT DATENAME(hour, '2007-06-01')
    ,DATENAME(minute, '2007-06-01')
    ,DATENAME(second, '2007-06-01');
```

Remarks

DATENAME can be used in the select list, WHERE, HAVING, GROUP BY, and ORDER BY clauses. In SQL Server 2008, DATENAME implicitly casts string literals as datetime2 types. When using DATENAME with DATEADD, avoid implicit casts of string literals. For more information, see DATEADD

Examples

The following example returns the date parts for the specified date.

```
SELECT DATENAME(datepart,'2007-10-30 12:15:32.1234567 +05:10')
```

Here is the result set.

datepart	Return value
year, yyyy, yy	2007
quarter, qq, q	4
month, mm, m	October
dayofyear, dy, y	303
day, dd, d	30
week, wk, ww	44
weekday, dw	Tuesday
hour, hh	12
minute, n	15
second, ss, s	32
millisecond, ms	123
microsecond, mcs	123456
nanosecond, ns	123456700
TZoffset, tz	310

3.6.4. DATEPART

Returns an integer that represents the specified *datepart* of the specified *date*.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
DATEPART ( datepart , date )
```

Arguments

datepart

Is the part of *date* (a date or time value) for which an integer will be returned. The following table lists all valid *datepart* arguments. User-defined variable equivalents are not valid.

datepart	Abbreviations
`year`	`yy, yyyy`
`quarter`	`qq, q`
`month`	`mm, m`
`dayofyear`	`dy, y`
`day`	`dd, d`
`week`	`wk, ww`
`weekday`	`dw`
`hour`	`hh`
`minute`	`mi, n`
`second`	`ss, s`
`millisecond`	`ms`
`microsecond`	`mcs`
`nanosecond`	`ns`
`TZoffset`	`tz`
`ISO_WEEK`	`isowk, isoww`

date

Is an expression that can be resolved to a time, date, smalldatetime, datetime, datetime2, or datetimeoffset value. *date* can be an expression, column expression, user-defined variable, or string literal.

To avoid ambiguity, use four-digit years. For information about two digits years, see two digit year cutoff Option.

Return Type

int

Return Value

Each *datepart* and its abbreviations return the same value.

The return value depends on the language environment set by using SET LANGUAGE and by the default language of the login. If *date* is a string literal for some formats, the return value depends on the format specified by using SET DATEFORMAT. SET DATEFORMAT does not affect the return value when the date is a column expression of a date or time data type.

The following table lists all *datepart* arguments with corresponding return values for the statement SELECT DATEPART(datepart,'2007-10-30 12:15:32.1234567 +05:10'). The data type of the *date* argument is datetimeoffset(7). The nanosecond *datepart* return value has a scale of 9 (.123456700) and the last two positions are always 00.

datepart	Return value
`year, yyyy, yy`	2007
`quarter, qq, q`	4
`month, mm, m`	10
`dayofyear, dy, y`	303
`day, dd, d`	30
`week, wk, ww`	45
`weekday, dw`	1
`hour, hh`	12
`minute, n`	15
`second, ss, s`	32
`millisecond, ms`	123
`microsecond, mcs`	123456
`nanosecond, ns`	123456700
`TZoffset, tz`	310

week and weekday datepart Arguments

When *datepart* is week (wk, ww) or weekday (dw), the return value depends on the value that is set by using SET DATEFIRST.

January 1 of any year defines the starting number for the week *datepart*, for example: DATEPART (wk, 'Jan 1, *xxxx*') = 1, where *xxxx* is any year.

The following table lists the return value for week and weekday *datepart* for '2007-04-21 ' for each SET DATEFIRST argument. January 1 is a Sunday in the year 2007. April 21 is a Saturday in the year 2007. SET DATEFIRST 7, Sunday, is the default for U.S. English.

SET DATEFIRST argument	week returned	weekday returned
1	16	6
2	17	5
3	17	4
4	17	3
5	17	2
6	17	1
7	16	7

year, month, and day datepart Arguments

The values that are returned for DATEPART (year, *date*), DATEPART (month, *date*), and DATEPART (day, *date*) are the same as those returned by the functions YEAR, MONTH, and DAY, f respectively.

ISO_WEEK datepart

ISO 8601 includes the ISO week-date system, a numbering system for weeks. Each week is associated with the year in which Thursday occurs. For example, week 1 of 2004 (2004W01) ran from Monday 29 December 2003 to Sunday, 4 January 2004. The highest week number in a year might be 52 or 53. This style of numbering is typically used in European countries/regions, but rare elsewhere.

The numbering system in different countries/regions might not comply with the ISO standard. There are at least six possibilities as shown in the following table

First day of week	First week of year contains	Weeks assigned two times	Used by/in

Sunday	1 January, First Saturday, 1–7 days of year	Yes	United States
Monday	1 January, First Sunday, 1–7 days of year	Yes	Most of Europe and the United Kingdom
Monday	4 January, First Thursday, 4–7 days of year	No	ISO 8601, Norway, and Sweden
Monday	7 January, First Monday, 7 days of year	No	
Wednesday	1 January, First Tuesday, 1–7 days of year	Yes	
Saturday	1 January, First Friday, 1–7 days of year	Yes	

TZoffset

The TZoffset (tz) is returned as the number of minutes (signed). The following statement returns a time zone offset of 310 minutes.

```
SELECT DATEPART (TZoffset, 2007-05-10  00:00:01.1234567 +05:10);
```

If the *datepart* argument is TZoffset (tz) and the *date* argument is not of datetimeoffset data type, NULL is returned.

smalldatetime date Argument

When *date* is smalldatetime, seconds are returned as 00.

Default Returned for a datepart That Is Not in a date Argument

If the data type of the *date* argument does not have the specified *datepart*, the default for that *datepart* will be returned.

For example, the default year-month-day for any date data type is 1900-01-01. The following statement has date part arguments for *datepart*, a time argument for *date*, and returns 1900, 1, 1, 1, 2.

```
SELECT DATEPART(year, '12:10:30.123')
    ,DATEPART(month, '12:10:30.123')
    ,DATEPART(day, '12:10:30.123')
    ,DATEPART(dayofyear, '12:10:30.123')
    ,DATEPART(weekday, '12:10:30.123');
```

The default hour-minute-second for the time data type is 00:00:00. The following statement has time part arguments for *datepart*, a date argument for *date*, and returns 0, 0, 0.

```
SELECT DATEPART(hour, '2007-06-01')
    ,DATEPART(minute, '2007-06-01')
    ,DATEPART(second, '2007-06-01');
```

Fractional Seconds

Fractional seconds are returned as shown in the following statements:

```
SELECT DATEPART(millisecond, '00:00:01.1234567'); -- Returns 123
SELECT DATEPART(microsecond, '00:00:01.1234567'); -- Returns 123456
SELECT DATEPART(nanosecond,  '00:00:01.1234567'); -- Returns 123456700
```

Remarks

DATEPART can be used in the select list, WHERE, HAVING, GROUP BY and ORDER BY clauses. In SQL Server 2008, DATEPART implicitly casts string literals as datetime2 types. When using DATEPART with DATEADD, avoid implicit casts of string literals. For more information, see DATEADD

Examples

The following example returns the base year. The base year is useful for date calculations. In the example, the date is specified as a number. Notice that SQL Server interprets 0 as January 1, 1900.

```
SELECT DATEPART(year, 0), DATEPART(month, 0), DATEPART(day, 0);
-- Returns: 1900    1    1 */
```

3.6.5. DAY

Returns an integer representing the day (day of the month) of the specified *date*.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
DAY ( date )
```

Arguments

date

Is an expression that can be resolved to a time, date, smalldatetime, datetime, datetime2, or datetimeoffset value. The *date* argument can be an expression, column expression, user-defined variable or string literal.

Return Type

int

Return Value

DAY returns the same value as DATEPART (day, *date*).

If *date* contains only a time part, the return value is 1, the base day.

Examples

The following statement returns 30. This is the number of the day.

```
SELECT DAY('2007-04-30T01:01:01.1234567 -07:00');
```

The following statement returns 1900, 1, 1. The argument for *date* is the number 0. SQL Server interprets 0 as January 1, 1900.

```
SELECT YEAR(0), MONTH(0), DAY(0);
```

3.6.6. GETDATE

Returns the current database system timestamp as a datetime value without the database time zone offset. This value is derived from the operating system of the computer on which the instance of SQL Server is running.

Note

SYSDATETIME and SYSUTCDATETIME have more fractional seconds precision than GETDATE and GETUTCDATE. SYSDATETIMEOFFSET includes the system time zone offset. SYSDATETIME, SYSUTCDATETIME, and SYSDATETIMEOFFSET can be assigned to a variable of any of the date and time types.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
GETDATE ( )
```

Return Type

datetime

Remarks

Transact-SQL statements can refer to GETDATE anywhere they can refer to a datetime expression.

GETDATE is a nondeterministic function. Views and expressions that reference this function in a column cannot be indexed.

Examples

The following examples use the six SQL Server system functions that return current date and time to return the date, time, or both. The values are returned in series; therefore, their fractional seconds might be different.

A. Getting the current system date and time

```
SELECT SYSDATETIME()
    ,SYSDATETIMEOFFSET()
    ,SYSUTCDATETIME()
    ,CURRENT_TIMESTAMP
    ,GETDATE()
    ,GETUTCDATE();
```

Here is the result set.

SYSDATETIME() 2007-04-30 13:10:02.0474381
SYSDATETIMEOFFSET()2007-04-30 13:10:02.0474381 -07:00
SYSUTCDATETIME() 2007-04-30 20:10:02.0474381
CURRENT_TIMESTAMP 2007-04-30 13:10:02.047
GETDATE() 2007-04-30 13:10:02.047
GETUTCDATE() 2007-04-30 20:10:02.047

B. Getting the current system date

```
SELECT CONVERT (date, SYSDATETIME())
    ,CONVERT (date, SYSDATETIMEOFFSET())
    ,CONVERT (date, SYSUTCDATETIME())
    ,CONVERT (date, CURRENT_TIMESTAMP)
    ,CONVERT (date, GETDATE())
    ,CONVERT (date, GETUTCDATE());
```

Here is the result set.

SYSDATETIME() 2007-05-03
SYSDATETIMEOFFSET() 2007-05-03
SYSUTCDATETIME() 2007-05-04
CURRENT_TIMESTAMP 2007-05-03
GETDATE() 2007-05-03

GETUTCDATE() 2007-05-04

C. Getting the current system time

```
SELECT CONVERT (time, SYSDATETIME())
    ,CONVERT (time, SYSDATETIMEOFFSET())
    ,CONVERT (time, SYSUTCDATETIME())
    ,CONVERT (time, CURRENT_TIMESTAMP)
    ,CONVERT (time, GETDATE())
    ,CONVERT (time, GETUTCDATE());
```

Here is the result set.

SYSDATETIME() 13:18:45.3490361
SYSDATETIMEOFFSET()13:18:45.3490361
SYSUTCDATETIME() 20:18:45.3490361
CURRENT_TIMESTAMP 13:18:45.3470000
GETDATE() 13:18:45.3470000
GETUTCDATE() 20:18:45.3470000

3.6.7. GETUTCDATE

Returns the current database system timestamp as a datetime value. The database time zone offset is not included. This value represents the current UTC time (Coordinated Universal Time). This value is derived from the operating system of the computer on which the instance of SQL Server is running.

Note
SYSDATETIME and SYSUTCDATETIME have more fractional seconds precision than GETDATE and GETUTCDATE. SYSDATETIMEOFFSET includes the system time zone offset. SYSDATETIME, SYSUTCDATETIME, and SYSDATETIMEOFFSET can be assigned to a variable of any of the date and time types.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
GETUTCDATE()
```

Return Types

datetime

Remarks

Transact-SQL statements can refer to GETUTCDATE anywhere they can refer to a datetime expression.

GETUTCDATE is a nondeterministic function. Views and expressions that reference this function in a column cannot be indexed.

Examples

The following examples use the six SQL Server system functions that return current date and time to return the date, time or both. The values are returned in series; therefore, their fractional seconds might be different.

A. Getting the current system date and time

```
SELECT 'SYSDATETIME()      ', SYSDATETIME();
SELECT 'SYSDATETIMEOFFSET()', SYSDATETIMEOFFSET();
SELECT 'SYSUTCDATETIME()   ', SYSUTCDATETIME();
SELECT 'CURRENT_TIMESTAMP  ', CURRENT_TIMESTAMP;
SELECT 'GETDATE()          ', GETDATE();
SELECT 'GETUTCDATE()       ', GETUTCDATE();
/* Returned:
SYSDATETIME()            2007-05-03 18:34:11.9351421
SYSDATETIMEOFFSET()      2007-05-03 18:34:11.9351421 -07:00
SYSUTCDATETIME()         2007-05-04 01:34:11.9351421
CURRENT_TIMESTAMP        2007-05-03 18:34:11.933
GETDATE()                2007-05-03 18:34:11.933
GETUTCDATE()             2007-05-04 01:34:11.933
*/
```

B. Getting the current system date

```
SELECT 'SYSDATETIME()      ', CONVERT (date, SYSDATETIME());
SELECT 'SYSDATETIMEOFFSET()', CONVERT (date, SYSDATETIMEOFFSET());
SELECT 'SYSUTCDATETIME()   ', CONVERT (date, SYSUTCDATETIME());
SELECT 'CURRENT_TIMESTAMP  ', CONVERT (date, CURRENT_TIMESTAMP);
SELECT 'GETDATE()          ', CONVERT (date, GETDATE());
SELECT 'GETUTCDATE()       ', CONVERT (date, GETUTCDATE());

/* Returned:
SYSDATETIME()            2007-05-03
SYSDATETIMEOFFSET()      2007-05-03
SYSUTCDATETIME()         2007-05-04
CURRENT_TIMESTAMP        2007-05-03
GETDATE()                2007-05-03
GETUTCDATE()             2007-05-04
*/
C. Getting the current system time
SELECT 'SYSDATETIME()      ', CONVERT (time, SYSDATETIME());
SELECT 'SYSDATETIMEOFFSET()', CONVERT (time, SYSDATETIMEOFFSET());
SELECT 'SYSUTCDATETIME()   ', CONVERT (time, SYSUTCDATETIME());
SELECT 'CURRENT_TIMESTAMP  ', CONVERT (time, CURRENT_TIMESTAMP);
SELECT 'GETDATE()          ', CONVERT (time, GETDATE());
SELECT 'GETUTCDATE()       ', CONVERT (time, GETUTCDATE());
/* Returned
SYSDATETIME()            18:25:01.6958841
SYSDATETIMEOFFSET()      18:25:01.6958841
SYSUTCDATETIME()         01:25:01.6958841
CURRENT_TIMESTAMP        18:25:01.6930000
```

```
GETDATE()                   18:25:01.6930000
GETUTCDATE()                01:25:01.6930000
*/
```

3.6.8. ISDATE

Returns 1 if the *expression* is a valid date, time, or datetime value; otherwise, 0.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
ISDATE ( expression )
```

Arguments

expression

Is a character string or expression that can be converted to a character string. The expression must be less than 4,000 characters.

Return Type

int

Remarks

ISDATE is deterministic only if you use it with the CONVERT function, if the CONVERT style parameter is specified, and style is not equal to 0, 100, 9, or 109.

The return value of ISDATE depends on the settings set by SET DATEFORMAT, SET LANGUAGE and default language option. For examples, see example C.

ISDATE expression Formats

For examples of valid formats for which ISDATE will return 1, see the section "Supported String Literal Formats for datetime" in the datetime and smalldatetime topics. For additional examples, also see the Input/Output column of the "Arguments" section of CAST and CONVERT.

The following table summarizes input expression formats that are not valid and that return 0 or an error.

ISDATE expression	ISDATE return value
NULL	0
Values of data types listed in Data Types in any data type category other than character	0

strings, Unicode character strings, or date and time.	
Values of text, ntext, or image data types.	0
Any value that has a seconds precision scale greater than 3, (.0000 through .0000000...n)	0
Any value that mixes a valid date with an invalid value, for example 1995-10-1a.	0

Examples

A. Using ISDATE to test for a valid datetime expression

The following example shows you how to use ISDATE to test whether a character string is a valid datetime.

```
IF ISDATE('2009-05-12 10:19:41.177') = 1
    PRINT 'VALID'
ELSE
    PRINT 'INVALID'
```

B. Showing the effects of the SET DATEFORMAT and SET LANGUAGE settings on return values

The following statements show the values that are returned as a result of the settings of SET DATEFORMAT and SET LANGUAGE.

```
/* Use these sessions settings. */
SET LANGUAGE us_english;
SET DATEFORMAT mdy;
/* Expression in mdy dateformat */
SELECT ISDATE('04/15/2008'); --Returns 1.
/* Expression in mdy dateformat */
SELECT ISDATE('04-15-2008'); --Returns 1.
/* Expression in mdy dateformat */
SELECT ISDATE('04.15.2008'); --Returns 1.
/* Expression in myd  dateformat */
SELECT ISDATE('04/2008/15'); --Returns 1.

SET DATEFORMAT mdy;
SELECT ISDATE('15/04/2008'); --Returns 0.
SET DATEFORMAT mdy;
SELECT ISDATE('15/2008/04'); --Returns 0.
SET DATEFORMAT mdy;
SELECT ISDATE('2008/15/04'); --Returns 0.
SET DATEFORMAT mdy;
SELECT ISDATE('2008/04/15'); --Returns 1.

SET DATEFORMAT dmy;
SELECT ISDATE('15/04/2008'); --Returns 1.
SET DATEFORMAT dym;
SELECT ISDATE('15/2008/04'); --Returns 1.
SET DATEFORMAT ydm;
```

```
SELECT ISDATE('2008/15/04'); --Returns 1.
SET DATEFORMAT ymd;
SELECT ISDATE('2008/04/15'); --Returns 1.

SET LANGUAGE English;
SELECT ISDATE('15/04/2008'); --Returns 0.
SET LANGUAGE Hungarian;
SELECT ISDATE('15/2008/04'); --Returns 0.
SET LANGUAGE Swedish;
SELECT ISDATE('2008/15/04'); --Returns 0.
SET LANGUAGE Italian;
SELECT ISDATE('2008/04/15'); --Returns 1.

/* Return to these sessions settings. */
SET LANGUAGE us_english;
SET DATEFORMAT mdy;
```

3.6.9. MONTH

Returns an integer that represents the month of the specified *date*.

For an overview of all Transact-SQL date and time data types and functions, seeDate and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
MONTH ( date )
```

Arguments

date

Is an expression that can be resolved to a time, date, smalldatetime, datetime, datetime2, or datetimeoffset value. The *date* argument can be an expression, column expression, user-defined variable, or string literal.

Return Type

int

Return Value

MONTH returns the same value as DATEPART (month, *date*).

If *date* contains only a time part, the return value is 1, the base month.

Examples

The following statement returns 4. This is the number of the month.

```
SELECT MONTH('2007-04-30T01:01:01.1234567 -07:00');
```

The following statement returns 1900, 1, 1. The argument for *date* is the number 0. SQL Server interprets 0 as January 1, 1900.

```
SELECT YEAR(0), MONTH(0), DAY(0);
```

3.6.10. SYSDATETIME

Returns a datetime2(7) value that contains the date and time of the computer on which the instance of SQL Server is running.

Note

SYSDATETIME and SYSUTCDATETIME have more fractional seconds precision than GETDATE and GETUTCDATE. SYSDATETIMEOFFSET includes the system time zone offset. SYSDATETIME, SYSUTCDATETIME, and SYSDATETIMEOFFSET can be assigned to a variable of any of the date and time types.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
SYSDATETIME ( )
```

Return Type

datetime2(7)

Remarks

Transact-SQL statements can refer to SYSDATETIME anywhere they can refer to a datetime2(7) expression.

SYSDATETIME is a nondeterministic function. Views and expressions that reference this function in a column cannot be indexed.

Note

SQL Server 2008 obtains the date and time values by using the **GetSystemTimeAsFileTime()** Windows API. The accuracy depends on the computer hardware and version of Windows on which the instance of SQL Server is running. The precision of this API is fixed at 100 nanoseconds. The accuracy can be determined by using the **GetSystemTimeAdjustment()** Windows API.

Examples

The following examples use the six SQL Server system functions that return current date and time to return the date, time or both. The values are returned in series; therefore, their fractional seconds might be different.

A. Getting the current system date and time

```
SELECT SYSDATETIME()
    ,SYSDATETIMEOFFSET()
    ,SYSUTCDATETIME()
    ,CURRENT_TIMESTAMP
    ,GETDATE()
    ,GETUTCDATE();
/* Returned:
SYSDATETIME()      2007-04-30 13:10:02.0474381
SYSDATETIMEOFFSET()2007-04-30 13:10:02.0474381 -07:00
SYSUTCDATETIME()   2007-04-30 20:10:02.0474381
CURRENT_TIMESTAMP  2007-04-30 13:10:02.047
GETDATE()          2007-04-30 13:10:02.047
GETUTCDATE()       2007-04-30 20:10:02.047
```

B. Getting the current system date

```
SELECT CONVERT (date, SYSDATETIME())
    ,CONVERT (date, SYSDATETIMEOFFSET())
    ,CONVERT (date, SYSUTCDATETIME())
    ,CONVERT (date, CURRENT_TIMESTAMP)
    ,CONVERT (date, GETDATE())
    ,CONVERT (date, GETUTCDATE());

/* All returned 2007-04-30 */
```

C. Getting the current system time

```
SELECT CONVERT (time, SYSDATETIME())
    ,CONVERT (time, SYSDATETIMEOFFSET())
    ,CONVERT (time, SYSUTCDATETIME())
    ,CONVERT (time, CURRENT_TIMESTAMP)
    ,CONVERT (time, GETDATE())
    ,CONVERT (time, GETUTCDATE());

/* Returned
SYSDATETIME()      13:18:45.3490361
SYSDATETIMEOFFSET()13:18:45.3490361
SYSUTCDATETIME()   20:18:45.3490361
CURRENT_TIMESTAMP  13:18:45.3470000
GETDATE()          13:18:45.3470000
GETUTCDATE()       20:18:45.3470000
*/
```

3.6.11. YEAR

Returns an integer that represents the year of the specified *date*.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

Syntax

```
YEAR ( date )
```

Arguments

date

Is an expression that can be resolved to a time, date, smalldatetime, datetime, datetime2, or datetimeoffset value. The *date* argument can be an expression, column expression, user-defined variable or string literal.

Return Types

int

Return Value

YEAR returns the same value as DATEPART (year, *date*).

If *date* only contains a time part, the return value is 1900, the base year.

Examples

The following statement returns 2007. This is the number of the year.

```
SELECT YEAR('2007-04-30T01:01:01.1234567-07:00');
```

The following statement returns 1900, 1, 1. The argument for *date* is the number 0. SQL Server interprets 0 as January 1, 1900.

```
SELECT YEAR(0), MONTH(0), DAY(0);
```

3.7. Mathematical Functions

The following scalar functions perform a calculation, usually based on input values that are provided as arguments, and return a numeric value:

ABS	A mathematical function that returns the absolute (positive) value of the specified numeric expression. `SELECT ABS(-1.0), ABS(0.0), ABS(1.0)` `---- ---- ----` `1.0  .0   1.0`
ACOS	A mathematical function that returns the angle, in radians, whose cosine is the specified fl

	oat expression; also called arccosine. **SET NOCOUNT OFF;** **DECLARE @cos float;** **SET @cos = -1.0;** **SELECT 'The ACOS of the number is: ' + CONVERT(varchar,** **ACOS(@cos));** **---** **The ACOS of the number is: 3.14159**
ASIN	Returns the angle, in radians, whose sine is the specified float expression. This is also calle d arcsine. **/* The first value will be -1.01. This fails because the value is outside the range.*/** **DECLARE @angle float** **SET @angle = -1.01** **SELECT 'The ASIN of the angle is: ' + CONVERT(varchar,** **ASIN(@angle))** **GO** **-- The next value is -1.00.** **DECLARE @angle float** **SET @angle = -1.00** **SELECT 'The ASIN of the angle is: ' + CONVERT(varchar,** **ASIN(@angle))** **GO** **-- The next value is 0.1472738.** **DECLARE @angle float** **SET @angle = 0.1472738** **SELECT 'The ASIN of the angle is: ' + CONVERT(varchar,** **ASIN(@angle))** **GO** **---** **.Net SqlClient Data Provider: Msg 3622, Level 16, State 1, Line 3** **A domain error occurred.** **---** **The ASIN of the angle is: -1.5708** **(1 row(s) affected)** **---** **The ASIN of the angle is: 0.147811** **(1 row(s) affected)**
ATAN	Returns the angle in radians whose tangent is a specified float expression. This is also calle d arctangent.

	SELECT 'The ATAN of -45.01 is: ' + CONVERT(varchar, ATAN(-45.01)) SELECT 'The ATAN of -181.01 is: ' + CONVERT(varchar, ATAN(-181.01)) SELECT 'The ATAN of 0 is: ' + CONVERT(varchar, ATAN(0)) SELECT 'The ATAN of 0.1472738 is: ' + CONVERT(varchar, ATAN(0.1472738)) SELECT 'The ATAN of 197.1099392 is: ' + CONVERT(varchar, ATAN(197.1099392)) GO ------------------------------ The ATAN of -45.01 is: -1.54858 (1 row(s) affected) ------------------------------ The ATAN of -181.01 is: -1.56527 (1 row(s) affected) ------------------------------ The ATAN of 0 is: 0 (1 row(s) affected) --------------------------------- The ATAN of 0.1472738 is: 0.146223 (1 row(s) affected) --------------------------------- The ATAN of 197.1099392 is: 1.56572 (1 row(s) affected)
ATN2	Returns the angle, in radians, between the positive x-axis and the ray from the origin to th e point (y, x), where x and y are the values of the two specified float expressions. DECLARE @x float DECLARE @y float SET @x = 35.175643 SET @y = 129.44 SELECT 'The ATN2 of the angle is: ' + CONVERT(varchar,ATN2(@x,@y)) GO The ATN2 of the angle is: 0.265345 (1 row(s) affected)

CEILING	Returns the smallest integer greater than, or equal to, the specified numeric expression. **SELECT CEILING($123.45), CEILING($-123.45), CEILING($0.0)** **GO** **--------- --------- ------------------------** **124.00 -123.00 0.00** **(1 row(s) affected)**
COS	Is a mathematical function that returns the trigonometric cosine of the specified angle, in radians, in the specified expression. **DECLARE @angle float** **SET @angle = 14.78** **SELECT 'The COS of the angle is: ' + CONVERT(varchar,COS(@angle))** **GO** The COS of the angle is: -0.599465 (1 row(s) affected)
COT	A mathematical function that returns the trigonometric cotangent of the specified angle, in radians, in the specified float expression. **DECLARE @angle float** **SET @angle = 124.1332** **SELECT 'The COT of the angle is: ' + CONVERT(varchar,COT(@angle))** **GO** **The COT of the angle is: -0.040312** **(1 row(s) affected)**
DEGREES	Returns the corresponding angle in degrees for an angle specified in radians. **SELECT 'The number of degrees in PI/2 radians is: ' +** **CONVERT(varchar, DEGREES((PI()/2)));** **GO** **The number of degrees in PI/2 radians is 90** **(1 row(s) affected)**
EXP	Returns the exponential value of the specified float expression. **DECLARE @var float** **SET @var = 10** **SELECT 'The EXP of the variable is: ' + CONVERT(varchar,EXP(@var))** **GO**

	`` --- The EXP of the variable is: 22026.5 (1 row(s) affected) ```
FLOOR	Returns the largest integer less than or equal to the specified numeric expression. ``` SELECT FLOOR(123.45), FLOOR(-123.45), FLOOR($123.45) --------- --------- ----------- 123 -124 123.0000 ```
LOG	Returns the natural logarithm of the specified float expression. ``` DECLARE @var float; SET @var = 10; SELECT 'The LOG of the variable is: ' + CONVERT(varchar, LOG(@var)); GO ----------------------------------- The LOG of the variable is: 2.30259 (1 row(s) affected) ```
LOG10	Returns the base-10 logarithm of the specified float expression. ``` DECLARE @var float; SET @var = 145.175643; SELECT 'The LOG10 of the variable is: ' + CONVERT(varchar,LOG10(@var)); GO The LOG10 of the variable is: 2.16189 (1 row(s) affected) ```
PI	Returns the constant value of PI. ``` SELECT PI() GO ------------------------ 3.14159265358979 (1 row(s) affected) ```
POWER	Returns the value of the specified expression to the specified power. ``` SELECT POWER(2.0, -100.0); GO -- 0.0 (1 row(s) affected) ```

```
DECLARE @value int, @counter int;
SET @value = 2;
SET @counter = 1;

WHILE @counter < 5
   BEGIN
      SELECT POWER(@value, @counter)
      SET NOCOUNT ON
      SET @counter = @counter + 1
      SET NOCOUNT OFF
   END;
GO
-----------
2
(1 row(s) affected)

-----------
4
(1 row(s) affected)

-----------
8
(1 row(s) affected)

-----------
16
(1 row(s) affected)
```

RADIANS

Returns radians when a numeric expression, in degrees, is entered.

```
SELECT RADIANS(1e-307)
GO
-------------------
0.0
(1 row(s) affected)

-- First value is -45.01.
DECLARE @angle float
SET @angle = -45.01
SELECT 'The RADIANS of the angle is: ' +
   CONVERT(varchar, RADIANS(@angle))
GO
-- Next value is -181.01.
DECLARE @angle float
SET @angle = -181.01
SELECT 'The RADIANS of the angle is: ' +
   CONVERT(varchar, RADIANS(@angle))
GO
-- Next value is 0.00.
DECLARE @angle float
SET @angle = 0.00
SELECT 'The RADIANS of the angle is: ' +
```

	CONVERT(varchar, RADIANS(@angle)) GO -- Next value is 0.1472738. DECLARE @angle float SET @angle = 0.1472738 SELECT 'The RADIANS of the angle is: ' + CONVERT(varchar, RADIANS(@angle)) GO -- Last value is 197.1099392. DECLARE @angle float SET @angle = 197.1099392 SELECT 'The RADIANS of the angle is: ' + CONVERT(varchar, RADIANS(@angle)) GO --- The RADIANS of the angle is: -0.785573 (1 row(s) affected) --- The RADIANS of the angle is: -3.15922 (1 row(s) affected) --- The RADIANS of the angle is: 0 (1 row(s) affected) --- The RADIANS of the angle is: 0.00257041 (1 row(s) affected) --- The RADIANS of the angle is: 3.44022 (1 row(s) affected)
RAND	Returns a pseudo-random float value from 0 through 1, exclusive. DECLARE @counter smallint; SET @counter = 1; WHILE @counter < 5 BEGIN SELECT RAND() Random_Number SET @counter = @counter + 1 END; GO
ROUND	
SIGN	Returns the positive (+1), zero (0), or negative (-1) sign of the specified expression. DECLARE @value real SET @value = -1 WHILE @value < 2 BEGIN SELECT SIGN(@value) SET NOCOUNT ON SELECT @value = @value + 1 SET NOCOUNT OFF

	END SET NOCOUNT OFF GO (1 row(s) affected) ---------- -1.0 (1 row(s) affected) ---------- 0.0 (1 row(s) affected) ---------- 1.0 (1 row(s) affected)
SIN	Returns the trigonometric sine of the specified angle, in radians, and in an approximate numeric, float, expression. DECLARE @angle float SET @angle = 45.175643 SELECT 'The SIN of the angle is: ' + CONVERT(varchar,SIN(@angle)) GO The SIN of the angle is: 0.929607 (1 row(s) affected)
SQRT	Returns the square root of the specified float value. DECLARE @myvalue float; SET @myvalue = 1.00; WHILE @myvalue < 10.00 BEGIN SELECT SQRT(@myvalue); SET @myvalue = @myvalue + 1 END; GO ---------- 1.0 ---------- 1.4142135623731 ---------- 1.73205080756888 ---------- 2.0 ---------- 2.23606797749979 ---------- 2.44948974278318

	---------- 2.64575131106459 ---------- 2.82842712474619 ---------- 3.0
SQUARE	Returns the square of the specified float value. `DECLARE @h float, @r float` `SET @h = 5` `SET @r = 1` `SELECT PI()* SQUARE(@r)* @h AS 'Cyl Vol'` Cyl Vol ---------- 15.707963267948966
TAN	Returns the tangent of the input expression. `SELECT TAN(PI()/2);` `----------` 1.6331778728383844E+16

Note

Arithmetic functions, such as ABS, CEILING, DEGREES, FLOOR, POWER, RADIANS, and SIGN, return a value having the same data type as the input value. Trigonometric and other functions, including EXP, LOG, LOG10, SQUARE, and SQRT, cast their input values to **float** and return a **float** value.

All mathematical functions, except for RAND, are deterministic functions. This means they return the same results each time they are called with a specific set of input values. RAND is deterministic only when a seed parameter is specified. For more information about function determinism, see Deterministic and Nondeterministic Functions.

3.7.1. ROUND

Returns a numeric value, rounded to the specified length or precision.

Syntax

```
ROUND ( numeric_expression , length [ ,function ] )
```

Arguments

numeric_expression

Is an expression of the exact numeric or approximate numeric data type category, except for the bit data type.

length

Is the precision to which *numeric_expression* is to be rounded. *length* must be an expression of type tinyint, smallint, or int. When *length* is a positive number, *numeric_expression* is rounded to the number of decimal positions specified by *length*. When *length* is a negative number, *numeric_expression* is rounded on the left side of the decimal point, as specified by *length*.

function

Is the type of operation to perform. *function* must be tinyint, smallint, or int. When *function* is omitted or has a value of 0 (default), *numeric_expression* is rounded. When a value other than 0 is specified, *numeric_expression* is truncated.

Return Types

Returns the following data types.

Expression result	Return type
tinyint	int
smallint	int
int	int
bigint	bigint
decimal and numeric category (p, s)	decimal(p, s)
money and smallmoney category	money
float and real category	float

Remarks

ROUND always returns a value. If *length* is negative and larger than the number of digits before the decimal point, ROUND returns 0.

Example	Result
`ROUND(748.58, -4)`	0

ROUND returns a rounded *numeric_expression*, regardless of data type, when *length* is a negative number.

Examples	Result
`ROUND(748.58, -1)`	750.00
`ROUND(748.58, -2)`	700.00
`ROUND(748.58, -3)`	Results in an arithmetic overflow, because 748.58 defaults to decimal(5,2), which cannot return 1000.00.
To round up to 4 digits, change the data type of the input. For example: `SELECT ROUND(CAST (748.58 AS decimal (6,2)),-3);`	1000.00

Examples

A. Using ROUND and estimates

The following example shows two expressions that demonstrate by using ROUND the last digit is always an estimate.

```
SELECT ROUND(123.9994, 3), ROUND(123.9995, 3)
GO
```

Here is the result set.

```
----------- -----------
123.9990    124.0000
```

B. Using ROUND and rounding approximations

The following example shows rounding and approximations.

```
SELECT ROUND(123.4545, 2);
GO
SELECT ROUND(123.45, -2);
GO
```

Here is the result set.

```
----------
123.4500
(1 row(s) affected)
--------
100.00
(1 row(s) affected)
```

C. Using ROUND to truncate

The following example uses two SELECT statements to demonstrate the difference between rounding and truncation. The first statement rounds the result. The second statement truncates the result.

```
SELECT ROUND(150.75, 0);
GO
SELECT ROUND(150.75, 0, 1);
GO
```

Here is the result set.

```
--------
151.00

(1 row(s) affected)

--------
150.00

(1 row(s) affected)
```

3.8. Metadata Functions

The following scalar functions return information about the database and database objects:

@@PROCID	Returns the object identifier (ID) of the current Transact-SQL module. A Transact-SQL module can be a stored procedure, user-defined function, or trigger. @@PROCID cannot be specified in CLR modules or the in-process data access provider.
APP_NAME	Returns the application name for the current session if set by the application.
APPLOCK_MODE	Returns the lock mode held by the lock owner on a particular application resource. APPLOCK_MODE is an application lock function, and it operates on the current database. The scope of application locks is the database.
APPLOCK_TEST	Returns information about whether or not a lock can be granted on a particular application resource for a specified lock owner without acquiring the lock. APPLOCK_TEST is an application lock function, and it operates on the current database. The scope of application

	locks is the database.
ASSEMBLYPROPERTY	Returns information about a property of an assembly.
COL_LENGTH	Returns the defined length, in bytes, of a column. **Syntax: COL_LENGTH ('table' , 'column')** **USE AdventureWorks2008R2;** **GO** **CREATE TABLE t1** **(c1 varchar(40),** **c2 nvarchar(40)** **);** **GO** **SELECT COL_LENGTH('t1','c1')AS 'VarChar',** **COL_LENGTH('t1','c2')AS 'NVarChar';** **GO** **DROP TABLE t1;** **VarChar NVarChar** **40 80**
COL_NAME	Returns the name of a column from a specified corresponding table identification number and column identification number. Syntax: **COL_NAME (table_id , column_id)**
COLUMNPROPERTY	
DATABASE_PRINCIPAL_ID	Returns the ID number of a principal in the current database. *Principals* are entities that can request SQL Server resources. For more information about principals, see Principals (Database Engine).
DATABASEPROPERTY	Returns the named database property value for the specified database and property name.
DATABASEPROPERTYEX	Returns the current setting of the specified database option or property for the specified database.
DB_ID	Returns the database identification (ID) number.
DB_NAME	Returns the database name.
FILE_ID	Returns the file identification (ID) number for the given logical file name in the current database.

FILE_IDEX	Returns the file identification (ID) number for the specified logical file name of the data, log, or full-text file in the current database.
FILE_NAME	Returns the logical file name for the given file identification (ID) number.
FILEGROUP_ID	Returns the filegroup identification (ID) number for a specified filegroup name.
FILEGROUP_NAME	Returns the filegroup name for the specified filegroup identification (ID) number.
FILEGROUPPROPERTY	Returns the specified filegroup property value when supplied with a filegroup and property name.
FILEPROPERTY	Returns the specified file name property value when a file name in the current database and a property name are specified. Returns NULL for files that are not in the current database.
FULLTEXTCATALOGPROPERTY	Returns information about full-text catalog properties.
FULLTEXTSERVICEPROPERTY	Returns information related to the properties of the Full-Text Engine. These properties can be set and retrieved by using **sp_fulltext_service**.
INDEX_COL	Returns the indexed column name. Returns NULL for XML indexes.
INDEXKEY_PROPERTY	Returns information about the index key. Returns NULL for XML indexes.
INDEXPROPERTY	Returns the named index or statistics property value of a specified table identification number, index or statistics name, and property name. Returns NULL for XML indexes.
OBJECT_DEFINITION	Returns the Transact-SQL source text of the definition of a specified object.
OBJECT_ID	Returns the database object identification number of a schema-scoped object.

OBJECT_NAME	Returns the database object name for schema-scoped objects. For a list of schema-scoped objects, see sys.objects.
OBJECT_SCHEMA_NAME	Returns the database schema name for schema-scoped objects. For a list of schema-scoped objects, see sys.objects.
OBJECTPROPERTY	Returns information about schema-scoped objects in the current database. For a list of schema-scoped objects, see sys.objects. This function cannot be used for objects that are not schema-scoped, such as data definition language (DDL) triggers and event notifications.
OBJECTPROPERTYEX	Returns information about schema-scoped objects in the current database. For a list of these objects, see sys.objects. OBJECTPROPERTYEX cannot be used for objects that are not schema-scoped, such as data definition language (DDL) triggers and event notifications.
ORIGINAL_DB_NAME	Returns the database name that is specified by the user in the database connection string. This is the database that is specified by using the **sqlcmd -d** option (USE *database*) or the ODBC data source expression (initial catalog =*databasename*).
PARSENAME	Returns the specified part of an object name. The parts of an object that can be retrieved are the object name, owner name, database name, and server name.
SCHEMA_ID	Returns the schema ID associated with a schema name.
SCHEMA_NAME	Returns the schema name associated with a schema ID.
SCOPE_IDENTITY	Returns the last identity value inserted into an identity column in the same scope. A scope is a module: a stored procedure, trigger, function, or batch. Therefore, two statements are in the same scope if they are in the same stored procedure, function, or batch.
SERVERPROPERTY	Returns property information about the server instance in SQL Server 2008 R2.

STATS_DATE	Returns the date of the most recent update for statistics on a table or indexed view.
TYPE_ID	Returns the ID for a specified data type name.
TYPE_NAME	Returns the unqualified type name of a specified type ID.
TYPEPROPERTY	Returns information about a data type.

All metadata functions are nondeterministic. This means these functions do not always return the same results every time they are called, even with the same set of input values. For more information about function determinism, see Deterministic and Nondeterministic Functions.

3.8.1. COLUMNPROPERTY

Returns information about a column or parameter.

Syntax

```
COLUMNPROPERTY ( id , column , property )
```

Arguments

id

Is an expression that contains the identifier (ID) of the table or procedure.

column

Is an expression that contains the name of the column or parameter.

property

Is an expression that contains the information to be returned for *id*, and can be any one of the following values.

Value	Description	Value returned
AllowsNull	Allows null values.	1 = TRUE 0 = FALSE NULL = Input is not valid.
ColumnId	Column ID value corresponding to **sys.columns.column_id**.	Column ID **Note**

		When querying multiple columns, gaps may appear in the sequence of Column ID values.
FullTextTypeColumn	The TYPE COLUMN in the table that holds the document type information of the *column*.	ID of the full-text TYPE COLUMN for the column passed as the second parameter of this property.
IsComputed	Column is a computed column.	1 = TRUE 0 = FALSE NULL = Input is not valid.
IsCursorType	Procedure parameter is of type CURSOR.	1 = TRUE 0 = FALSE NULL = Input is not valid.
IsDeterministic	Column is deterministic. This property applies only to computed columns and view columns.	1 = TRUE 0 = FALSE NULL = Input is not valid. Not a computed column or view column.
IsFulltextIndexed	Column has been registered for full-text indexing.	1 = TRUE 0 = FALSE NULL = Input is not valid.
IsIdentity	Column uses the IDENTITY property.	1 = TRUE 0 = FALSE NULL = Input is not valid.
IsIdNotForRepl	Column checks for the IDENTITY_INSERT setting. If IDENTITY NOT FOR REPLICATION is specified, the IDENTITY_INSERT setting is not checked.	1 = TRUE 0 = FALSE NULL = Input is not valid.

IsIndexable	Column can be indexed.	1 = TRUE 0 = FALSE NULL = Input is not valid.
IsOutParam	Procedure parameter is an output parameter.	1 = TRUE 0 = FALSE NULL = Input is not valid.
IsPrecise	Column is precise. This property applies only to deterministic columns.	1 = TRUE 0 = FALSE NULL = Input is not valid. Not a deterministic column
IsRowGuidCol	Column has the uniqueidentifier data type and is defined with the ROWGUIDCOL property.	1 = TRUE 0 = FALSE NULL = Input is not valid.
IsSystemVerified	The determinism and precision properties of the column can be verified by the Database Engine. This property applies only to computed columns and columns of views.	1 = TRUE 0 = FALSE NULL = Input is not valid.
IsXmlIndexable	The XML column can be used in an XML index.	1 = TRUE 0 = FALSE NULL = Input is not valid.
Precision	Length for the data type of the column or parameter.	The length of the specified column data type -1 = xml or large value types NULL = Input is not valid.
Scale	Scale for the data type of the column or parameter.	The scale NULL = Input is not valid.
SystemDataAccess	Column is derived from a function that accesses data in the	1 = TRUE (Indicates read-only access.)

	system catalogs or virtual system tables of SQL Server. This property applies only to computed columns and columns of views.	0 = FALSE NULL = Input is not valid.
UserDataAccess	Column is derived from a function that accesses data in user tables, including views and temporary tables, stored in the local instance of SQL Server. This property applies only to computed columns and columns of views.	1 = TRUE (Indicates read-only access.) 0 = FALSE NULL = Input is not valid.
UsesAnsiTrim	ANSI_PADDING was set ON when the table was first created. This property applies only to columns or parameters of type char or varchar.	1= TRUE 0= FALSE NULL = Input is not valid.
IsSparse	Column is a sparse column. For more information, see Using Sparse Columns.	1= TRUE 0= FALSE NULL = Input is not valid.
IsColumnSet	Column is a column set. For more information, see Using Column Sets.	1= TRUE 0= FALSE NULL = Input is not valid.

Return Types

int

Exceptions

Returns NULL on error or if a caller does not have permission to view the object.

In SQL Server 2008, a user can only view the metadata of securables that the user owns or on which the user has been granted permission. This means that metadata-emitting, built-in functions such as

COLUMNPROPERTY may return NULL if the user does not have any permission on the object. For more information, see Metadata Visibility Configuration and Troubleshooting Metadata Visibility.

Remarks

When you check the deterministic property of a column, first test whether the column is a computed column. **IsDeterministic** returns NULL for noncomputed columns. Computed columns can be specified as index columns.

Examples

The following example returns the length of the LastName column.

```
USE AdventureWorks2008R2;
GO
SELECT COLUMNPROPERTY( OBJECT_ID('Person.Person'),'LastName','PRECISION')AS '
Column Length';
GO
```

Here is the result set.

```
Column Length
-------------
50

(1 row(s) affected)
```

3.9. Rowset Functions

The following rowset functions return an object that can be used in place of a table reference in a Transact-SQL statement.

CONTAINSTABLE	OPENQUERY
FREETEXTTABLE	OPENROWSET
OPENDATASOURCE	OPENXML

All rowset functions are nondeterministic. This means these functions do not always return the same results every time they are called, even with the same set of input values. For more information about function determinism, see Deterministic and Nondeterministic Functions.

3.9.1. CONTAINSTABLE

Returns a table of zero, one, or more rows for those columns containing character-based data types for precise or fuzzy (less precise) matches to single words and phrases, the proximity of words within a

certain distance of one another, or weighted matches. CONTAINSTABLE can only be referenced in the FROM clause of a SELECT statement as if it were a regular table name.

Queries using CONTAINSTABLE specify contains-type full-text queries that return a relevance ranking value (RANK) and full-text key (KEY) for each row. The CONTAINSTABLE function uses the same search conditions as the CONTAINS predicate.

Syntax

```
CONTAINSTABLE ( table , { column_name | (column_list ) | * } , ' < contains_s
earch_condition > '
     [ , LANGUAGE language_term]
  [ , top_n_by_rank ]
          )
< contains_search_condition > ::=
    { < simple_term >
    | < prefix_term >
    | < generation_term >
    | < proximity_term >
    |  < weighted_term >
    }
    | { ( < contains_search_condition > )
    { { AND | & } | { AND NOT | &! } | { OR | | } }
     < contains_search_condition > [ ...n ]
    }
< simple_term > ::=
     word | " phrase "
< prefix term > ::=
     { "word * " | "phrase *" }
< generation_term > ::=
     FORMSOF ( { INFLECTIONAL | THESAURUS } , < simple_term > [ ,...n ] )
< proximity_term > ::=
     { < simple_term > | < prefix_term > }
     { { NEAR | ~ } { < simple_term > | < prefix_term > } } [ ...n ]
< weighted_term > ::=
     ISABOUT
    ( { {
  < simple_term >
  | < prefix_term >
  | < generation_term >
  | < proximity_term >
  }
   [ WEIGHT ( weight_value ) ]
   } [ ,...n ]
    )
```

Arguments

table

Is the name of a table that has been full-text indexed. *table* can be a one-, two-, three-, or four-part database object name. When querying a view, only one full-text indexed base table can be involved.

table cannot specify a server name and cannot be used in queries against linked servers.

column_name

Is the name of one or more columns that are indexed for full-text searching. The columns can be of type char, varchar, nchar, nvarchar, text, ntext, image, xml, varbinary, or varbinary(max).

column_list

Indicates that several columns, separated by a comma, can be specified. *column_list* must be enclosed in parentheses. Unless *language_term* is specified, the language of all columns of *column_list* must be the same.

*

Specifies that all full-text indexed columns in *table* should be used to search for the given search condition. Unless *language_term* is specified, the language of all columns of the table must be the same.

LANGUAGE *language_term*

Is the language whose resources will be used for word breaking, stemming, and thesaurus and noise-word (or stopword) removal as part of the query. This parameter is optional and can be specified as a string, integer, or hexadecimal value corresponding to the locale identifier (LCID) of a language. If *language_term* is specified, the language it represents will be applied to all elements of the search condition. If no value is specified, the column full-text language is used.

If documents of different languages are stored together as binary large objects (BLOBs) in a single column, the locale identifier (LCID) of a given document determines what language is used to index its content. When querying such a column, specifying *LANGUAGE language_term* can increase the probability of a good match.

When specified as a string, *language_term* corresponds to the **alias** column value in the sys.syslanguages compatibility view. The string must be enclosed in single quotation marks, as in '*language_term*'. When specified as an integer, *language_term* is the actual LCID that identifies the language. When specified as a hexadecimal value, *language_term* is 0x followed by the hexadecimal value of the LCID. The hexadecimal value must not exceed eight digits, including leading zeros.

If the value is in double-byte character set (DBCS) format, Microsoft SQL Server will convert it to Unicode.

If the language specified is not valid or there are no resources installed that correspond to that language, SQL Server returns an error. To use the neutral language resources, specify 0x0 as *language_term*.

top_n_by_rank

Specifies that only the *n*highest ranked matches, in descending order, are returned. Applies only when an integer value, *n*, is specified. If *top_n_by_rank* is combined with other parameters, the query could return fewer rows than the number of rows that actually match all the predicates. *top_n_by_rank* allows you to increase query performance by recalling only the most relevant hits.

<contains_search_condition>

Specifies the text to search for in *column_name* and the conditions for a match. For information about search conditions, see CONTAINS.

Remarks

Full-text predicates and functions work on a single table, which is implied in the FROM predicate. To search on multiple tables, use a joined table in your FROM clause to search on a result set that is the product of two or more tables.

The table returned has a column named **KEY** that contains full-text key values. Each full-text indexed table has a column whose values are guaranteed to be unique, and the values returned in the **KEY** column are the full-text key values of the rows that match the selection criteria specified in the contains search condition. The **TableFulltextKeyColumn** property, obtained from the OBJECTPROPERTYEX function, provides the identity of this unique key column. To obtain the ID of the column associated with the full-text key of the full-text index, use **sys.fulltext_indexes**. For more information, see sys.fulltext_indexes.

To obtain the rows you want from the original table, specify a join with the CONTAINSTABLE rows. The typical form of the FROM clause for a SELECT statement using CONTAINSTABLE is:

```
SELECT select_list
FROM table AS FT_TBL INNER JOIN
   CONTAINSTABLE(table, column, contains_search_condition) AS KEY_TBL
   ON FT_TBL.unique_key_column = KEY_TBL.[KEY]
```

The table produced by CONTAINSTABLE includes a column named **RANK**. The **RANK** column is a value (from 0 through 1000) for each row indicating how well a row matched the selection criteria. This rank value is typically used in one of these ways in the SELECT statement:

- In the ORDER BY clause to return the highest-ranking rows as the first rows in the table.
- In the select list to see the rank value assigned to each row.

CONTAINSTABLE is not recognized as a keyword if the compatibility level is less than 70. For more information, see sp_dbcmptlevel.

Permissions

Execute permissions are available only by users with the appropriate SELECT privileges on the table or the referenced table's columns.

Examples

A. Returning rank values using CONTAINSTABLE

The following example searches for all product names containing the words breads, fish, or beers, and different weightings are given to each word. For each returned row matching this search criteria, the relative closeness (ranking value) of the match is shown. In addition, the highest ranking rows are returned first.

```
USE Northwind;
GO
SELECT FT_TBL.CategoryName, FT_TBL.Description, KEY_TBL.RANK
    FROM Categories AS FT_TBL
        INNER JOIN CONTAINSTABLE(Categories, Description,
        'ISABOUT (breads weight (.8),
        fish weight (.4), beers weight (.2) )' ) AS KEY_TBL
            ON FT_TBL.CategoryID = KEY_TBL.[KEY]
ORDER BY KEY_TBL.RANK DESC;
GO
```

B. Returning rank values greater than specified value using CONTAINSTABLE

The following example returns the description and category name of all food categories for which the Description column contains the words "sweet and savory" near either the word sauces or the word candies. All rows that have a category name Seafood are disregarded. Only rows with a rank value of 2 or higher are returned.

```
USE Northwind;
GO
SELECT FT_TBL.Description, FT_TBL.CategoryName, KEY_TBL.RANK
FROM Categories AS FT_TBL
    INNER JOIN CONTAINSTABLE (Categories, Description,
        '("sweet and savory" NEAR sauces) OR
        ("sweet and savory" NEAR candies)'
        ) AS KEY_TBL
        ON FT_TBL.CategoryID = KEY_TBL.[KEY]
WHERE KEY_TBL.RANK > 2
    AND FT_TBL.CategoryName <> 'Seafood'
ORDER BY KEY_TBL.RANK DESC;
GO
```

C. Returning top 10 ranked results using CONTAINSTABLE and top_n_by_rank

The following example returns the description and category name of the top 10 food categories where the Description column contains the words "sweet and savory" near either the word "sauces" or the word "candies".

```
USE Northwind;
SELECT FT_TBL.Description, FT_TBL.CategoryName , KEY_TBL.RANK
FROM Categories AS FT_TBL
    INNER JOIN CONTAINSTABLE (Categories, Description,
        '("sweet and savory" NEAR sauces) OR
        ("sweet and savory" NEAR candies)', 10)
        AS KEY_TBL
```

```
        ON FT_TBL.CategoryID = KEY_TBL.[KEY]
GO
```

D. Specifying the LANGUAGE argument

The following example shows using the LANGUAGE argument.

```
USE Northwind;
SELECT FT_TBL.Description , FT_TBL.CategoryName , KEY_TBL.RANK
FROM dbo.Categories AS FT_TBL
    INNER JOIN CONTAINSTABLE (dbo.Categories, Description,
        '("sweet and savory" NEAR sauces) OR
        ("sweet and savory" NEAR candies)',LANGUAGE N'English', 10)
        AS KEY_TBL
        ON FT_TBL.CategoryID = KEY_TBL.[KEY];
```

3.9.2. FREETEXTTABLE

Returns a table of zero, one, or more rows for those columns containing character-based data types for values that match the meaning, but not the exact wording, of the text in the specified *freetext_string*. FREETEXTTABLE can only be referenced in the FROM clause of a SELECT statement like a regular table name.

Queries using FREETEXTTABLE specify freetext-type full-text queries that return a relevance ranking value (RANK) and full-text key (KEY) for each row.

Syntax

```
FREETEXTTABLE (table , { column_name | (column_list) | * }
          , 'freetext_string'
     [ , LANGUAGE language_term ]
     [ , top_n_by_rank ] )
```

Arguments

table

Is the name of the table that has been marked for full-text querying. *table* or *view*can be a one-, two-, or three-part database object name. When querying a view, only one full-text indexed base table can be involved.

table cannot specify a server name and cannot be used in queries against linked servers.

column_name

Is the name of one or more full-text indexed columns of the table specified in the FROM clause. The columns can be of type char, varchar, nchar, nvarchar, text, ntext, image, xml, varbinary, or varbinary(max).

column_list

Indicates that several columns, separated by a comma, can be specified. *column_list* must be enclosed in parentheses. Unless *language_term* is specified, the language of all columns of *column_list* must be the same.

*

Specifies that all columns that have been registered for full-text searching should be used to search for the given *freetext_string*. Unless *language_term* is specified, the language of all full-text indexed columns in the table must be the same.

freetext_string

Is text to search for in the *column_name*. Any text, including words, phrases or sentences, can be entered. Matches are generated if any term or the forms of any term is found in the full-text index.

Unlike in the CONTAINS search condition where AND is a keyword, when used in *freetext_string* the word 'and' is considered a noise word, or stopword, and will be discarded.

Use of WEIGHT, FORMSOF, wildcards, NEAR and other syntax is not allowed. *freetext_string* is wordbroken, stemmed, and passed through the thesaurus. If *freetext_string* is enclosed in double quotation marks, a phrase match is instead performed; stemming and thesaurus are not performed.

LANGUAGE *language_term*

Is the language whose resources will be used for word breaking, stemming, and thesaurus and stopword removal as part of the query. This parameter is optional and can be specified as a string, integer, or hexadecimal value corresponding to the locale identifier (LCID) of a language. If *language_term* is specified, the language it represents will be applied to all elements of the search condition. If no value is specified, the column full-text language is used.

If documents of different languages are stored together as binary large objects (BLOBs) in a single column, the locale identifier (LCID) of a given document determines what language is used to index its content. When querying such a column, specifying *LANGUAGE language_term* can increase the probability of a good match.

When specified as a string, *language_term* corresponds to the **alias** column value in the sys.syslanguages compatibility view. The string must be enclosed in single quotation marks, as in '*language_term*'. When specified as an integer, *language_term* is the actual LCID that identifies the language. When specified as a hexadecimal value, *language_term* is 0x followed by the hexadecimal value of the LCID. The hexadecimal value must not exceed eight digits, including leading zeros.

If the value is in double-byte character set (DBCS) format, Microsoft SQL Server will convert it to Unicode.

If the language specified is not valid or there are no resources installed that correspond to that language, SQL Server returns an error. To use the neutral language resources, specify 0x0 as *language_term*.

top_n_by_rank

Specifies that only the *n*highest ranked matches, in descending order, are returned. Applies only when an integer value, *n*, is specified. If *top_n_by_rank* is combined with other parameters, the query could return fewer rows than the number of rows that actually match all the predicates. *top_n_by_rank* allows you to increase query performance by recalling only the most relevant hits.

Remarks

Full-text predicates and functions work on a single table, which is implied in the FROM predicate. To search on multiple tables, use a joined table in your FROM clause to search on a result set that is the product of two or more tables.

FREETEXTTABLE uses the same search conditions as the FREETEXT predicate.

Like CONTAINSTABLE, the table returned has columns named **KEY** and **RANK**, which are referenced within the query to obtain the appropriate rows and use the row ranking values.

FREETEXTTABLE is not recognized as a keyword if the compatibility level is less than 70. For more information, see sp_dbcmptlevel.

Permissions

FREETEXTTABLE can be invoked only by users with appropriate SELECT privileges for the specified table or the referenced columns of the table.

Examples

The following example returns the category name and description of all categories that relate to sweet, candy, bread, dry, or meat.

Note

To run this example, you will have to install the **Northwind** database. For information about how to install the **Northwind** database, see Downloading Northwind and pubs Sample Databases.

```
USE Northwind;
SELECT FT_TBL.CategoryName
    ,FT_TBL.Description
    ,KEY_TBL.RANK
FROM dbo.Categories AS FT_TBL
    INNER JOIN FREETEXTTABLE(dbo.Categories, Description,
        'sweetest candy bread and dry meat') AS KEY_TBL
        ON FT_TBL.CategoryID = KEY_TBL.[KEY];
GO
```

The following example is identical and shows the use of the LANGUAGE *language_term* and *top_n_by_rank* parameters.

```
USE Northwind;
SELECT FT_TBL.CategoryName
    ,FT_TBL.Description
    ,KEY_TBL.RANK
FROM dbo.Categories AS FT_TBL
    INNER JOIN FREETEXTTABLE(dbo.Categories, Description,
        'sweetest candy bread and dry meat',LANGUAGE 'English',2)
        AS KEY_TBL
        ON FT_TBL.CategoryID = KEY_TBL.[KEY];
GO
```

3.9.3. OPENDATASOURCE

Provides ad hoc connection information as part of a four-part object name without using a linked server name.

Syntax

```
OPENDATASOURCE ( provider_name, init_string )
```

Arguments

provider_name

Is the name registered as the PROGID of the OLE DB provider used to access the data source. *provider_name* is a char data type, with no default value.

init_string

Is the connection string passed to the **IDataInitialize** interface of the destination provider. The provider string syntax is based on keyword-value pairs separated by semicolons, such as: '*keyword1*=*value*; *keyword2*=*value*'.

For specific keyword-value pairs supported on the provider, see the Microsoft Data Access SDK. This documentation defines the basic syntax. The following table lists the most frequently used keywords in the *init_string* argument.

Keyword	OLE DB property	Valid values and description
Data Source	DBPROP_INIT_DATASOURCE	Name of the data source to connect to. Different providers interpret this in different ways. For SQL Server Native Client OLE DB provider, this indicates the name of the server. For Jet OLE DB provider, this indicates the full path of the .mdb file or .xls

		file.
Location	DBPROP_INIT_LOCATION	Location of the database to connect to.
Extended Properties	DBPROP_INIT_PROVIDERSTRING	The provider-specific connect-string.
Connect timeout	DBPROP_INIT_TIMEOUT	Time-out value after which the connection try fails.
User ID	DBPROP_AUTH_USERID	User ID to be used for the connection.
Password	DBPROP_AUTH_PASSWORD	Password to be used for the connection.
Catalog	DBPROP_INIT_CATALOG	The name of the initial or default catalog when connecting to the data source.
Integrated Security	DBPROP_AUTH_INTEGRATED	SSPI, to specify Windows Authentication

Remarks

OPENDATASOURCE can be used to access remote data from OLE DB data sources only when the DisallowAdhocAccess registry option is explicitly set to 0 for the specified provider, and the Ad Hoc Distributed Queries advanced configuration option is enabled. When these options are not set, the default behavior does not allow for ad hoc access.

The OPENDATASOURCE function can be used in the same Transact-SQL syntax locations as a linked-server name. Therefore, OPENDATASOURCE can be used as the first part of a four-part name that refers to a table or view name in a SELECT, INSERT, UPDATE, or DELETE statement, or to a remote stored procedure in an EXECUTE statement. When executing remote stored procedures, OPENDATASOURCE should refer to another instance of SQL Server. OPENDATASOURCE does not accept variables for its arguments.

Like the OPENROWSET function, OPENDATASOURCE should only reference OLE DB data sources that are accessed infrequently. Define a linked server for any data sources accessed more than several times. Neither OPENDATASOURCE nor OPENROWSET provide all the functionality of linked-server definitions, such as security management and the ability to query catalog information. All connection information, including passwords, must be provided every time that OPENDATASOURCE is called.

Important
Windows Authentication is much more secure than SQL Server Authentication. You should use Windows Authentication whenever possible. OPENDATASOURCE should not be used with explicit passwords in the

connection string.

The connection requirements for each provider are similar to the requirements for those parameters when creating linked servers. The details for many common providers are listed in the topic sp_addlinkedserver.

Permissions

Any user can execute OPENDATASOURCE. The permissions that are used to connect to the remote server are determined from the connection string.

Examples

The following example creates an ad hoc connection to the Payroll instance of SQL Server on server London, and queries the AdventureWorks2008R2.HumanResources.Employee table. (Use SQLNCLI and SQL Server will redirect to the latest version of SQL Server Native Client OLE DB Provider.)

```
SELECT *
FROM OPENDATASOURCE('SQLNCLI',
    'Data Source=London\Payroll;Integrated Security=SSPI')
    .AdventureWorks2008R2.HumanResources.Employee
```

The following example creates an ad hoc connection to an Excel spreadsheet in the 1997 - 2003 format.

```
SELECT * FROM OPENDATASOURCE('Microsoft.Jet.OLEDB.4.0',
'Data Source=C:\DataFolder\Documents\TestExcel.xls;Extended Properties=EXCEL
5.0')...[Sheet1$] ;
```

3.9.4. OPENQUERY

Executes the specified pass-through query on the specified linked server. This server is an OLE DB data source. OPENQUERY can be referenced in the FROM clause of a query as if it were a table name. OPENQUERY can also be referenced as the target table of an INSERT, UPDATE, or DELETE statement. This is subject to the capabilities of the OLE DB provider. Although the query may return multiple result sets, OPENQUERY returns only the first one.

Syntax

```
OPENQUERY ( linked_server ,'query' )
```

Arguments

linked_server

Is an identifier representing the name of the linked server.

'*query*'

Is the query string executed in the linked server. The maximum length of the string is 8 KB.

Remarks

OPENQUERY does not accept variables for its arguments.

OPENQUERY cannot be used to execute extended stored procedures on a linked server. However, an extended stored procedure can be executed on a linked server by using a four-part name. For example:

```
EXEC SeattleSales.master.dbo.xp_msver
```

Permissions

Any user can execute OPENQUERY. The permissions that are used to connect to the remote server are obtained from the settings defined for the linked server.

Examples

A. Executing a SELECT pass-through query

The following example creates a linked server named OracleSvr against an Oracle database by using the Microsoft OLE DB Provider for Oracle. Then, this example uses a pass-through SELECT query against this linked server.

Note
This example assumes that an Oracle database alias called ORCLDB has been created.

```
EXEC sp_addlinkedserver 'OracleSvr',
   'Oracle 7.3',
   'MSDAORA',
   'ORCLDB'
GO
SELECT *
FROM OPENQUERY(OracleSvr, 'SELECT name, id FROM joe.titles')
GO
```

B. Executing an UPDATE pass-through query

The following example uses a pass-through UPDATE query against the linked server created in example A.

```
UPDATE OPENQUERY (OracleSvr, 'SELECT name FROM joe.titles WHERE id = 101')
SET name = 'ADifferentName';
```

C. Executing an INSERT pass-through query

The following example uses a pass-through INSERT query against the linked server created in example A.

```
INSERT OPENQUERY (OracleSvr, 'SELECT name FROM joe.titles')
VALUES ('NewTitle');
```

D. Executing a DELETE pass-through query

The following example uses a pass-through DELETE query to delete the row inserted in example C.

```
DELETE OPENQUERY (OracleSvr, 'SELECT name FROM joe.titles WHERE name = ''NewT
itle''');
```

3.9.5. OPENROWSET

Includes all connection information that is required to access remote data from an OLE DB data source. This method is an alternative to accessing tables in a linked server and is a one-time, ad hoc method of connecting and accessing remote data by using OLE DB. For more frequent references to OLE DB data sources, use linked servers instead. For more information, see Linking Servers. The OPENROWSET function can be referenced in the FROM clause of a query as if it were a table name. The OPENROWSET function can also be referenced as the target table of an INSERT, UPDATE, or DELETE statement, subject to the capabilities of the OLE DB provider. Although the query might return multiple result sets, OPENROWSET returns only the first one.

OPENROWSET also supports bulk operations through a built-in BULK provider that enables data from a file to be read and returned as a rowset.

Syntax

```
OPENROWSET
( { 'provider_name' , { 'datasource' ; 'user_id' ; 'password'
   | 'provider_string' }
   , {   [ catalog. ] [ schema. ] object
       | 'query'
     }
   | BULK 'data_file' ,
       { FORMATFILE = 'format_file_path' [ <bulk_options> ]
       | SINGLE_BLOB | SINGLE_CLOB | SINGLE_NCLOB }
} )

<bulk_options> ::=
   [ , CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'code_page' } ]
   [ , ERRORFILE = 'file_name' ]
   [ , FIRSTROW = first_row ]
   [ , LASTROW = last_row ]
   [ , MAXERRORS = maximum_errors ]
   [ , ROWS_PER_BATCH = rows_per_batch ]
   [ , ORDER ( { column [ ASC | DESC ] } [ ,...n ] ) [ UNIQUE ]
```

Arguments

'*provider_name*'

Is a character string that represents the friendly name (or PROGID) of the OLE DB provider as specified in the registry. *provider_name* has no default value.

'*datasource*'

Is a string constant that corresponds to a particular OLE DB data source. *datasource* is the DBPROP_INIT_DATASOURCE property to be passed to the **IDBProperties** interface of the provider to initialize the provider. Typically, this string includes the name of the database file, the name of a database server, or a name that the provider understands to locate the database or databases.

'*user_id*'

Is a string constant that is the user name passed to the specified OLE DB provider. *user_id* specifies the security context for the connection and is passed in as the DBPROP_AUTH_USERID property to initialize the provider. *user_id* cannot be a Microsoft Windows login name.

'*password*'

Is a string constant that is the user password to be passed to the OLE DB provider. *password* is passed in as the DBPROP_AUTH_PASSWORD property when initializing the provider. *password* cannot be a Microsoft Windows password.

'*provider_string*'

Is a provider-specific connection string that is passed in as the DBPROP_INIT_PROVIDERSTRING property to initialize the OLE DB provider. *provider_string* typically encapsulates all the connection information required to initialize the provider. For a list of keywords that are recognized by the SQL Server Native Client OLE DB provider, see Initialization and Authorization Properties.

catalog

Is the name of the catalog or database in which the specified object resides.

schema

Is the name of the schema or object owner for the specified object.

object

Is the object name that uniquely identifies the object to work with.

'*query*'

Is a string constant sent to and executed by the provider. The local instance of SQL Server does not process this query, but processes query results returned by the provider, a pass-through query. Pass-through queries are useful when used on providers that do not make available their tabular data through table names, but only through a command language. Pass-through queries are supported on the remote server, as long as the query provider supports the OLE DB **Command** object and its mandatory interfaces. For more information, see SQL Server Native Client (OLE DB) Reference.

BULK

Uses the BULK rowset provider for OPENROWSET to read data from a file. In SQL Server, OPENROWSET can read from a data file without loading the data into a target table. This lets you use OPENROWSET with a simple SELECT statement.

The arguments of the BULK option allow for significant control over where to start and end reading data, how to deal with errors, and how data is interpreted. For example, you can specify that the data file be read as a single-row, single-column rowset of type varbinary, varchar, or nvarchar. The default behavior is described in the argument descriptions that follow.

For information about how to use the BULK option, see "Remarks," later in this topic. For information about the permissions that are required by the BULK option, see "Permissions," later in this topic.

Note
When used to import data with the full recovery model, OPENROWSET (BULK ...) does not optimize logging.

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

'*data_file*'

Is the full path of the data file whose data is to be copied into the target table.

FORMATFILE ='*format_file_path*'

Specifies the full path of a format file. SQL Server supports two types of format files: XML and non-XML.

A format file is required to define column types in the result set. The only exception is when SINGLE_CLOB, SINGLE_BLOB, or SINGLE_NCLOB is specified; in which case, the format file is not required.

For information about format files, see Using a Format File to Bulk Import Data.

< bulk_options >

Specifies one or more arguments for the BULK option.

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

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

Note
We recommend that you specify a collation name for each column in a format file.

CODEPAGE value	Description

ACP	Converts columns of char, varchar, or text data type from the ANSI/Microsoft Windows code page (ISO 1252) to the SQL Server code page.
OEM (default)	Converts columns of char, varchar, or text data type from the system OEM code page to the SQL Server code page.
RAW	No conversion occurs from one code page to another. This is the fastest option.
code_page	Indicates the source code page on which the character data in the data file is encoded; for example, 850. **Important** SQL Server does not support code page 65001 (UTF-8 encoding).

ERRORFILE ='*file_name*'

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

The error file is created at the start of the command execution. An error will be raised if the file already exists. Additionally, a control file that has the extension .ERROR.txt is created. This file references each row in the error file and provides error diagnostics. After the errors have been corrected, the data can be loaded.

FIRSTROW =*first_row*

Specifies the number of the first row to load. The default is 1. This indicates the first row in the specified data file. The row numbers are determined by counting the row terminators. FIRSTROW is 1-based.

LASTROW =*last_row*

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

MAXERRORS =*maximum_errors*

Specifies the maximum number of syntax errors or nonconforming rows, as defined in the format file, that can occur before OPENROWSET throws an exception. Until MAXERRORS is reached, OPENROWSET ignores each bad row, not loading it, and counts the bad row as one error.

The default for *maximum_errors* is 10.

Note
MAX_ERRORS does not apply to CHECK constraints, or to converting money and bigint data types.

ROWS_PER_BATCH =*rows_per_batch*

Specifies the approximate number of rows of data in the data file. This value should be of the same order as the actual number of rows.

OPENROWSET always imports a data file as a single batch. However, if you specify *rows_per_batch* with a value > 0, the query processor uses the value of *rows_per_batch* as a hint for allocating resources in the query plan.

By default, ROWS_PER_BATCH is unknown. Specifying ROWS_PER_BATCH = 0 is the same as omitting ROWS_PER_BATCH.

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

An optional hint that specifies how the data in the data file is sorted. By default, the bulk operation assumes the data file is unordered. Performance might improve if the order specified can be exploited by the query optimizer to generate a more efficient query plan. Examples for when specifying a sort can be beneficial include the following:

- Inserting rows into a table that has a clustered index, where the rowset data is sorted on the clustered index key.
- Joining the rowset with another table, where the sort and join columns match.
- Aggregating the rowset data by the sort columns.
- Using the rowset as a source table in the FROM clause of a query, where the sort and join columns match.

UNIQUE specifies that the data file does not have duplicate entries.

If the actual rows in the data file are not sorted according to the order that is specified, or if the UNIQUE hint is specified and duplicates keys are present, an error is returned.

Column aliases are required when ORDER is used. The column alias list must reference the derived table that is being accessed by the BULK clause. The column names that are specified in the ORDER clause refer to this column alias list. Large value types (varchar(max), nvarchar(max), varbinary(max), and xml) and large object (LOB) types (text, ntext, and image) columns cannot be specified.

SINGLE_BLOB

Returns the contents of *data_file* as a single-row, single-column rowset of type varbinary(max).

> **Important**
> We recommend that you import XML data only using the SINGLE_BLOB option, rather than SINGLE_CLOB and SINGLE_NCLOB, because only SINGLE_BLOB supports all Windows encoding conversions.

SINGLE_CLOB

By reading *data_file* as ASCII, returns the contents as a single-row, single-column rowset of type varchar(max), using the collation of the current database.

SINGLE_NCLOB

By reading *data_file* as UNICODE, returns the contents as a single-row, single-column rowset of type nvarchar(max), using the collation of the current database.

Remarks

OPENROWSET can be used to access remote data from OLE DB data sources only when the **DisallowAdhocAccess** registry option is explicitly set to 0 for the specified provider, and the Ad Hoc Distributed Queries advanced configuration option is enabled. When these options are not set, the default behavior does not allow for ad hoc access.

When accessing remote OLE DB data sources, the login identity of trusted connections is not automatically delegated from the server on which the client is connected to the server that is being queried. Authentication delegation must be configured. For more information, see Configuring Linked Servers for Delegation.

Catalog and schema names are required if the OLE DB provider supports multiple catalogs and schemas in the specified data source. Values for *catalog* and *schema* can be omitted when the OLE DB provider does not support them. If the provider supports only schema names, a two-part name of the form *schema.object* must be specified. If the provider supports only catalog names, a three-part name of the form *catalog.schema.object* must be specified. Three-part names must be specified for pass-through queries that use the SQL Server Native Client OLE DB provider. For more information, see Transact-SQL Syntax Conventions.

OPENROWSET does not accept variables for its arguments.

Using OPENROWSET with the BULK Option

The following Transact-SQL enhancements support the OPENROWSET(BULK...) function:

- A FROM clause that is used with SELECT can call OPENROWSET(BULK...) instead of a table name, with full SELECT functionality.

  OPENROWSET with the BULK option requires a correlation name, also known as a range variable or alias, in the FROM clause. Column aliases can be specified. If a column alias list is not specified, the format file must have column names. Specifying column aliases overrides the column names in the format file, such as:

  FROM OPENROWSET(BULK...) AS table_alias

  FROM OPENROWSET(BULK...) AS table_alias(column_alias,...n)

- A SELECT...FROM OPENROWSET(BULK...) statement queries the data in a file directly, without importing the data into a table. SELECT...FROM OPENROWSET(BULK...) statements can also list bulk-column aliases by using a format file to specify column names, and also data types.
- Using OPENROWSET(BULK...) as a source table in an INSERT or MERGE statement bulk imports data from a data file into a SQL Server table. For more information, see Importing Bulk Data by Using BULK INSERT or OPENROWSET(BULK...) .
- When the OPENROWSET BULK option is used with an INSERT statement, the BULK clause supports table hints. In addition to the regular table hints, such as TABLOCK, the BULK clause can accept the following specialized table hints: IGNORE_CONSTRAINTS (ignores only the CHECK and FOREIGN

KEY constraints), IGNORE_TRIGGERS, KEEPDEFAULTS, and KEEPIDENTITY. For more information, see Table Hints.

For information about how to use INSERT...SELECT * FROM OPENROWSET(BULK...) statements, see Importing and Exporting Bulk Data. For information about when row-insert operations that are performed by bulk import are logged in the transaction log, see Prerequisites for Minimal Logging in Bulk Import.

Note

When you use OPENROWSET, it is important to understand how SQL Server handles impersonation. For information about security considerations, see Importing Bulk Data by Using BULK INSERT or OPENROWSET(BULK...).

Bulk Importing SQLCHAR, SQLNCHAR or SQLBINARY Data

OPENROWSET(BULK...) assumes that, if not specified, the maximum length of SQLCHAR, SQLNCHAR or SQLBINARY data does not exceed 8000 bytes. If the data being imported is in a LOB data field that contains any varchar(max), nvarchar(max), or varbinary(max) objects that exceed 8000 bytes, you must use an XML format file that defines the maximum length for the data field. To specify the maximum length, edit the format file and declare the MAX_LENGTH attribute. For more information, see Schema Syntax for XML Format Files.

Note

An automatically generated format file does not specify the length or maximum length for a LOB field. However, you can edit a format file and specify the length or maximum length manually.

Bulk Exporting or Importing SQLXML Documents

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

Data type	Effect
SQLCHAR or SQLVARYCHAR	The data is sent in the client code page or in the code page implied by the collation).
SQLNCHAR or SQLNVARCHAR	The data is sent as Unicode.
SQLBINARY or SQLVARYBIN	The data is sent without any conversion.

Permissions

OPENROWSET permissions are determined by the permissions of the user name that is being passed to the OLE DB provider. To use the BULK option requires ADMINISTER BULK OPERATIONS permission.

Examples

A. Using OPENROWSET with SELECT and the SQL Server Native Client OLE DB Provider

The following example uses the SQL Server Native Client OLE DB provider to access the HumanResources.Department table in the AdventureWorks2008R2 database on the remote server Seattle1. (Use SQLNCLI and SQL Server will redirect to the latest version of SQL Server Native Client OLE DB Provider.) A SELECT statement is used to define the row set returned. The provider string contains the Server and Trusted_Connection keywords. These keywords are recognized by the SQL Server Native Client OLE DB provider.

```
SELECT a.*
FROM OPENROWSET('SQLNCLI', 'Server=Seattle1;Trusted_Connection=yes;',
     'SELECT GroupName, Name, DepartmentID
      FROM AdventureWorks2008R2.HumanResources.Department
      ORDER BY GroupName, Name') AS a;
```

B. Using the Microsoft OLE DB Provider for Jet

The following example accesses the Customers table in the Microsoft Access Northwind database through the Microsoft OLE DB Provider for Jet.

Note

This example assumes that Access is installed. To run this example, you must install the Northwind database. For information about how to install the Northwind database, see Downloading Northwind and pubs Sample Databases.

```
SELECT CustomerID, CompanyName
   FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
      'C:\Program Files\Microsoft Office\OFFICE11\SAMPLES\Northwind.mdb';
      'admin';'',Customers);
GO
```

C. Using OPENROWSET and another table in an INNER JOIN

The following example selects all data from the Customers table from the local instance of SQL Server Northwind database and from the Orders table from the Access Northwind database stored on the same computer.

Note

This example assumes that Access is installed. To run this example, you must install the Northwind database. For information about how to install the Northwind database, see Downloading Northwind and pubs Sample Databases.

```
USE Northwind  ;
GO
SELECT c.*, o.*
FROM Northwind.dbo.Customers AS c
```

```
   INNER JOIN OPENROWSET('Microsoft.Jet.OLEDB.4.0',
   'C:\Program Files\Microsoft Office\OFFICE11\SAMPLES\Northwind.mdb';'admin'
;'', Orders)
   AS o
   ON c.CustomerID = o.CustomerID ;
GO
```

D. Using OPENROWSET to bulk insert file data into a varbinary(max) column

The following example creates a small table for demonstration purposes, and inserts file data from a file named Text1.txt located in the C: root directory into a varbinary(max) column.

```
USE AdventureWorks2008R2;
GO
CREATE TABLE myTable(FileName nvarchar(60),
  FileType nvarchar(60), Document varbinary(max));
GO

INSERT INTO myTable(FileName, FileType, Document)
   SELECT 'Text1.txt' AS FileName,
      '.txt' AS FileType,
      * FROM OPENROWSET(BULK N'C:\Text1.txt', SINGLE_BLOB) AS Document;
GO
```

E. Using the OPENROWSET BULK provider with a format file to retrieve rows from a text file

The following example uses a format file to retrieve rows from a tab-delimited text file, values.txt that contains the following data:

```
1     Data Item 1
2     Data Item 2
3     Data Item 3
```

The format file, values.fmt, describes the columns in values.txt:

```
9.0
2
1  SQLCHAR  0  10 "\t"        1  ID                SQL_Latin1_General_Cp437_B
IN
2  SQLCHAR  0  40 "\r\n"      2  Description        SQL_Latin1_General_Cp437_
BIN
```

This is the query that retrieves that data:

```
SELECT a.* FROM OPENROWSET( BULK 'c:\test\values.txt',
   FORMATFILE = 'c:\test\values.fmt') AS a;
```

3.9.6. OPENXML

OPENXML provides a rowset view over an XML document. Because OPENXML is a rowset provider, OPENXML can be used in Transact-SQL statements in which rowset providers such as a table, view, or the OPENROWSET function can appear.

Syntax

```
OPENXML( idoc int [ in] , rowpattern nvarchar [ in ] , [ flags byte [ in ] ]
)
[ WITH ( SchemaDeclaration | TableName ) ]
```

Arguments

idoc

Is the document handle of the internal representation of an XML document. The internal representation of an XML document is created by calling **sp_xml_preparedocument**.

rowpattern

Is the XPath pattern used to identify the nodes (in the XML document whose handle is passed in the *idoc* parameter) to be processed as rows.

flags

Indicates the mapping that should be used between the XML data and the relational rowset, and how the spill-over column should be filled. *flags* is an optional input parameter, and can be one of the following values.

Byte value	Description
0	Defaults to **attribute-centric** mapping.
1	Use the **attribute-centric** mapping. Can be combined with XML_ELEMENTS. In this case, **attribute-centric** mapping is applied first, and then **element-centric** mapping is applied for all columns that are not yet dealt with.
2	Use the **element-centric** mapping. Can be combined with XML_ATTRIBUTES. In this case, **attribute-centric** mapping is applied first, and then **element-centric** mapping is applied for all columns not yet dealt with.
8	Can be combined (logical OR) with XML_ATTRIBUTES or XML_ELEMENTS. In the context of retrieval, this flag indicates that the consumed data should not be copied to the overflow property **@mp:xmltext**.

SchemaDeclaration

Is the schema definition of the form: *ColName ColType* [*ColPattern* | *MetaProperty*] [**,** *ColNameColType* [*ColPattern* | *MetaProperty*]...]

ColName

Is the column name in the rowset.

ColType

Is the SQL Server data type of the column in the rowset. If the column types differ from the underlying xml data type of the attribute, type coercion occurs.

ColPattern

Is an optional, general XPath pattern that describes how the XML nodes should be mapped to the columns. If *ColPattern* is not specified, the default mapping (**attribute-centric** or **element-centric** mapping as specified by *flags*) takes place.

The XPath pattern specified as *ColPattern* is used to specify the special nature of the mapping (in the case of **attribute-centric** and **element-centric** mapping) that overwrites or enhances the default mapping indicated by *flags*.

The general XPath pattern specified as *ColPattern* also supports the metaproperties.

MetaProperty

Is one of the metaproperties provided by OPENXML. If *MetaProperty* is specified, the column contains information provided by the metaproperty. The metaproperties allow you to extract information (such as relative position and namespace information) about XML nodes. This provides more information than is visible in the textual representation.

TableName

Is the table name that can be given (instead of *SchemaDeclaration*) if a table with the desired schema already exists and no column patterns are required.

Remarks

The WITH clause provides a rowset format (and additional mapping information as required) by using either *SchemaDeclaration* or specifying an existing *TableName*. If the optional WITH clause is not specified, the results are returned in an **edge** table format. Edge tables represent the fine-grained XML document structure (such as element/attribute names, the document hierarchy, the namespaces, PIs, and son on) in a single table.

The following table describes the structure of the **edge** table.

Column name	Data type	Description
id	bigint	Is the unique ID of the document node. The root element has an ID value 0. The negative ID values are reserved.

parentid	bigint	Identifies the parent of the node. The parent identified by this ID is not necessarily the parent element, but it depends on the NodeType of the node whose parent is identified by this ID. For example, if the node is a text node, the parent of it may be an attribute node. If the node is at the top level in the XML document, its **ParentID** is NULL.
nodetype	int	Identifies the node type. Is an integer that corresponds to the XML DOM node type numbering. The node types are: 1 = Element node 2 = Attribute node 3 = Text node
localname	nvarchar	Gives the local name of the element or attribute. Is NULL if the DOM object does not have a name.
prefix	nvarchar	Is the namespace prefix of the node name.
namespaceuri	nvarchar	Is the namespace URI of the node. If the value is NULL, no namespace is present.
datatype	nvarchar	Is the actual data type of the element or attribute row, otherwise is NULL. The data type is inferred from the inline DTD or from the inline schema.
prev	bigint	Is the XML ID of the previous sibling element. Is NULL if there is no direct previous sibling.
text	ntext	Contains the attribute value or the element content in text form (or is NULL if the **edge** table entry does not require a value).

Examples

A. Using a simple SELECT statement with OPENXML

The following example creates an internal representation of the XML image by using sp_xml_preparedocument. A SELECT statement that uses an OPENXML rowset provider is then executed against the internal representation of the XML document.

The *flag* value is set to 1. This indicates **attribute-centric** mapping. Therefore, the XML attributes map to the columns in the rowset. The *rowpattern* specified as /ROOT/Customer identifies the <Customers> nodes to be processed.

The optional *ColPattern* (column pattern) parameter is not specified because the column name matches the XML attribute names.

The OPENXML rowset provider creates a two-column rowset (CustomerID and ContactName) from which the SELECT statement retrieves the necessary columns (in this case, all the columns).

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
   <Order CustomerID="VINET" EmployeeID="5" OrderDate="1996-07-04T00:00:00">
      <OrderDetail OrderID="10248" ProductID="11" Quantity="12"/>
      <OrderDetail OrderID="10248" ProductID="42" Quantity="10"/>
   </Order>
</Customer>
<Customer CustomerID="LILAS" ContactName="Carlos Gonzlez">
   <Order CustomerID="LILAS" EmployeeID="3" OrderDate="1996-08-16T00:00:00">
      <OrderDetail OrderID="10283" ProductID="72" Quantity="3"/>
   </Order>
</Customer>
</ROOT>'
--Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
-- Execute a SELECT statement that uses the OPENXML rowset provider.
SELECT    *
FROM       OPENXML (@idoc, '/ROOT/Customer',1)
            WITH (CustomerID  varchar(10),
                  ContactName varchar(20))
```

Here is the result set.

```
CustomerID ContactName
---------- --------------------
VINET      Paul Henriot
LILAS      Carlos Gonzlez
```

If the same SELECT statement is executed with *flags* set to 2, indicating **element-centric** mapping, the values of CustomerID and ContactName for both of the customers in the XML document are returned as NULL, because the <Customers> elements do not have any subelements.

Here is the result set.

```
CustomerID ContactName
---------- -----------
NULL       NULL
NULL       NULL
```

B. Specifying ColPattern for mapping between columns and the XML attributes

The following query returns customer ID, order date, product ID and quantity attributes from the XML document. The *rowpattern* identifies the <OrderDetails> elements. ProductID and Quantity are the

attributes of the <OrderDetails> element. However, OrderID, CustomerID, and OrderDate are the attributes of the parent element (<Orders>).

The optional *ColPattern* is specified. This indicates the following:

- The OrderID, CustomerID, and OrderDate in the rowset map to the attributes of the parent of the nodes identified by *rowpattern* in the XML document.
- The ProdID column in the rowset maps to the ProductID attribute, and the Qty column in the rowset maps to the Quantity attribute of the nodes identified in *rowpattern*.

Although the **element-centric** mapping is specified by the *flags* parameter, the mapping specified in *ColPattern* overwrites this mapping.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
   <Order OrderID="10248" CustomerID="VINET" EmployeeID="5"
          OrderDate="1996-07-04T00:00:00">
      <OrderDetail ProductID="11" Quantity="12"/>
      <OrderDetail ProductID="42" Quantity="10"/>
   </Order>
</Customer>
<Customer CustomerID="LILAS" ContactName="Carlos Gonzlez">
   <Order OrderID="10283" CustomerID="LILAS" EmployeeID="3"
          OrderDate="1996-08-16T00:00:00">
      <OrderDetail ProductID="72" Quantity="3"/>
   </Order>
</Customer>
</ROOT>'
--Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
-- SELECT stmt using OPENXML rowset provider
SELECT *
FROM   OPENXML (@idoc, '/ROOT/Customer/Order/OrderDetail',2)
         WITH (OrderID       int          '../@OrderID',
               CustomerID  varchar(10) '../@CustomerID',
               OrderDate   datetime    '../@OrderDate',
               ProdID      int         '@ProductID',
               Qty         int         '@Quantity')
```

Here is the result set.

```
OrderID CustomerID           OrderDate                ProdID    Qty
-----------------------------------------------------------------------
10248      VINET       1996-07-04 00:00:00.000   11      12
10248      VINET       1996-07-04 00:00:00.000   42      10
10283      LILAS       1996-08-16 00:00:00.000   72      3
```

C. Obtaining results in an edge table format

The sample XML document in the following example consists of <Customers>, <Orders>, and <Order_0020_Details> elements. First, **sp_xml_preparedocument** is called to obtain a document handle. This document handle is passed to OPENXML.

In the OPENXML statement, the *rowpattern* (/ROOT/Customers) identifies the <Customers> nodes to process. Because the WITH clause is not provided, OPENXML returns the rowset in an **edge** table format.

Finally the SELECT statement retrieves all the columns in the **edge** table.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customers CustomerID="VINET" ContactName="Paul Henriot">
   <Orders CustomerID="VINET" EmployeeID="5" OrderDate=
           "1996-07-04T00:00:00">
      <Order_x0020_Details OrderID="10248" ProductID="11" Quantity="12"/>
      <Order_x0020_Details OrderID="10248" ProductID="42" Quantity="10"/>
   </Orders>
</Customers>
<Customers CustomerID="LILAS" ContactName="Carlos Gonzlez">
   <Orders CustomerID="LILAS" EmployeeID="3" OrderDate=
           "1996-08-16T00:00:00">
      <Order_x0020_Details OrderID="10283" ProductID="72" Quantity="3"/>
   </Orders>
</Customers>
</ROOT>'
--Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
-- SELECT statement that uses the OPENXML rowset provider.
SELECT    *
FROM       OPENXML (@idoc, '/ROOT/Customers')
EXEC sp_xml_removedocument @idoc
```

3.10. Security Functions

The following functions return information that is useful in managing security.

CURRENT_USER	Returns the name of the current user. This function is equivalent to USER_NAME().
ORIGINAL_LOGIN	Returns the name of the login that connected to the instance of SQL Server. You can use this function to return the identity of the original login in sessions in which there are many explicit or implicit context switches. For more information about context switching, see Understanding Context Switching.

PERMISSIONS	
SESSION_USER	SESSION_USER returns the user name of the current context in the current database.
SYSTEM_USER	Allows a system-supplied value for the current login to be inserted into a table when no default value is specified.

3.10.1. PERMISSIONS

Returns a value containing a bitmap that indicates the statement, object, or column permissions of the current user.

Important This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Use fn_my_permissions and Has_Perms_By_Name instead. Continued use of the PERMISSIONS function may result in slower performance.

Syntax

```
PERMISSIONS ( [ objectid [ , 'column' ] ] )
```

Arguments

objectid

Is the ID of a securable. If *objectid* is not specified, the bitmap value contains statement permissions for the current user; otherwise, the bitmap contains permissions on the securable for the current user. The securable specified must be in the current database. Use the OBJECT_ID function to determine the *objectid* value.

'*column*'

Is the optional name of a column for which permission information is being returned. The column must be a valid column name in the table specified by *objectid*.

Return Types

int

Remarks

PERMISSIONS can be used to determine whether the current user has the permissions required to execute a statement or to GRANT a permission to another user.

The permissions information returned is a 32-bit bitmap.

The lower 16 bits reflect permissions granted to the user, and also permissions that are applied to Windows groups or and fixed server roles of which the current user is a member. For example, a returned value of 66 (hex value 0x42), when no *objectid* is specified, indicates that the user has permission to execute the CREATE TABLE (decimal value 2) and BACKUP DATABASE (decimal value 64) statements.

The upper 16 bits reflect the permissions that the user can GRANT to other users. The upper 16 bits are interpreted exactly as those for the lower 16 bits described in the following tables, except they are shifted to the left by 16 bits (multiplied by 65536). For example, 0x8 (decimal value 8) is the bit that indicates INSERT permission when an *objectid* is specified. Whereas, 0x80000 (decimal value 524288) indicates the ability to GRANT INSERT permission, because 524288 = 8 x 65536.

Because of membership in roles, a user that does not have permission to execute a statement may still be able to grant that permission to another user.

The following table shows the bits that are used for statement permissions (*objectid* is not specified).

Bit (dec)	Bit (hex)	Statement permission
1	0x1	CREATE DATABASE (master database only)
2	0x2	CREATE TABLE
4	0x4	CREATE PROCEDURE
8	0x8	CREATE VIEW
16	0x10	CREATE RULE
32	0x20	CREATE DEFAULT
64	0x40	BACKUP DATABASE
128	0x80	BACKUP LOG
256	0x100	Reserved

The following table shows the bits used for object permissions that are returned when only *objectid* is specified.

Bit (dec)	Bit (hex)	Statement permission
1	0x1	SELECT ALL
2	0x2	UPDATE ALL
4	0x4	REFERENCES ALL

8	0x8	INSERT
16	0x10	DELETE
32	0x20	EXECUTE (procedures only)
4096	0x1000	SELECT ANY (at least one column)
8192	0x2000	UPDATE ANY
16384	0x4000	REFERENCES ANY

The following table shows the bits used for column-level object permissions that are returned when both *objectid* and column are specified.

Bit (dec)	Bit (hex)	Statement permission
1	0x1	SELECT
2	0x2	UPDATE
4	0x4	REFERENCES

A NULL is returned when a specified parameter is NULL or not valid (for example, an *objectid* or column that does not exist). The bit values for permissions that do not apply (for example EXECUTE permission, bit 0x20, for a table) are undefined.

Use the bitwise AND (&) operator to determine each bit set in the bitmap that is returned by the PERMISSIONS function.

The sp_helprotect system stored procedure can also be used to return a list of permissions for a user in the current database.

Examples

A. Using the PERMISSIONS function with statement permissions

The following example determines whether the current user can execute the CREATE TABLE statement.

```
IF PERMISSIONS()&2=2
   CREATE TABLE test_table (col1 INT)
ELSE
   PRINT 'ERROR: The current user cannot create a table.';
```

B. Using the PERMISSIONS function with object permissions

The following example determines whether the current user can insert a row of data into the Address table in the AdventureWorks2008R2 database.

```
IF PERMISSIONS(OBJECT_ID('AdventureWorks2008R2.Person.Address','U'))&8=8
   PRINT 'The current user can insert data into Person.Address.'
ELSE
   PRINT 'ERROR: The current user cannot insert data into Person.Address.';
```

C. Using the PERMISSIONS function with grantable permissions

The following example determines whether the current user can grant the INSERT permission on the Address table in the AdventureWorks2008R2 database to another user.

```
IF PERMISSIONS(OBJECT_ID('AdventureWorks2008R2.Person.Address','U'))&0x80000=
0x80000
   PRINT 'INSERT on Person.Address is grantable.'
ELSE
   PRINT 'You may not GRANT INSERT permissions on Person.Address.';
```

3.11. String Functions

The following scalar functions perform an operation on a string input value and return a string or numeric value:

ASCII	Returns the ASCII code value of the leftmost character of a character expression.
CHAR	Converts an int ASCII code to a character.
CHARINDEX	Searches *expression2* for *expression1* and returns its starting position if found. The search starts at *start_location*. Syntax: **CHARINDEX (expression1 ,expression2 [, start_location])**
DIFFERENCE	Returns an integer value that indicates the difference between the SOUNDEX values of two character expressions. Syntax: **DIFFERENCE (character_expression , character_expression)** **USE AdventureWorks2008R2;** **GO** **-- Returns a DIFFERENCE value of 4, the least possible differen ce.** **SELECT SOUNDEX('Green'), SOUNDEX('Greene'), DIFFERENCE('Green', 'Greene');** **GO** **-- Returns a DIFFERENCE value of 0, the highest possible differ**

	ence. SELECT SOUNDEX('Blotchet-Halls'), SOUNDEX('Greene'), DIFFERENCE ('Blotchet-Halls', 'Greene'); GO Here is the result set. ----- ----- ----------- G650 G650 4 (1 row(s) affected) ----- ----- ----------- B432 G650 0 (1 row(s) affected)
LEFT	Returns the left part of a character string with the specified number of characters. Syntax: **LEFT (character_expression , integer_expression)**
LEN	Returns the number of characters of the specified string expression, excluding trailing blanks.
LOWER	Returns a character expression after converting uppercase character data to lowercase.
LTRIM	Returns a character expression after it removes leading blanks.
NCHAR	Returns the Unicode character with the specified integer code, as defined by the Unicode standard.
PATINDEX	Returns the starting position of the first occurrence of a pattern in a specified expression, or zeros if the pattern is not found, on all valid text and character data types. For more information, see Pattern Matching in Search Conditions. Syntax: **PATINDEX ('%pattern%' , expression)** '*pattern*' Is a literal string. Wildcard characters can be used; however, the % character must come before and follow *pattern* (except when you search for first or last characters). *pattern* is an expression of the character string data type category.
QUOTENAME	Returns a Unicode string with the delimiters added to make the input string a valid SQL Server delimited identifier. Syntax: **QUOTENAME ('character_string' [, 'quote_character'])**
REPLACE	Replaces all occurrences of a specified string value with another string value. Syntax: **REPLACE (string_expression , string_pattern , string_replacement)**

REPLICATE	Repeats a string value a specified number of times. Syntax: **REPLICATE (string_expression ,integer_expression)**
REVERSE	Returns the reverse of a string value.
RIGHT	Returns the right part of a character string with the specified number of characters. Syntax: **RIGHT (character_expression , integer_expression)**
RTRIM	Returns a character string after truncating all trailing blanks.
SOUNDEX	
SPACE	Returns a string of repeated spaces.
STR	Returns character data converted from numeric data. Syntax: **STR (float_expression [, length [, decimal]])**
STUFF	The STUFF function inserts a string into another string. It deletes a specified length of characters in the first string at the start position and then inserts the second string into the first string at the start position. Syntax: **STUFF (character_expression , start , length ,character_expression)** **SELECT STUFF('abcdef', 2, 3, 'ijklmn');** GO --------- aijklmnef (1 row(s) affected)
SUBSTRING	Returns part of a character, binary, text, or image expression. For more information about the valid SQL Server data types that can be used with this function, see Data Types. Syntax: **SUBSTRING (value_expression , start_expression , length_expression)**
UNICODE	Returns the integer value, as defined by the Unicode standard, for the first character of the input expression.
UPPER	Returns a character expression with lowercase character data converted to uppercase.

3.11.1. SOUNDEX

Returns a four-character (SOUNDEX) code to evaluate the similarity of two strings.

Syntax

```
SOUNDEX ( character_expression )
```

Arguments

character_expression

Is an alphanumeric expression of character data. *character_expression* can be a constant, variable, or column.

Return Types

varchar

Remarks

SOUNDEX converts an alphanumeric string to a four-character code to find similar-sounding words or names. The first character of the code is the first character of *character_expression* and the second through fourth characters of the code are numbers. Vowels in *character_expression* are ignored unless they are the first letter of the string. String functions can be nested.

Examples

The following example shows the SOUNDEX function and the related DIFFERENCE function. In the first example, the standard SOUNDEX values are returned for all consonants. Returning the SOUNDEX for Smith and Smythe returns the same SOUNDEX result because all vowels, the letter y, doubled letters, and the letter h, are not included.

```
-- Using SOUNDEX
SELECT SOUNDEX ('Smith'), SOUNDEX ('Smythe');
```

Here is the result set.

```
----- -----
S530  S530

(1 row(s) affected)
```

The DIFFERENCE function compares the difference of the SOUNDEX pattern results. The following example shows two strings that differ only in vowels. The difference returned is 4, the lowest possible difference.

```
-- Using DIFFERENCE
SELECT DIFFERENCE('Smithers', 'Smythers');
GO
```

Here is the result set.

```
-----------
```

```
4

(1 row(s) affected)
```

In the following example, the strings differ in consonants; therefore, the difference returned is 2, the greater difference.

```
SELECT DIFFERENCE('Anothers', 'Brothers');
GO
```

Here is the result set.

```
-----------
2

(1 row(s) affected)
```

3.12. System Functions

The following functions perform operations on and return information about values, objects, and settings in SQL Server.

@@ERROR	Returns the error number for the last Transact-SQL statement executed. Returns 0 if the previous Transact-SQL statement encountered no errors.
@@IDENTITY	Is a system function that returns the last-inserted identity value.
@@PACK_RECEIVED	Returns the number of input packets read from the network by SQL Server since it was last started.
@@ROWCOUNT	Returns the number of rows affected by the last statement. If the number of rows is more than 2 billion, use ROWCOUNT_BIG.
@@TRANCOUNT	Returns the number of BEGIN TRANSACTION statements that have occurred on the current connection.
CAST and CONVERT	
CONNECTIONPROPERTY	
ERROR_LINE	Returns the line number at which an error occurred that caused the CATCH block of a TRY…CATCH construct to be run.
ERROR_MESSAGE	Returns the message text of the error that caused the CATCH block of a TRY…CATCH construct to be run.

ERROR_NUMBER	Returns the error number of the error that caused the CATCH block of a TRY…CATCH construct to be run.
ERROR_PROCEDURE	Returns the name of the stored procedure or trigger where an error occurred that caused the CATCH block of a TRY…CATCH construct to be run.
ERROR_SEVERITY	Returns the severity of the error that caused the CATCH block of a TRY…CATCH construct to be run.
ERROR_STATE	Returns the state number of the error that caused the CATCH block of a TRY…CATCH construct to be run.
FORMATMESSAGE	
HOST_ID	Returns the workstation identification number. The workstation identification number is the process ID (PID) of the application on the client computer that is connecting to SQL Server.
HOST_NAME	Returns the workstation name.
ISNULL	
ISNUMERIC	Determines whether an expression is a valid numeric type.
NEWID	Creates a unique value of type uniqueidentifier.
NEWSEQUENTIALID	
PathName	
XACT_STATE	

3.12.1. CAST and CONVERT

Converts an expression of one data type to another in SQL Server 2008 R2.

Syntax

```
Syntax for CAST:
CAST ( expression AS data_type [ ( length ) ] )
```

Syntax for CONVERT:

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

Arguments

expression

Is any valid expression.

data_type

Is the target data type. This includes xml, bigint, and sql_variant. Alias data types cannot be used. For more information about available data types, see Data Types.

length

Is an optional integer that specifies the length of the target data type. The default value is 30.

style

Is an integer expression that specifies how the CONVERT function is to translate *expression*. If style is NULL, NULL is returned. The range is determined by *data_type*. For more information, see the Remarks section.

Return Types

Returns *expression* translated to *data_type*.

Remarks

Date and Time Styles

When *expression* is a date or time data type, *style* can be one of the values shown in the following table. Other values are processed as 0. SQL Server supports the date format in Arabic style by using the Kuwaiti algorithm.

Without century (yy) ([1])	With century (yyyy)	Standard	Input/Output ([3])
-	**0** or **100** ([1, 2])	Default	mon dd yyyy hh:miAM (or PM)
1	**101**	U.S.	mm/dd/yyyy
2	**102**	ANSI	yy.mm.dd
3	**103**	British/French	dd/mm/yyyy
4	**104**	German	dd.mm.yy
5	**105**	Italian	dd-mm-yy

6	**106** [(1)]	-	dd mon yy
7	**107** [(1)]	-	Mon dd, yy
8	**108**	-	hh:mi:ss
-	**9** or **109** ([1, 2])	Default + milliseconds	mon dd yyyy hh:mi:ss:mmmAM (or PM)
10	**110**	USA	mm-dd-yy
11	**111**	JAPAN	yy/mm/dd
12	**112**	ISO	yymmdd yyyymmdd
-	**13** or **113** ([1, 2])	Europe default + milliseconds	dd mon yyyy hh:mi:ss:mmm(24h)
14	**114**	-	hh:mi:ss:mmm(24h)
-	**20** or **120** ([2])	ODBC canonical	yyyy-mm-dd hh:mi:ss(24h)
-	**21** or **121** ([2])	ODBC canonical (with milliseconds)	yyyy-mm-dd hh:mi:ss.mmm(24h)
-	**126** ([4])	ISO8601	yyyy-mm-ddThh:mi:ss.mmm (no spaces)
-	**127**([6, 7])	ISO8601 with time zone Z.	yyyy-mm-ddThh:mi:ss.mmmZ (no spaces)
-	**130** ([1, 2])	Hijri ([5])	dd mon yyyy hh:mi:ss:mmmAM
-	**131** ([2])	Hijri ([5])	dd/mm/yy hh:mi:ss:mmmAM

[1] These style values return nondeterministic results. Includes all (yy) (without century) styles and a subset of (yyyy) (with century) styles.

[2] The default values (*style* **0** or **100**, **9** or **109**, **13** or **113**, **20** or **120**, and **21** or **121**) always return the century (yyyy).

[3] Input when you convert to datetime; output when you convert to character data.

[4] Designed for XML use. For conversion from datetime or smalldatetime to character data, the output format is as described in the previous table.

[5] Hijri is a calendar system with several variations. SQL Server uses the Kuwaiti algorithm.

Important

By default, SQL Server interprets two-digit years based on a cutoff year of 2049. That is, the two-digit year 49 is interpreted as 2049 and the two-digit year 50 is interpreted as 1950. Many client applications, such as those based on Automation objects, use a cutoff year of 2030. SQL Server provides the two digit year cutoff configuration option that changes the cutoff year used by SQL Server and allows for the consistent treatment of dates. We recommend specifying four-digit years.

[6] Only supported when casting from character data to datetime or smalldatetime. When character data that represents only date or only time components is cast to the datetime or smalldatetime data types, the unspecified time component is set to 00:00:00.000, and the unspecified date component is set to 1900-01-01.

[7]The optional time zone indicator, Z, is used to make it easier to map XML datetime values that have time zone information to SQL Server datetime values that have no time zone. Z is the indicator for time zone UTC-0. Other time zones are indicated with HH:MM offset in the + or - direction. For example: 2006-12-12T23:45:12-08:00.

When you convert to character data from smalldatetime, the styles that include seconds or milliseconds show zeros in these positions. You can truncate unwanted date parts when you convert from datetime or smalldatetime values by using an appropriate char or varchar data type length.

When you convert to datetimeoffset from character data with a style that includes a time, a time zone offset is appended to the result.

float and real Styles

When *expression* is float or real, *style* can be one of the values shown in the following table. Other values are processed as 0.

Value	Output
0 (default)	A maximum of 6 digits. Use in scientific notation, when appropriate.
1	Always 8 digits. Always use in scientific notation.
2	Always 16 digits. Always use in scientific notation.
126, 128, 129	Included for legacy reasons and might be deprecated in a future release.

money and smallmoney Styles

When *expression* is money or smallmoney, *style* can be one of the values shown in the following table. Other values are processed as 0.

Value	Output
0 (default)	No commas every three digits to the left of the decimal point, and two digits to the right of the decimal point; for example, 4235.98.
1	Commas every three digits to the left of the decimal point, and two digits to the right of the decimal point; for example, 3,510.92.
2	No commas every three digits to the left of the decimal point, and four digits to the right of the decimal point; for example, 4235.9819.
126	Equivalent to style 2 when converting to char(n) or varchar(n)

xml Styles

When *expression* is xml*, style* can be one of the values shown in the following table. Other values are processed as 0.

Value	Output
0 (default)	Use default parsing behavior that discards insignificant white space and does not allow for an internal DTD subset. **Note** When you convert to the xml data type, SQL Server insignificant white space is handled differently than in XML 1.0. For more information, see Generating XML Instances.
1	Preserve insignificant white space. This style setting sets the default **xml:space** handling to behave the same as if **xml:space="preserve"** has been specified instead.
2	Enable limited internal DTD subset processing. If enabled, the server can use the following information that is provided in an internal DTD subset to perform nonvalidating parse operations. • Defaults for attributes are applied. • Internal entity references are resolved and expanded. • The DTD content model will be checked for syntactical correctness. The parser will ignore external DTD subsets. It also does not evaluate the XML declaration to see whether the standalone attribute is set **yes** or **no**, but instead parses the XML instance as if it is a stand-alone document.

3	Preserve insignificant white space and enable limited internal DTD subset processing.

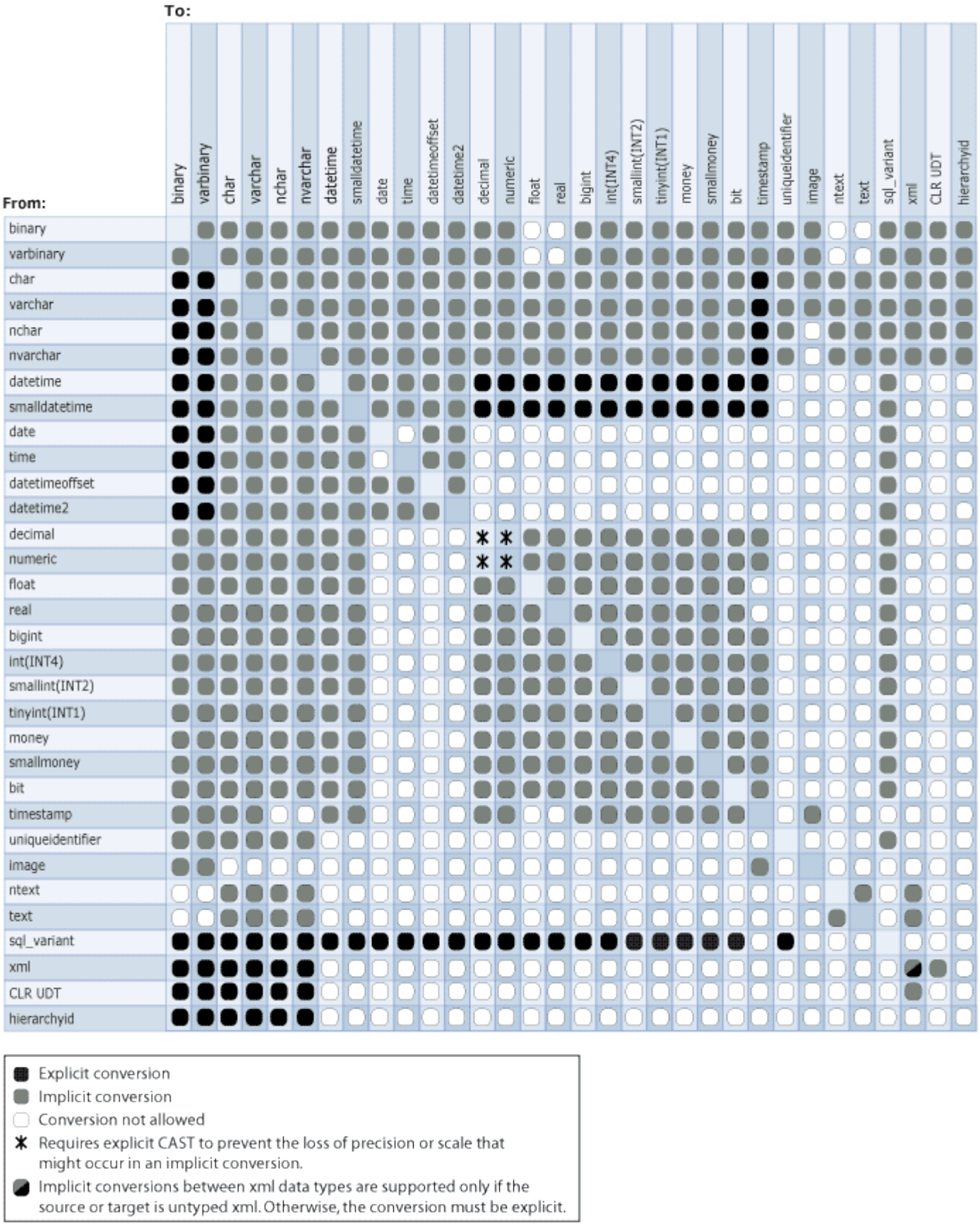
Binary Styles

When *expression* is binary(n), varbinary(n), char(n), or varchar(n), *style* can be one of the values shown in the following table. Style values that are not listed in the table return an error.

Value	Output
0 (default)	Translates ASCII characters to binary bytes or binary bytes to ASCII characters. Each character or byte is converted 1:1. If the *data_type* is a binary type, the characters 0x are added to the left of the result.
1, **2**	If the *data_type* is a binary type, the expression must be a character expression. The *expression* must be composed of an even number of hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, a, b, c, d, e, f). If the *style* is set to 1 the characters 0x must be the first two characters in the expression. If the expression contains an odd number of characters or if any of the characters are invalid an error is raised. If the length of the converted expression is greater than the length of the *data_type* the result will be right truncated. Fixed length *data_types* that are larger then the converted result will have zeros added to the right of the result. If the data_type is a character type, the expression must be a binary expression. Each binary character is converted into two hexadecimal characters. If the length of the converted expression is greater than the *data_type* length it will be right truncated. If the *data_type* is a fix sized character type and the length of the converted result is less than its length of the *data_type*; spaces are added to the right of the converted expression to maintain an even number of hexadecimal digits. The characters 0x will be added to the left of the converted result for *style* 1.

Implicit Conversions

Implicit conversions are those conversions that occur without specifying either the CAST or CONVERT function. Explicit conversions are those conversions that require the CAST or CONVERT function to be specified. The following illustration shows all explicit and implicit data type conversions that are allowed for SQL Server system-supplied data types. These include xml, bigint, and sql_variant. There is no implicit conversion on assignment from the sql_variant data type, but there is implicit conversion to sql_variant.



When you convert between datetimeoffset and the character types char, varchar, nchar, and nvarchar the converted time zone offset part should always be double digits for both HH and MM for example, -08:00.

Note

Because Unicode data always uses an even number of bytes, use caution when you convert binary or varbinary to or from Unicode supported data types. For example, the following conversion does not return a hexadecimal value of 41; it returns 4100: SELECT CAST(CAST(0x41 AS nvarchar) AS varbinary).

Large-Value Data Types

Large-value data types exhibit the same implicit and explicit conversion behavior as their smaller counterparts, specifically the varchar, nvarchar and varbinary data types. However, you should consider the following guidelines:

- Conversion from image to varbinary(max) and vice-versa is an implicit conversion, and so are conversions between text and varchar(max), and ntext and nvarchar(max).
- Conversion from large-value data types, such as varchar(max), to a smaller counterpart data type, such as varchar, is an implicit conversion, but truncation will occur if the large value is too big for the specified length of the smaller data type.
- Conversion from varchar, nvarchar, or varbinary to their corresponding large-value data types is performed implicitly.
- Conversion from the sql_variant data type to the large-value data types is an explicit conversion.
- Large-value data types cannot be converted to the sql_variant data type.

For information about how to convert Microsoft .NET Framework common language runtime (CLR) user-defined types, see Performing Operations on User-defined Types. For more information about how to convert from the xml data type, see Generating XML Instances.

xml Data Type

When you explicitly or implicitly cast the xml data type to a string or binary data type, the content of the xml data type is serialized based on a set of rules. For information about these rules, see Serialization of XML Data. For information on how to cast from XML to a CLR user-defined type, see Performing Operations on User-defined Types. For information about how to convert from other data types to the xml data type, see Generating XML Instances.

text and image Data Types

Automatic data type conversion is not supported for the text and image data types. You can explicitly convert text data to character data, and image data to binary or varbinary, but the maximum length is 8000 bytes. If you try an incorrect conversion such as trying to convert a character expression that includes letters to an int, SQL Server returns an error message.

Output Collation

When the output of CAST or CONVERT is a character string, and the input is a character string, the output has the same collation and collation label as the input. If the input is not a character string, the output has the default collation of the database, and a collation label of coercible-default. For more information, see Collation Precedence.

To assign a different collation to the output, apply the COLLATE clause to the result expression of the CAST or CONVERT function. For example:

```
SELECT CAST('abc' AS varchar(5)) COLLATE French_CS_AS
```

Truncating and Rounding Results

When you convert character or binary expressions (char, nchar, nvarchar, varchar, binary, or varbinary) to an expression of a different data type, data can be truncated, only partially displayed, or an error is returned because the result is too short to display. Conversions to char, varchar, nchar, nvarchar, binary, and varbinary are truncated, except for the conversions shown in the following table.

From data type	To data type	Result
int, smallint, or tinyint	char	*
	varchar	*
	nchar	E
	nvarchar	E
money, smallmoney, numeric, decimal, float, or real	char	E
	varchar	E
	nchar	E
	nvarchar	E

* = Result length too short to display. E = Error returned because result length is too short to display.

SQL Server guarantees that only roundtrip conversions, conversions that convert a data type from its original data type and back again, will yield the same values from version to version. The following example shows such a roundtrip conversion:

```
DECLARE @myval decimal (5, 2)
SET @myval = 193.57
SELECT CAST(CAST(@myval AS varbinary(20)) AS decimal(10,5))
-- Or, using CONVERT
SELECT CONVERT(decimal(10,5), CONVERT(varbinary(20), @myval))
```

Note

Do not try to construct binary values and then convert them to a data type of the numeric data type category. SQL Server does not guarantee that the result of a decimal or numeric data type conversion to binary will be the same between versions of SQL Server.

The following example shows a resulting expression that is too small to display.

```
USE AdventureWorks2008R2;
GO
SELECT p.FirstName, p.LastName, SUBSTRING(p.Title, 1, 25) AS Title, CAST(e.Si
ckLeaveHours AS char(1)) AS 'Sick Leave'
FROM HumanResources.Employee e
JOIN Person.Person p ON e.BusinessEntityID = p.BusinessEntityID
WHERE NOT e.BusinessEntityID >5;
```

Here is the result set.
FirstName LastName Title Sick Leave
--------- ------------ ---------- -----------
Ken Sanchez NULL *
Terri Duffy NULL *
Roberto Tamburello NULL *
Rob Walters NULL *
Gail Erickson Ms. *
(5 row(s) affected)

When you convert data types that differ in decimal places, sometimes the result value is truncated and at other times it is rounded. The following table shows the behavior.

From	To	Behavior
numeric	numeric	Round
numeric	int	Truncate
numeric	money	Round
money	int	Round
money	numeric	Round
float	int	Truncate
float	numeric	Round
float	datetime	Round
datetime	int	Round

For example, the result of the following conversion is 10:

```
SELECT CAST(10.6496 AS int)
```

When you convert data types in which the target data type has fewer decimal places than the source data type, the value is rounded. For example, the result of the following conversion is $10.3497:

```
SELECT CAST(10.3496847 AS money)
```

SQL Server returns an error message when nonnumeric char, nchar, varchar, or nvarchar data is converted to int, float, numeric, or decimal. SQL Server also returns an error when an empty string (" ") is converted to numeric or decimal.

Certain datetime Conversions Are Nondeterministic in SQL Server 2005 and Later Versions

In SQL Server 2000, string to date and time conversions are marked as deterministic. However, this is not true for the styles listed in the following table. For these styles, the conversions depend on the language settings. SQL Server 2005 and later versions mark these conversions as nondeterministic.

The following table lists the styles for which the string-to-datetime conversion is nondeterministic.

All styles below 100[1]	106
107	109
113	130

[1] With the exception of styles 20 and 21

Examples

A. Using both CAST and CONVERT

Each example retrieves the name of the product for those products that have a 3 in the first digit of their list price and converts their ListPrice to int.

```
-- Use CAST
USE AdventureWorks2008R2;
GO
SELECT SUBSTRING(Name, 1, 30) AS ProductName, ListPrice
FROM Production.Product
WHERE CAST(ListPrice AS int) LIKE '3%';
GO

-- Use CONVERT.
USE AdventureWorks2008R2;
GO
SELECT SUBSTRING(Name, 1, 30) AS ProductName, ListPrice
FROM Production.Product
WHERE CONVERT(int, ListPrice) LIKE '3%';
GO
```

B. Using CAST with arithmetic operators

The following example calculates a single column computation (Computed) by dividing the total year-to-date sales (SalesYTD) by the commission percentage (CommissionPCT). This result is converted to an int data type after being rounded to the nearest whole number.

```
USE AdventureWorks2008R2;
GO
SELECT CAST(ROUND(SalesYTD/CommissionPCT, 0) AS int) AS 'Computed'
FROM Sales.SalesPerson
WHERE CommissionPCT != 0;
GO
```

Here is the result set.

```
Computed
------
379753754
346698349
257144242
176493899
281101272
0
301872549
212623750
298948202
250784119
239246890
101664220
124511336
97688107
(14 row(s) affected)
```

C. Using CAST to concatenate

The following example concatenates noncharacter, nonbinary expressions by using CAST.

```
USE AdventureWorks2008R2;
GO
SELECT 'The list price is ' + CAST(ListPrice AS varchar(12)) AS ListPrice
FROM Production.Product
WHERE ListPrice BETWEEN 350.00 AND 400.00;
GO
```

Here is the result set.

```
ListPrice
-----------------
The list price is 357.06
The list price is 364.09
```

The list price is 364.09
The list price is 364.09
The list price is 364.09
(5 row(s) affected)

D. Using CAST to produce more readable text

The following example uses CAST in the select list to convert the Name column to a char(10) column.

```
USE AdventureWorks2008R2;
GO
SELECT DISTINCT CAST(p.Name AS char(10)) AS Name, s.UnitPrice
FROM Sales.SalesOrderDetail s JOIN Production.Product p on s.ProductID = p.Pr
oductID
WHERE Name LIKE 'Long-Sleeve Logo Jersey, M';
GO
```

Here is the result set.

```
Name UnitPrice
---------- ---------------------
Long-Sleev 31.2437
Long-Sleev 32.4935
Long-Sleev 49.99
(3 row(s) affected)
```

E. Using CAST with the LIKE clause

The following example converts the money column SalesYTD to an int and then to a char(20) column so that it can be used with the LIKE clause.

```
USE AdventureWorks2008R2;
GO
SELECT p.FirstName, p.LastName, s.SalesYTD, s.BusinessEntityID
FROM Person.Person p JOIN Sales.SalesPerson s ON p.BusinessEntityID = s.Busin
essEntityID
WHERE CAST(CAST(s.SalesYTD AS int) AS char(20)) LIKE '2%';
GO
```

Here is the result set.

```
FirstName LastName SalesYTD SalesPersonID
--------------- ------------------ --------------- -------------
Tsvi Reiter 2811012.7151 279
Syed Abbas 219088.8836 288
Rachel Valdez 2241204.0424 289
(3 row(s) affected)
```

F. Using CONVERT or CAST with typed XML

The following are several examples that show using CONVERT to convert to typed XML by using the xml data type.

This example converts a string with white space, text and markup into typed XML and removes all insignificant white space (boundary white space between nodes):

```
CONVERT(XML, '<root><child/></root>')
```

This example converts a similar string with white space, text and markup into typed XML and preserves insignificant white space (boundary white space between nodes):

```
CONVERT(XML, '<root>          <child/>         </root>', 1)
```

This example casts a string with white space, text, and markup into typed XML:

```
CAST('<Name><FName>Carol</FName><LName>Elliot</LName></Name>'  AS XML)
```

For more examples, see Generating XML Instances.

G. Using CAST and CONVERT with datetime data

The following example displays the current date and time, uses CAST to change the current date and time to a character data type, and then uses CONVERT display the date and time in the ISO 8901 format.

```
SELECT
   GETDATE() AS UnconvertedDateTime,
   CAST(GETDATE() AS nvarchar(30)) AS UsingCast,
   CONVERT(nvarchar(30), GETDATE(), 126) AS UsingConvertTo_ISO8601  ;
GO
```

Here is the result set.

```
UnconvertedDateTime UsingCast UsingConvertTo_ISO8601
---------------------- ---------------------------- ----------------------------
2006-04-18 09:58:04.570 Apr 18 2006 9:58AM 2006-04-18T09:58:04.570
(1 row(s) affected)
```

The following example is approximately the opposite of the previous example. The example displays a date and time as character data, uses CAST to change the character data to the datetime data type, and then uses CONVERT to change the character data to the datetime data type.

```
SELECT
   '2006-04-25T15:50:59.997' AS UnconvertedText,
   CAST('2006-04-25T15:50:59.997' AS datetime) AS UsingCast,
   CONVERT(datetime, '2006-04-25T15:50:59.997', 126) AS UsingConvertFrom_ISO8
601 ;
GO
```

Here is the result set.

```
UnconvertedText UsingCast UsingConvertFrom_ISO8601
----------------------- ----------------------- -----------------------
2006-04-25T15:50:59.997 2006-04-25 15:50:59.997 2006-04-25 15:50:59.997
(1 row(s) affected)
```

H. Using CONVERT with binary and character data

The following examples show the results of converting binary and character data by using different styles.

```
--Convert the binary value 0x4E616d65 to a character value.
SELECT CONVERT(char(8), 0x4E616d65, 0) AS 'Style 0, binary to character'
```

Here is the result set.

```
Style 0, binary to character
----------------------------
Name
(1 row(s) affected)
```

```
--The following example shows how Style 1 can force the result
--to be truncated. The truncation is caused by
--including the characters 0x in the result.
SELECT CONVERT(char(8), 0x4E616d65, 1) AS 'Style 1, binary to character'
```

Here is the result set.

```
Style 1, binary to character
----------------------------
0x4E616D
(1 row(s) affected)
```

```
--The following example shows that Style 2 does not truncate the
--result because the characters 0x are not included in
--the result.
SELECT CONVERT(char(8), 0x4E616d65, 2) AS 'Style 2, binary to character'
```

Here is the result set.

```
Style 2, binary to character
----------------------------
4E616D65
(1 row(s) affected)
```

```
--Convert the character value 'Name' to a binary value.
SELECT CONVERT(binary(8), 'Name', 0) AS 'Style 0, character to binary'
```

Here is the result set.

```
Style 0, character to binary
--------------------------------
0x4E616D6500000000
(1 row(s) affected)
```

```
SELECT CONVERT(binary(4), '0x4E616D65', 1) AS 'Style 1, character to binary'
```

Here is the result set.

```
Style 1, character to binary
--------------------------------
0x4E616D65
(1 row(s) affected)
```

```
SELECT CONVERT(binary(4), '4E616D65', 2) AS 'Style 2, character to binary'
```

Here is the result set.

Style 2, character to binary

0x4E616D65
(1 row(s) affected)

3.12.2. CONNECTIONPROPERTY

Returns information about the connection properties for the unique connection that a request came in on.

Syntax

```
CONNECTIONPROPERTY ( property )
```

Arguments

property

Is the property of the connection. *property* can be one of the following values.

Value	Data type	Description
net_transport	nvarchar(40)	Returns the physical transport protocol that is used by this connection. Is not nullable. **Note** Always returns Session when a connection has multiple active result sets (MARS) enabled, and connection pooling is enabled.
protocol_type	nvarchar(40)	Returns the protocol type of the payload. It currently distinguishes between TDS (TSQL) and SOAP. Is nullable.
auth_scheme	nvarchar(40)	Returns the SQL Server Authentication scheme for a connection. The authentication scheme is either Windows Authentication (NTLM, KERBEROS, DIGEST, BASIC, NEGOTIATE) or SQL Server Authentication. Is not nullable.
local_net_address	varchar(48)	Returns the IP address on the server that this connection targeted. Available only for connections that are using the TCP transport provider. Is nullable.
local_tcp_port	int	Returns the server TCP port that this connection targeted if the connection were a connection that is using the TCP transport. Is nullable.

client_net_address	varchar(48)	Asks for the address of the client that is connecting to this server. Is nullable.
<Any other string>		Returns NULL if the input is not valid.

Remarks

The values that are returned are the same as the options shown for the corresponding columns in the sys.dm_exec_connections dynamic management view. For example:

```
SELECT
ConnectionProperty('net_transport') AS 'Net transport',
ConnectionProperty('protocol_type') AS 'Protocol type'
```

3.12.3. FORMATMESSAGE

Constructs a message from an existing message in **sys.messages**. The functionality of FORMATMESSAGE resembles that of the RAISERROR statement. However, RAISERROR prints the message immediately, while FORMATMESSAGE returns the formatted message for further processing.

Syntax

```
FORMATMESSAGE ( msg_number , [ param_value [ ,...n ] ] )
```

Arguments

msg_number

Is the ID of the message stored in **sys.messages**. If *msg_number* is <= 13000, or if the message does not exist in **sys.messages**, NULL is returned.

param_value

Is a parameter value for use in the message. Can be more than one parameter value. The values must be specified in the order in which the placeholder variables appear in the message. The maximum number of values is 20.

Return Types

nvarchar

Remarks

Like the RAISERROR statement, FORMATMESSAGE edits the message by substituting the supplied parameter values for placeholder variables in the message. For more information about the placeholders allowed in error messages and the editing process, see RAISERROR.

Note

FORMATMESSAGE works only with messages created using **sp_addmessage**.

FORMATMESSAGE looks up the message in the current language of the user. If there is no localized version of the message, the U.S. English version is used.

For localized messages, the supplied parameter values must correspond to the parameter placeholders in the U.S. English version. That is, parameter 1 in the localized version must correspond to parameter 1 in the U.S. English version, parameter 2 must correspond to parameter 2, and so on.

Examples

This example uses a hypothetical message 50001 stored in **sys.messages** as, "The number of rows in %s is %1d." FORMATMESSAGE substitutes the values Table1 and 5 for the parameter placeholders. The resulting string, "The number of rows in Table1 is 5," is stored in the local variable @var1.

```
DECLARE @var1 VARCHAR(100)
SELECT @var1 = FORMATMESSAGE(50001, 'Table1', 5)
```

3.12.4. ISNULL

Replaces NULL with the specified replacement value.

Syntax

```
ISNULL ( check_expression , replacement_value )
```

Arguments

check_expression

Is the expression to be checked for NULL. *check_expression* can be of any type.

replacement_value

Is the expression to be returned if *check_expression* is NULL. *replacement_value* must be of a type that is implicitly convertible to the type of *check_expresssion*.

Return Types

Returns the same type as *check_expression*.

Remarks

The value of *check_expression* is returned if it is not NULL; otherwise, *replacement_value* is returned after it is implicitly converted to the type of *check_expression*, if the types are different.

Examples

A. Using ISNULL with AVG

The following example finds the average of the weight of all products. It substitutes the value 50 for all NULL entries in the Weight column of the Product table.

```
USE AdventureWorks2008R2;
GO
SELECT AVG(ISNULL(Weight, 50))
FROM Production.Product;
GO
```

Here is the result set.

```
-------------------------
59.79
(1 row(s) affected)
```

B. Using ISNULL

The following example selects the description, discount percentage, minimum quantity, and maximum quantity for all special offers in AdventureWorks2008R2. If the maximum quantity for a particular special offer is NULL, the MaxQty shown in the result set is 0.00.

```
USE AdventureWorks2008R2;
GO
SELECT Description, DiscountPct, MinQty, ISNULL(MaxQty, 0.00) AS 'Max Quantit
y'
FROM Sales.SpecialOffer;
GO
```

Here is the result set.

```
Description DiscountPct MinQty Max Quantity
--------------- ------------ -------- ---------------
No Discount 0.00 0 0
Volume Discount 0.02 11 14
Volume Discount 0.05 15 4
Volume Discount 0.10 25 0
Volume Discount 0.15 41 0
Volume Discount 0.20 61 0
Mountain-100 Cl 0.35 0 0
Sport Helmet Di 0.10 0 0
Road-650 Overst 0.30 0 0
Mountain Tire S 0.50 0 0
Sport Helmet Di 0.15 0 0
LL Road Frame S 0.35 0 0
Touring-3000 Pr 0.15 0 0
Touring-1000 Pr 0.20 0 0
Half-Price Peda 0.50 0 0
Mountain-500 Si 0.40 0 0
(16 row(s) affected)
```

C. Testing for NULL in a WHERE clause

Do not use ISNULL to find NULL values. Use IS NULL instead. The following example finds all products that have NULL in the weight column. Note the space between IS and NULL.

```
USE AdventureWorks2008R2;
GO
SELECT Name, Weight
FROM Production.Product
WHERE Weight IS NULL;
GO
```

4. Control-of-Flow Language

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

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

4.1. BEGIN...END

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.

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. For more information, see Batches and the individual statements used.

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 AdventureWorks2008R2;
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 tranaction.
*/
```

4.2. BREAK

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.

4.3. CONTINUE

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 and Control-of-Flow Language.

4.4. ELSE (IF...ELSE)

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.

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 AdventureWorks2008R2;
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 AdventureWorks2008R2;
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(@AvgW
eight 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 v
archar(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
```

4.5. END (BEGIN...END)

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

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

4.6. GOTO

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.

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.'
```

4.7. IF...ELSE

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.

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.

Examples

The following example uses IF...ELSE with output from the uspGetList stored procedure. This stored procedure is defined in Creating Stored Procedures. In this example, the stored procedure returns a list of bikes with a list price less than $700. This causes the first PRINT statement to execute.

```
DECLARE @compareprice money, @cost money
EXECUTE Production.uspGetList '%Bikes%', 700,
    @compareprice OUT,
    @cost OUTPUT
IF @cost <= @compareprice
BEGIN
    PRINT 'These products can be purchased for less than
    $'+RTRIM(CAST(@compareprice AS varchar(20)))+'.'
END
ELSE
    PRINT 'The prices for all products in this category exceed
    $'+ RTRIM(CAST(@compareprice AS varchar(20)))+'.'
```

4.8. RETURN

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.

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 BusinessEntityID that did not match a row).

```
USE AdventureWorks2008R2;
GO
CREATE PROCEDURE checkstate @param varchar(11)
AS
IF (SELECT StateProvince FROM Person.vAdditionalContactInfo WHERE BusinessEnt
ityID = @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 '291';
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
```

4.9. TRY...CATCH

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.

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. For more information about how to use RAISERROR with TRY...CATCH, see Using TRY...CATCH in Transact-SQL.

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.

```
USE AdventureWorks2008R2;
GO
-- 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;
```

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. For more information, see Using TRY...CATCH in Transact-SQL.

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.

```
USE AdventureWorks2008R2;
GO

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;
```

- For more information about batches, see Batches.

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 Using TRY...CATCH in Transact-SQL and XACT_STATE.

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.

```
USE AdventureWorks2008R2;
GO

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.

```
USE AdventureWorks2008R2;
GO
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.

```
USE AdventureWorks2008R2;
GO

-- 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
```

4.10. WAITFOR

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.

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.

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.

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.

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.

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 and GET CONVERSATION GROUP.

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 query wait 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.

Examples

A. Using WAITFOR TIME

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

```
USE msdb;
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.

```
USE AdventureWorks2008R2;
GO
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.

4.11. WHILE

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.

Syntax

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

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 AdventureWorks2008R2;
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 AdventureWorks2008R2.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
```

5. Cursors

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
```

5.1. CLOSE

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.

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 BusinessEntityID, JobTitle
FROM AdventureWorks2008R2.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
```

5.2. DEALLOCATE

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

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 AdventureWorks2008R2;
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 AdventureWorks2008R2;
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
```

5.3. 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. DECLARE CURSOR accepts both a syntax based on the ISO standard and a syntax using a set of Transact-SQL extensions.

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. For more information about rules for identifiers, see Using Identifiers As Object Names.

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 COMPUTE, COMPUTE BY, 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. For more information, see Using Implicit Cursor Conversions.

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. For more information about rules for identifiers, see Using Identifiers As Object Names.

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. In SQL Server version 7.0, this option defaults to FALSE to match earlier versions of SQL Server, in which all cursors were global. The default of this option may change in future versions of SQL Server. For more information, see Setting Database Options.

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**
> In SQL Server 2000, FAST_FORWARD and FORWARD_ONLY cursor options are mutually exclusive. If both are specified, an error is raised. In SQL Server 2005 and later, both keywords 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. In SQL Server version 6.5 and earlier, variable values are refreshed every time a cursor is reopened.

Permissions

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

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.

```
USE AdventureWorks2008R2;
GO
DECLARE vend_cursor CURSOR
    FOR SELECT BusinessEntityID, Name, CreditRating 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.

```
USE AdventureWorks2008R2;
GO
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 BusinessEntityID, Name
FROM Purchasing.Vendor
WHERE PreferredVendorStatus = 1
ORDER BY BusinessEntityID;

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 AS pv
    INNER JOIN Production.Product AS v
        ON pv.ProductID = v.ProductID AND
           pv.BusinessEntityID = @vendor_id;  -- Variable value from the oute
r 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;
```

5.4. FETCH

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

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.

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 AdventureWorks2008R2;
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 AdventureWorks2008R2;
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 AdventureWorks2008R2;
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
```

5.5. OPEN

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.

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 AdventureWorks2008R2.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;
```

6. Data Types

In SQL Server, each column, local variable, expression, and parameter has a related data type. A data type is an attribute that specifies the type of data that the object can hold: integer data, character data, monetary data, date and time data, binary strings, and so on.

SQL Server supplies a set of system data types that define all the types of data that can be used with SQL Server. You can also define your own data types in Transact-SQL or the Microsoft .NET Framework. Alias data types are based on the system-supplied data types. For more information about alias data types, see Working with Alias Data Types. User-defined types obtain their characteristics from the methods and operators of a class that you create by using one of the programming languages support by the .NET Framework. For more information, see Working with CLR User-defined Types.

When two expressions that have different data types, collations, precision, scale, or length are combined by an operator, the characteristics of result are determined by the following:

- The data type of the result is determined by applying the rules of data type precedence to the data types of the input expressions. For more information, see Data Type Precedence.
- The collation of the result is determined by the rules of collation precedence when the result data type is char, varchar, text, nchar, nvarchar, or ntext. For more information, see Collation Precedence.
- The precision, scale, and length of the result depend on the precision, scale, and length of the input expressions. For more information, see Precision, Scale, and Length.

SQL Server provides data type synonyms for ISO compatibility. For more information, see Data Type Synonyms.

Data Type Categories

Data types in SQL Server are organized into the following categories:

Exact numerics	Unicode character strings
Approximate numerics	Binary strings
Date and time	Other data types
Character strings	

In SQL Server, based on their storage characteristics, some data types are designated as belonging to the following groups:

- Large value data types: varchar(max), nvarchar(max), and varbinary(max)
- Large object data types: text, ntext, image, varchar(max), nvarchar(max), varbinary(max), and xml

Note

sp_help returns -1 as the length for the large-value and xml data types.

Exact Numerics

bigint	numeric
bit	smallint
decimal	smallmoney
int	tinyint
money	

Approximate Numerics

float	real

Date and Time

date	datetimeoffset
datetime2	smalldatetime
datetime	time

Character Strings

char	varchar
text	

Unicode Character Strings

nchar	nvarchar
ntext	

Binary Strings

binary	varbinary
image	

Other Data Types

cursor	timestamp
hierarchyid	uniqueidentifier
sql_variant	xml

`table`	

6.1. Constants

A constant, also known as a literal or a scalar value, is a symbol that represents a specific data value. The format of a constant depends on the data type of the value it represents.

Character string constants

Character string constants are enclosed in single quotation marks and include alphanumeric characters (a-z, A-Z, and 0-9) and special characters, such as exclamation point (!), at sign (@), and number sign (#). Character string constants are assigned the default collation of the current database, unless the COLLATE clause is used to specify a collation. Character strings typed by users are evaluated through the code page of the computer and are translated to the database default code page if it is required.

If the QUOTED_IDENTIFIER option has been set OFF for a connection, character strings can also be enclosed in double quotation marks, but the Microsoft SQL Server Native Client Provider and ODBC driver automatically use SET QUOTED_IDENTIFIER ON. We recommend using single quotation marks.

If a character string enclosed in single quotation marks contains an embedded quotation mark, represent the embedded single quotation mark with two single quotation marks. This is not required in strings embedded in double quotation marks.

The following are examples of character strings:

```
'Cincinnati'
'O''Brien'
'Process X is 50% complete.'
'The level for job_id: %d should be between %d and %d.'
"O'Brien"
```

Empty strings are represented as two single quotation marks with nothing in between. In 6.x compatibility mode, an empty string is treated as a single space.

Character string constants support enhanced collations.

Note
Character constants greater than 8000 bytes are typed as **varchar(max)** data.

Unicode strings

Unicode strings have a format similar to character strings but are preceded by an N identifier (N stands for National Language in the SQL-92 standard). The N prefix must be uppercase. For example, 'Michél' is a character constant while N'Michél' is a Unicode constant. Unicode constants are interpreted as Unicode data, and are not evaluated by using a code page. Unicode constants do have a collation. This collation primarily controls comparisons and case sensitivity. Unicode

constants are assigned the default collation of the current database, unless the COLLATE clause is used to specify a collation. Unicode data is stored by using 2 bytes per character instead of 1 byte per character for character data. For more information, see Using Unicode Data.

Unicode string constants support enhanced collations.

> **Note**
>
> Unicode constants greater than 8000 bytes are typed as **nvarchar(max)** data.

Binary constants

Binary constants have the prefix 0x and are a string of hexadecimal numbers. They are not enclosed in quotation marks.

The following are examples of binary strings are:

```
0xAE
0x12Ef
0x69048AEFDD010E
0x  (empty binary string)
```

> **Note**
>
> Binary constants greater than 8000 bytes are typed as **varbinary(max)** data.

bit constants

bit constants are represented by the numbers 0 or 1, and are not enclosed in quotation marks. If a number larger than one is used, it is converted to one.

datetime constants

datetime constants are represented by using character date values in specific formats, enclosed in single quotation marks. For more information about the formats for datetime constants, see Using Date and Time Data.

The following are examples of datetime constants:

```
'December 5, 1985'
'5 December, 1985'
'851205'
'12/5/98'
```

Examples of time constants are:

```
'14:30:24'
'04:24 PM'
```

integer constants

integer constants are represented by a string of numbers that are not enclosed in quotation marks and do not contain decimal points. integer constants must be whole numbers; they cannot contain decimals.

The following are examples of integer constants:

```
1894
2
```

decimal constants

decimal constants are represented by a string of numbers that are not enclosed in quotation marks and contain a decimal point.

The following are examples of decimal constants:

```
1894.1204
2.0
```

float and real constants

float and real constants are represented by using scientific notation.

The following are examples of float or real values:

```
101.5E5
0.5E-2
```

money constants

money constants are represented as string of numbers with an optional decimal point and an optional currency symbol as a prefix. **money** constantsare not enclosed in quotation marks.

SQL Server does not enforce any kind of grouping rules such as inserting a comma (,) every three digits in strings that represent money.

Note
Commas are ignored anywhere in the specified money literal.

The following are examples of money constants:

```
$12
$542023.14
```

uniqueidentifier constants

uniqueidentifier constants are a string representing a GUID. They can be specified in either a character or binary string format.

The following examples both specify the same GUID:

```
'6F9619FF-8B86-D011-B42D-00C04FC964FF'
0xff19966f868b11d0b42d00c04fc964ff
```

Specifying Negative and Positive Numbers

To indicate whether a number is positive or negative, apply the **+** or **-** unary operators to a numeric constant. This creates a numeric expression that represents the signed numeric value. Numeric constants use positive when the **+** or **-** unary operators are not applied.

- Signed integer expressions:

  ```
  +145345234
  -2147483648
  ```

- Signed decimal expressions:

  ```
  +145345234.2234
  -2147483648.10
  ```

- Signed float expressions:

  ```
  +123E-3
  -12E5
  ```

- Signed money expressions:

  ```
  -$45.56
  +$423456.99
  ```

6.2. Data Type Precedence

When an operator combines two expressions of different data types, the rules for data type precedence specify that the data type with the lower precedence is converted to the data type with the higher precedence. If the conversion is not a supported implicit conversion, an error is returned. When both operand expressions have the same data type, the result of the operation has that data type.

SQL Server uses the following precedence order for data types:

1. **user-defined data types (highest)**
2. **sql_variant**
3. **xml**
4. **datetimeoffset**
5. **datetime2**
6. **datetime**
7. **smalldatetime**
8. **date**
9. **time**
10. **float**
11. **real**
12. **decimal**
13. **money**
14. **smallmoney**
15. **bigint**
16. **int**
17. **smallint**

18. **tinyint**
19. **bit**
20. **ntext**
21. **text**
22. **image**
23. **timestamp**
24. **uniqueidentifier**
25. **nvarchar** (including nvarchar(max))
26. **nchar**
27. **varchar** (including varchar(max))
28. **char**
29. **varbinary** (including varbinary(max))
30. **binary** (lowest)

6.3. Precision, Scale, and Length

Precision is the number of digits in a number. Scale is the number of digits to the right of the decimal point in a number. For example, the number 123.45 has a precision of 5 and a scale of 2.

In SQL Server, the default maximum precision of numeric and decimal data types is 38. In earlier versions of SQL Server, the default maximum is 28.

Length for a numeric data type is the number of bytes that are used to store the number. Length for a character string or Unicode data type is the number of characters. The length for binary, varbinary, and image data types is the number of bytes. For example, an int data type can hold 10 digits, is stored in 4 bytes, and does not accept decimal points. The int data type has a precision of 10, a length of 4, and a scale of 0.

When two char, varchar, binary, or varbinary expressions are concatenated, the length of the resulting expression is the sum of the lengths of the two source expressions or 8,000 characters, whichever is less.

When two nchar or nvarchar expressions are concatenated, the length of the resulting expression is the sum of the lengths of the two source expressions or 4,000 characters, whichever is less.

When two expressions of the same data type but different lengths are compared by using UNION, EXCEPT, or INTERSECT, the resulting length is the maximum length of the two expressions.

The precision and scale of the numeric data types besides decimal are fixed. If an arithmetic operator has two expressions of the same type, the result has the same data type with the precision and scale defined for that type. If an operator has two expressions with different numeric data types, the rules of data type precedence define the data type of the result. The result has the precision and scale defined for its data type.

The following table defines how the precision and scale of the result are calculated when the result of an operation is of type decimal. The result is decimal when either of the following is true:

- Both expressions are decimal.
- One expression is decimal and the other is a data type with a lower precedence than decimal.

The operand expressions are denoted as expression e1, with precision p1 and scale s1, and expression e2, with precision p2 and scale s2. The precision and scale for any expression that is not decimal is the precision and scale defined for the data type of the expression.

Operation	Result precision	Result scale *
e1 + e2	max(s1, s2) + max(p1-s1, p2-s2) + 1	max(s1, s2)
e1 - e2	max(s1, s2) + max(p1-s1, p2-s2) + 1	max(s1, s2)
e1 * e2	p1 + p2 + 1	s1 + s2
e1 / e2	p1 - s1 + s2 + max(6, s1 + p2 + 1)	max(6, s1 + p2 + 1)
e1 { UNION | EXCEPT | INTERSECT } e2	max(s1, s2) + max(p1-s1, p2-s2)	max(s1, s2)
e1 % e2	min(p1-s1, p2 -s2) + max(s1,s2)	max(s1, s2)

* The result precision and scale have an absolute maximum of 38. When a result precision is greater than 38, the corresponding scale is reduced to prevent the integral part of a result from being truncated.

6.4. bit

An integer data type that can take a value of 1, 0, or NULL.

Remarks

The SQL Server Database Engine optimizes storage of bit columns. If there are 8 or less bit columns in a table, the columns are stored as 1 byte. If there are from 9 up to 16 bit columns, the columns are stored as 2 bytes, and so on.

The string values TRUE and FALSE can be converted to bit values: TRUE is converted to 1 and FALSE is converted to 0.

6.5. cursor

A data type for variables or stored procedure OUTPUT parameters that contain a reference to a cursor. Any variables created with the cursordata type are nullable.

The operations that can reference variables and parameters having a **cursor** data type are:

- The DECLARE *@local_variable* and SET *@local_variable* statements.
- The OPEN, FETCH, CLOSE, and DEALLOCATE cursor statements.

- Stored procedure output parameters.
- The CURSOR_STATUS function.
- The **sp_cursor_list**, **sp_describe_cursor**, **sp_describe_cursor_tables**, and **sp_describe_cursor_columns** system stored procedures.

Important
The cursordata type cannot be used for a column in a CREATE TABLE statement.

Note
In this version of SQL Server, the **cursor_name** output column of **sp_cursor_list** and **sp_describe_cursor** returns the name of the cursor variable. In previous releases, this output column returns a system-generated name.

6.6. Date and Time Types

SQL Server supports the following date and time types.

6.6.1. date

Defines a date.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions see Using Date and Time Data.

date Description

Property	Value
Syntax	date
Usage	DECLARE @MyDate date CREATE TABLE Table1 (Column1 date)
Default string literal format (used for down-level client)	YYYY-MM-DD For more information, see the "Backward Compatibility for Down-level Clients" section of Using Date and Time Data.
Range	0001-01-01 through 9999-12-31 January 1, 1 A.D. through December 31, 9999 A.D.

Element ranges	YYYY is four digits from 0001 to 9999 that represent a year. MM is two digits from 01 to 12 that represent a month in the specified year. DD is two digits from 01 to 31, depending on the month, that represent a day of the specified month.
Character length	10 positions
Precision, scale	10, 0
Storage size	3 bytes, fixed
Storage structure	1, 3-byte integer stores date.
Accuracy	One day
Default value	1900-01-01 This value is used for the appended date part for implicit conversion from time to datetime2 or datetimeoffset.
Calendar	Gregorian
User-defined fractional second precision	No
Time zone offset aware and preservation	No
Daylight saving aware	No

Supported String Literal Formats for date

The following tables show the valid string literal formats for the date data type.

Numeric	Description
mdy [m]m/dd/[yy]yy [m]m-dd-[yy]yy [m]m.dd.[yy]yy	[m]m, dd, and [yy]yy represents month, day, and year in a string with slash marks (/), hyphens (-), or periods (.) as separators. Only four- or two-digit years are supported. Use four-digit years whenever possible. To specify an integer from 0001 to 9999 that represents the cutoff year

myd mm/[yy]yy/dd mm-[yy]yy/dd [m]m.[yy]yy.dd dmy dd/[m]m/[yy]yy dd-[m]m-[yy]yy dd.[m]m.[yy]yy dym dd/[yy]yy/[m]m dd-[yy]yy-[m]m dd.[yy]yy.[m]m ymd [yy]yy/[m]m/dd [yy]yy-[m]m-dd [yy]yy-[m]m-dd	for interpreting two-digit years as four-digit years, use the two digit year cutoff Option. A two-digit year that is less than or equal to the last two digits of the cutoff year is in the same century as the cutoff year. A two-digit year that is greater than the last two digits of the cutoff year is in the century that comes before the cutoff year. For example, if the two-digit year cutoff is the default 2049, the two-digit year 49 is interpreted as 2049 and the two-digit year 50 is interpreted as 1950. The default date format is determined by the current language setting. You can change the date format by using the SET LANGUAGE and SET DATEFORMAT statements. The ydm format is not supported for date.
Alphabetical	Description
mon [dd][,] yyyy mon dd[,] [yy]yy mon yyyy [dd] [dd] mon[,] yyyy dd mon[,][yy]yy dd [yy]yy mon [dd] yyyy mon yyyy mon [dd] yyyy [dd] mon	mon represents the full month name or the month abbreviation given in the current language. Commas are optional and capitalization is ignored. To avoid ambiguity, use four-digit years. If the day is missing, the first day of the month is supplied.
ISO 8601	Descripton
YYYY-MM-DD YYYYMMDD	Same as the SQL standard. This is the only format that is defined as an international standard.
Unseparated	Description
[yy]yymmdd yyyy[mm][dd]	The **date** data can be specified with four, six, or eight digits. A six- or eight-digit string is always interpreted as ymd. The month and day must always be two digits. A four-digit string is interpreted as year.

ODBC	Description
{ d 'yyyy-mm-dd' }	ODBC API specific. Functions in SQL Server 2008 as in SQL Server 2005.
W3C XML format	**Description**
yyyy-mm-ddTZD	Specifically supported for XML/SOAP usage. TZD is the time zone designator (Z or +hh:mm or -hh:mm): • hh:mm represents the time zone offset. hh is two digits, ranging from 0 to 14, that represent the number of hours in the time zone offset. • MM is two digits, ranging from 0 to 59, that represent the number of additional minutes in the time zone offset. • + (plus) or – (minus) the mandatory sign of the time zone offset. This indicates that the time zone offset is added or subtracted from the Coordinated Universal Times (UTC) time to obtain the local time. The valid range of time zone offset is from -14:00 to +14:00.

ANSI and ISO 8601 Compliance

date complies with the ANSI SQL standard definition for the Gregorian calendar: "NOTE 85 - Datetime data types will allow dates in the Gregorian format to be stored in the date range 0001–01–01 CE through 9999–12–31 CE."

The default string literal format, which is used for down-level clients, complies with the SQL standard form which is defined as YYYY-MM-DD. This format is the same as the ISO 8601 definition for DATE.

Examples

The following example compares the results of casting a string to each date and time data type.

```
SELECT
     CAST('2007-05-08 12:35:29. 1234567 +12:15' AS time(7)) AS 'time'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS date) AS 'date'
    ,CAST('2007-05-08 12:35:29.123' AS smalldatetime) AS
        'smalldatetime'
    ,CAST('2007-05-08 12:35:29.123' AS datetime) AS 'datetime'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS datetime2(7)) AS
        'datetime2'
    ,CAST('2007-05-08 12:35:29.1234567 +12:15' AS datetimeoffset(7)) AS
        'datetimeoffset';
```

Here is the result set.

Data type	Output

time	12:35:29. 1234567
date	2007-05-08
smalldatetime	2007-05-08 12:35:00
datetime	2007-05-08 12:35:29.123
datetime2	2007-05-08 12:35:29. 1234567
datetimeoffset	2007-05-08 12:35:29.1234567 +12:15

6.6.2. datetime

Defines a date that is combined with a time of day with fractional seconds that is based on a 24-hour clock.

Note

Use the time, date, datetime2 and datetimeoffset data types for new work. These types align with the SQL Standard. They are more portable. time, datetime2 and datetimeoffset provide more seconds precision. datetimeoffset provides time zone support for globally deployed applications.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

datetime Description

Property	Value
Syntax	datetime
Usage	DECLARE @MyDatetime datetime CREATE TABLE Table1 (Column1 datetime)
Default string literal formats (used for down-level client)	Not applicable
Date range	January 1, 1753, through December 31, 9999
Time range	00:00:00 through 23:59:59.997

Time zone offset range	None
Element ranges	YYYY is four digits from 1753 through 9999 that represent a year. MM is two digits, ranging from 01 to 12, that represent a month in the specified year. DD is two digits, ranging from 01 to 31 depending on the month, that represent a day of the specified month. hh is two digits, ranging from 00 to 23, that represent the hour. mm is two digits, ranging from 00 to 59, that represent the minute. ss is two digits, ranging from 00 to 59, that represent the second. n* is zero to three digits, ranging from 0 to 999, that represent the fractional seconds.
Character length	19 positions minimum to 23 maximum
Storage size	8 bytes
Accuracy	Rounded to increments of .000, .003, or .007 seconds
Default value	1900-01-01 00:00:00
Calendar	Gregorian (Does not include the complete range of years.)
User-defined fractional second precision	No
Time zone offset aware and preservation	No
Daylight saving aware	No

Supported String Literal Formats for datetime

The following tables list the supported string literal formats for datetime. Except for ODBC, datetime string literals are in single quotation marks ('), for example, 'string_literaL'. If the environment is not **us_english**, the string literals should be in the format N'string_literaL'.

Numeric	Description
Date formats:	You can specify date data with a numeric month specified. For example,

[0]4/15/[19]96 -- (mdy) [0]4-15-[19]96 -- (mdy) [0]4.15.[19]96 -- (mdy) [0]4/[19]96/15 -- (myd) 15/[0]4/[19]96 -- (dmy) 15/[19]96/[0]4 -- (dym) [19]96/15/[0]4 -- (ydm) [19]96/[0]4/15 -- (ymd) Time formats: 14:30 14:30[:20:999] 14:30[:20.9] 4am 4 PM	5/20/97 represents the twentieth day of May 1997. When you use numeric date format, specify the month, day, and year in a string that uses slash marks (/), hyphens (-), or periods (.) as separators. This string must appear in the following form: • number separator number separator *number* [*time*] [*time*] When the language is set to **us_english**, the default order for the date is mdy. You can change the date order by using the SET DATEFORMAT statement. The setting for SET DATEFORMAT determines how date values are interpreted. If the order does not match the setting, the values are not interpreted as dates, because they are out of range or the values are misinterpreted. For example, 12/10/08 can be interpreted as one of six dates, depending on the DATEFORMAT setting. A four-part year is interpreted as the year.
Alphabetical	Description
Apr[il] [15][,] 1996 Apr[il] 15[,] [19]96 Apr[il] 1996 [15] [15] Apr[il][,] 1996 15 Apr[il][,][19]96 15 [19]96 apr[il] [15] 1996 apr[il] 1996 APR[IL] [15] 1996 [15] APR[IL]	You can specify date data with a month specified as the full month name. For example, April or the month abbreviation of Apr specified in the current language; commas are optional and capitalization is ignored. Here are some guidelines for using alphabetical date formats: • Enclose the date and time data in single quotation marks ('). For languages other than English, use N' • Characters that are enclosed in brackets are optional. • If you specify only the last two digits of the year, values less than the last two digits of the value of the two digit year cutoff configuration option are in the same century as the cutoff year. Values greater than or equal to the value of this option are in the century that comes before the cutoff year. For example, if **two digit year cutoff** is 2050 (default), 25 is interpreted as 2025 and 50 is interpreted as 1950. To avoid ambiguity, use four-digit years. • If the day is missing, the first day of the month is supplied. The SET DATEFORMAT session setting is not applied when you specify the month in alphabetical form.
ISO 8601	Description

YYYY-MM-DDThh:mm:ss[.mmm] YYYYMMDDThh:mm:ss[.mmm]	Examples: • 2004-05-23T14:25:10 • 2004-05-23T14:25:10.487 To use the ISO 8601 format, you must specify each element in the format. This also includes the T, the colons (:), and the period (.) that are shown in the format. The brackets indicate that the fraction of second component is optional. The time component is specified in the 24-hour format. The T indicates the start of the time part of the datetime value. The advantage in using the ISO 8601 format is that it is an international standard with unambiguous specification. Also, this format is not affected by the SET DATEFORMAT or SET LANGUAGE setting.

Unseparated	Description
YYYYMMDD hh:mm:ss[.mmm]	

ODBC	Description
{ ts '1998-05-02 01:23:56.123' } { d '1990-10-02' } { t '13:33:41' }	The ODBC API defines escape sequences to represent date and time values, which ODBC calls timestamp data. This ODBC timestamp format is also supported by the OLE DB language definition (DBGUID-SQL) supported by the Microsoft OLE DB provider for SQL Server. Applications that use the ADO, OLE DB, and ODBC-based APIs can use this ODBC timestamp format to represent dates and times. ODBC timestamp escape sequences are of the format: { *literal_type* '*constant_value*' }: • *literal_type* specifies the type of the escape sequence. Timestamps have three *literal_type* specifiers: • d = date only • t = time only • ts = timestamp (time + date) • '*constant_value*' is the value of the escape sequence. *constant_value* must follow these formats for each *literal_type*. literal_typeconstant_value format d yyyy-mm-dd t hh:mm:ss[.fff] ts yyyy-mm-dd hh:mm:ss[.fff]

Rounding of datetime Fractional Second Precision

datetime values are rounded to increments of .000, .003, or .007 seconds, as shown in the following table.

User-specified value	System stored value
01/01/98 23:59:59.999	1998-01-02 00:00:00.000
01/01/98 23:59:59.995 01/01/98 23:59:59.996 01/01/98 23:59:59.997 01/01/98 23:59:59.998	1998-01-01 23:59:59.997
01/01/98 23:59:59.992 01/01/98 23:59:59.993 01/01/98 23:59:59.994	1998-01-01 23:59:59.993
01/01/98 23:59:59.990 01/01/98 23:59:59.991	1998-01-01 23:59:59.990

ANSI and ISO 8601 Compliance

datetime is not ANSI or ISO 8601 compliant.

Examples

The following example compares the results of casting a string to each date and time data type.

```
SELECT
     CAST('2007-05-08 12:35:29. 1234567 +12:15' AS time(7)) AS 'time'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS date) AS 'date'
    ,CAST('2007-05-08 12:35:29.123' AS smalldatetime) AS
        'smalldatetime'
    ,CAST('2007-05-08 12:35:29.123' AS datetime) AS 'datetime'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS datetime2(7)) AS
        'datetime2'
    ,CAST('2007-05-08 12:35:29.1234567 +12:15' AS datetimeoffset(7)) AS
        'datetimeoffset';
```

Here is the result set.

Data type	Output
time	12:35:29. 1234567
date	2007-05-08
smalldatetime	2007-05-08 12:35:00
datetime	2007-05-08 12:35:29.123

datetime2	2007-05-08 12:35:29. 1234567
datetimeoffset	2007-05-08 12:35:29.1234567 +12:15

6.6.3. datetime2

Defines a date that is combined with a time of day that is based on 24-hour clock. datetime2 can be considered as an extension of the existing datetime type that has a larger date range, a larger default fractional precision, and optional user-specified precision.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

datetime2 Description

Property	Value
Syntax	datetime2 [(*fractional seconds precision*)]
Usage	DECLARE @MyDatetime2 datetime2(7) CREATE TABLE Table1 (Column1 datetime2(7))
Default string literal format (used for down-level client)	YYYY-MM-DD hh:mm:ss[.fractional seconds] For more information, see the "Backward Compatibility for Down-level Clients" section of Using Date and Time Data.
Date range	0001-01-01 through 9999-12-31 January 1,1 AD through December 31, 9999 AD
Time range	00:00:00 through 23:59:59.9999999
Time zone offset range	None
Element ranges	YYYY is a four-digit number, ranging from 0001 through 9999, that represents a year. MM is a two-digit number, ranging from 01 to 12, that represents a month in the specified year. DD is a two-digit number, ranging from 01 to 31 depending on the month, that represents a day of the specified month. hh is a two-digit number, ranging from 00 to 23, that represents the hour.

	mm is a two-digit number, ranging from 00 to 59, that represents the minute. ss is a two-digit number, ranging from 00 to 59, that represents the second. n* is a zero- to seven-digit number from 0 to 9999999 that represents the fractional seconds.
Character length	19 positions minimum (YYYY-MM-DD hh:mm:ss) to 27 maximum (YYYY-MM-DD hh:mm:ss.0000000)
Precision, scale	0 to 7 digits, with an accuracy of 100ns. The default precision is 7 digits.
Storage size	6 bytes for precisions less than 3; 7 bytes for precisions 3 and 4. All other precisions require 8 bytes.
Accuracy	100 nanoseconds
Default value	1900-01-01 00:00:00
Calendar	Gregorian
User-defined fractional second precision	Yes
Time zone offset aware and preservation	No
Daylight saving aware	No

For data type metadata, see sys.systypes or TYPEPROPERTY. Precision and scale are variable for some date and time data types. To obtain the precision and scale for a column, see COLUMNPROPERTY, COL_LENGTH, or sys.columns.

Supported String Literal Formats for datetime2

The following tables list the supported ISO 8601 and ODBC string literal formats for datetime2. For information about alphabetical, numeric, unseparated, and time formats for the date and time parts of datetime2, see date and time.

ISO 8601	Descriptions
YYYY-MM-DDThh:mm:ss[.nnnnnnn]	This format is not affected by the SET LANGUAGE and SET

YYYY-MM-DDThh:mm:ss[.nnnnnnn]	DATEFORMAT session locale settings. The T, the colons (:) and the period (.) are included in the string literal, for example '2007-05-02T19:58:47.1234567'.
ODBC	Description
{ ts 'yyyy-mm-dd hh:mm:ss[.fractional seconds]' }	ODBC API specific: • The number of digits to the right of the decimal point, which represents the fractional seconds, can be specified from 0 up to 7 (100 nanoseconds). • In SQL Server 2008, with compatibility level set to 10, the literal will internally map to the new time type.

ANSI and ISO 8601 Compliance

The ANSI and ISO 8601 compliance of date and time apply to datetime2.

Examples

The following example compares the results of casting a string to each date and time data type.

```
SELECT
     CAST('2007-05-08 12:35:29. 1234567 +12:15' AS time(7)) AS 'time'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS date) AS 'date'
    ,CAST('2007-05-08 12:35:29.123' AS smalldatetime) AS
        'smalldatetime'
    ,CAST('2007-05-08 12:35:29.123' AS datetime) AS 'datetime'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS datetime2(7)) AS
        'datetime2'
    ,CAST('2007-05-08 12:35:29.1234567 +12:15' AS datetimeoffset(7)) AS
        'datetimeoffset';
```

Here is the result set.

Data type	Output
time	12:35:29. 1234567
date	2007-05-08
smalldatetime	2007-05-08 12:35:00
datetime	2007-05-08 12:35:29.123
datetime2	2007-05-08 12:35:29. 1234567

`datetimeoffset`	2007-05-08 12:35:29.1234567 +12:15

6.6.4. smalldatetime

Defines a date that is combined with a time of day. The time is based on a 24-hour day, with seconds always zero (:00) and without fractional seconds.

Note

Use the time, date, datetime2 and datetimeoffset data types for new work. These types align with the SQL Standard. They are more portable. time, datetime2 and datetimeoffset provide more seconds precision. datetimeoffset provides time zone support for globally deployed applications.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

smalldatetime Description

Syntax	smalldatetime
Usage	DECLARE @MySmalldatetime smalldatetime CREATE TABLE Table1 (Column1 smalldatetime)
Default string literal formats (used for down-level client)	Not applicable
Date range	1900-01-01 through 2079-06-06 January 1, 1900, through June 6, 2079
Time range	00:00:00 through 23:59:59 2007-05-09 23:59:59 will round to 2007-05-10 00:00:00
Element ranges	YYYY is four digits, ranging from 1900, to 2079, that represent a year. MM is two digits, ranging from 01 to 12, that represent a month in the specified year. DD is two digits, ranging from 01 to 31 depending on the month, that represent a day of the specified month.

	hh is two digits, ranging from 00 to 23, that represent the hour. mm is two digits, ranging from 00 to 59, that represent the minute. ss is two digits, ranging from 00 to 59, that represent the second. Values that are 29.998 seconds or less are rounded down to the nearest minute, Values of 29.999 seconds or more are rounded up to the nearest minute.
Character length	19 positions maximum
Storage size	4 bytes, fixed.
Accuracy	One minute
Default value	1900-01-01 00:00:00
Calendar	Gregorian (Does not include the complete range of years.)
User-defined fractional second precision	No
Time zone offset aware and preservation	No
Daylight saving aware	No

ANSI and ISO 8601 Compliance

smalldatetime is not ANSI or ISO 8601 compliant.

Examples

A. Casting string literals with seconds to smalldatetime

The following example compares the conversion of seconds in string literals to smalldatetime.

```
SELECT
     CAST('2007-05-08 12:35:29'     AS smalldatetime)
    ,CAST('2007-05-08 12:35:30'     AS smalldatetime)
    ,CAST('2007-05-08 12:59:59.998' AS smalldatetime);
```

Input	Output
2007-05-08 12:35:29	2007-05-08 12:35:00

2007-05-08 12:35:30	2007-05-08 12:36:00
2007-05-08 12:59:59.998	2007-05-08 13:00:00

B. Comparing date and time data types

The following example compares the results of casting a string to each date and time data type.

```
SELECT
     CAST('2007-05-08 12:35:29. 1234567 +12:15' AS time(7)) AS 'time'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS date) AS 'date'
    ,CAST('2007-05-08 12:35:29.123' AS smalldatetime) AS
        'smalldatetime'
    ,CAST('2007-05-08 12:35:29.123' AS datetime) AS 'datetime'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS datetime2(7)) AS
        'datetime2'
    ,CAST('2007-05-08 12:35:29.1234567 +12:15' AS datetimeoffset(7)) AS
        'datetimeoffset';
```

Data type	Output
time	12:35:29. 1234567
date	2007-05-08
smalldatetime	2007-05-08 12:35:00
datetime	2007-05-08 12:35:29.123
datetime2	2007-05-08 12:35:29. 1234567
datetimeoffset	2007-05-08 12:35:29.1234567 +12:15

6.6.5. time

Defines a time of a day. The time is without time zone awareness and is based on a 24-hour clock.

For an overview of all Transact-SQL date and time data types and functions, see Date and Time Functions. For information and examples that are common to date and time data types and functions, see Using Date and Time Data.

time Description

Property	Value

Syntax	time [(*fractional second precision*)]
Usage	DECLARE @MyTime time(7) CREATE TABLE Table1 (Column1 time(7))
fractional seconds precision	Specifies the number of digits for the fractional part of the seconds. This can be an integer from 0 to 7. The default fractional precision is 7 (100ns).
Usage	DECLARE @MyTime time(7) CREATE TABLE Table1 (Column1 time(7))
Default string literal format (used for down-level client)	hh:mm:ss[.nnnnnnn] For more information, see the "Backward Compatibility for Down-level Clients" section of Using Date and Time Data.
Range	00:00:00.0000000 through 23:59:59.9999999
Element ranges	hh is two digits, ranging from 0 to 23, that represent the hour. mm is two digits, ranging from 0 to 59, that represent the minute. ss is two digits, ranging from 0 to 59, that represent the second. n* is zero to seven digits, ranging from 0 to 9999999, that represent the fractional seconds.
Character length	8 positions minimum (hh:mm:ss) to 16 maximum (hh:mm:ss.nnnnnnn)
Precision, scale (user specifies scale only)	Specified scaleResult (precision, scale)Column length (bytes)Fractional seconds precision time(16,7)57 time(0)(8,0)30-2 time(1)(10,1)30-2 time(2)(11,2)30-2 time(3)(12,3)43-4 time(4)(13,4)43-4 time(5)(14,5)55-7 time(6)(15,6)55-7 time(7)(16,7)55-7

Storage size	5 bytes, fixed, is the default with the default of 100ns fractional second precision.
Accuracy	100 nanoseconds
Default value	00:00:00 This value is used for the appended time part for implicit conversion from date to datetime2 or datetimeoffset.
User-defined fractional second precision	Yes
Time zone offset aware and preservation	No
Daylight saving aware	No

Supported String Literal Formats for time

The following table shows the valid string literal formats for the time data type.

SQL Server	Description
hh:mm[:ss][:fractional seconds][AM][PM] hh:mm[:ss][.fractional seconds][AM][PM] hhAM[PM] hh AM[PM]	The hour value of 0 represents the hour after midnight (AM), regardless of whether AM is specified. PM cannot be specified when the hour equals 0. Hour values from 01 through 11 represent the hours before noon if neither AM nor PM is specified. The values represent the hours before noon when AM is specified. The values represent hours after noon if PM is specified. The hour value 12 represents the hour that starts at noon if neither AM nor PM is specified. If AM is specified, the value represents the hour that starts at midnight. If PM is specified, the value represents the hour that starts at noon. For example, 12:01 is 1 minute after noon, as is 12:01 PM; and 12:01 AM is one minute after midnight. Specifying 12:01 AM is the same as specifying 00:01 or 00:01 AM. Hour values from 13 through 23 represent hours after noon if AM or PM is not specified. The values also represent the hours after noon when PM is specified. AM cannot be specified when the hour value is from 13 through 23. An hour value of 24 is not valid. To represent midnight, use 12:00 AM or 00:00.

	Milliseconds can be preceded by either a colon (:) or a period (.). If a colon is used, the number means thousandths-of-a-second. If a period is used, a single digit means tenths-of-a-second, two digits mean hundredths-of-a-second, and three digits mean thousandths-of-a-second. For example, 12:30:20:1 indicates 20 and one-thousandth seconds past 12:30; 12:30:20.1 indicates 20 and one-tenth seconds past 12:30.

ISO 8601	Notes
hh:mm:ss hh:mm[:ss][.fractional seconds]	• hh is two digits, ranging from 0 to 14, that represent the number of hours in the time zone offset. • mm is two digits, ranging from 0 to 59, that represent the number of additional minutes in the time zone offset.

ODBC	Notes
{t 'hh:mm:ss[.fractional seconds]'}	ODBC API specific. Functions in SQL Server 2008 as in SQL Server 2005.

time Compliance with ANSI and ISO 8601 Standards

Using hour 24 to represent midnight and leap second over 59 as defined by ISO 8601 (5.3.2 and 5.3) are not supported to be backward compatible and consistent with the existing date and time types. They are not defined by SQL standard 2003.

The default string literal format (used for down-level client) will align with the SQL standard form, which is defined as hh:mm:ss[.nnnnnnn]. This format resembles the ISO 8601 definition for TIME excluding fractional seconds.

Examples

A. Comparing date and time Data Types

The following example compares the results of casting a string to each date and time data type.

```
SELECT
     CAST('2007-05-08 12:35:29. 1234567 +12:15' AS time(7)) AS 'time'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS date) AS 'date'
    ,CAST('2007-05-08 12:35:29.123' AS smalldatetime) AS
        'smalldatetime'
    ,CAST('2007-05-08 12:35:29.123' AS datetime) AS 'datetime'
    ,CAST('2007-05-08 12:35:29. 1234567 +12:15' AS datetime2(7)) AS
        'datetime2'
    ,CAST('2007-05-08 12:35:29.1234567 +12:15' AS datetimeoffset(7)) AS
        'datetimeoffset';
```

Data type	Output
time	12:35:29. 1234567
date	2007-05-08
smalldatetime	2007-05-08 12:35:00
datetime	2007-05-08 12:35:29.123
datetime2	2007-05-08 12:35:29. 1234567
datetimeoffset	2007-05-08 12:35:29.1234567 +12:15

B. Inserting Valid Time String Literals into a time(7) Column

The following table lists different string literals that can be inserted into a column of data type time(7) with the values that are then stored in that column.

String literal format type	Inserted string literal	time(7) value that is stored	Description
SQL Server	'01:01:01:123AM'	01:01:01.1230000	When a colon (:) comes before fractional seconds precision, scale cannot exceed three positions or an error will be raised.
SQL Server	'01:01:01.1234567 AM'	01:01:01.1234567	When AM or PM is specified, the time is stored in 24-hour format without the literal AM or PM
SQL Server	'01:01:01.1234567 PM'	13:01:01.1234567	When AM or PM is specified, the time is stored in 24-hour format without the literal AM or PM
SQL Server	'01:01:01.1234567PM'	13:01:01.1234567	A space before AM or PM is optional.
SQL Server	'01AM'	01:00:00.0000000	When only the hour is specified, all other values are 0.
SQL Server	'01 AM'	01:00:00.0000000	A space before AM or PM is optional.
SQL Server	'01:01:01'	01:01:01.0000000	When fractional seconds precision is not

			specified, each position that is defined by the data type is 0.
ISO 8601	'01:01:01.1234567'	01:01:01.1234567	To comply with ISO 8601, use 24-hour format, not AM or PM.
ISO 8601	'01:01:01.1234567 +01:01'	01:01:01.1234567	The optional time zone difference (TZD) is allowed in the input but is not stored.

C. Inserting Time String Literal into Columns of Each date and time Date Type

In the following table the first column shows a time string literal to be inserted into a database table column of the date or time data type shown in the second column. The third column shows the value that will be stored in the database table column.

Inserted string literal	Column data type	Value that is stored in column	Description
'12:12:12.1234567'	time(7)	12:12:12.1234567	If the fractional seconds precision exceeds the value specified for the column, the string will be truncated without error.
'2007-05-07'	date	NULL	Any time value will cause the INSERT statement to fail.
'12:12:12'	smalldatetime	1900-01-01 12:12:00	Any fractional seconds precision value will cause the INSERT statement to fail.
'12:12:12.123'	datetime	1900-01-01 12:12:12.123	Any second precision longer than three positions will cause the INSERT statement to fail.
'12:12:12.1234567'	datetime2(7)	1900-01-01 12:12:12.1234567	If the fractional seconds precision exceeds the value specified for the column, the string will be truncated without error.
'12:12:12.1234567'	datetimeoffset(7)	1900-01-01 12:12:12.1234567	If the fractional seconds precision exceeds the value specified for the

		+00:00	column, the string will be truncated without error.

6.7. hierarchyid

The hierarchyid data type is a variable length, system data type. Use hierarchyid to represent position in a hierarchy. A column of type hierarchyid does not automatically represent a tree. It is up to the application to generate and assign hierarchyid values in such a way that the desired relationship between rows is reflected in the values.

A value of the hierarchyid data type represents a position in a tree hierarchy. Values for hierarchyid have the following properties:

- Extremely compact

  The average number of bits that are required to represent a node in a tree with *n* nodes depends on the average fanout (the average number of children of a node). For small fanouts (0-7), the size is about $6*\log_A n$ bits, where A is the average fanout. A node in an organizational hierarchy of 100,000 people with an average fanout of 6 levels takes about 38 bits. This is rounded up to 40 bits, or 5 bytes, for storage.

- Comparison is in depth-first order

  Given two hierarchyid values **a** and **b**, **a<b** means a comes before b in a depth-first traversal of the tree. Indexes on hierarchyid data types are in depth-first order, and nodes close to each other in a depth-first traversal are stored near each other. For example, the children of a record are stored adjacent to that record. For more information, see Using hierarchyid Data Types (Database Engine).

- Support for arbitrary insertions and deletions

  By using the GetDescendant method, it is always possible to generate a sibling to the right of any given node, to the left of any given node, or between any two siblings. The comparison property is maintained when an arbitrary number of nodes is inserted or deleted from the hierarchy. Most insertions and deletions preserve the compactness property. However, insertions between two nodes will produce hierarchyid values with a slightly less compact representation.

- The encoding used in the hierarchyid type is limited to 892 bytes. Consequently, nodes which have too many levels in their representation to fit into 892 bytes cannot be represented by the hierarchyid type.

The hierarchyid type is available to CLR clients as the SqlHierarchyId data type.

Remarks

The hierarchyid type logically encodes information about a single node in a hierarchy tree by encoding the path from the root of the tree to the node. Such a path is logically represented as a sequence of node labels of all children visited after the root. A slash starts the representation, and a path that only

visits the root is represented by a single slash. For levels underneath the root, each label is encoded as a sequence of integers separated by dots. Comparison between children is performed by comparing the integer sequences separated by dots in dictionary order. Each level is followed by a slash. Therefore a slash separates parents from their children. For example, the following are valid hierarchyid paths of lengths 1, 2, 2, 3, and 3 levels respectively:

- /
- /1/
- /0.3.-7/
- /1/3/
- /0.1/0.2/

Nodes can be inserted in any location. Nodes inserted after **/1/2/** but before **/1/3/** can be represented as **/1/2.5/**. Nodes inserted before 0 have the logical representation as a negative number. For example, a node that comes before **/1/1/** can be represented as **/1/-1/**. Nodes cannot have leading zeros. For example, **/1/1.1/** is valid, but **/1/1.01/** is not valid. To prevent errors, insert nodes by using the GetDescendant method.

Data Type Conversion

The hierarchyid data type can be converted to other data types as follows:

- Use the ToString() method to convert the hierarchyid value to the logical representation as a nvarchar(4000) data type.
- Use Read () and Write () to convert hierarchyid to varbinary.
- Conversion from hierarchyid to XML is not supported. To transmit hierarchyid parameters through SOAP first cast them as strings. A query with the FOR XML clause will fail on a table with hierarchyid unless the column is first converted to a character data type.

Upgrading Databases

When a database is upgraded to SQL Server 2008, the new assembly and the hierarchyid data type will automatically be installed. Upgrade advisor rules detect any user type or assemblies with conflicting names. The upgrade advisor will advise renaming of any conflicting assembly, and either renaming any conflicting type, or using two-part names in the code to refer to that preexisting user type.

If a database upgrade detects a user assembly with conflicting name, it will automatically rename that assembly and put the database into suspect mode.

If a user type with conflicting name exists during the upgrade, no special steps are taken. After the upgrade, both the old user type, and the new system type, will exist. The user type will be available only through two-part names.

Using hierarchyid Columns in Replicated Tables

Columns of type hierarchyid can be used on any replicated table. The requirements for your application depend on whether replication is one directional or bidirectional, and on the versions of SQL Server that are used.

One-Directional Replication

One-directional replication includes snapshot replication, transactional replication, and merge replication in which changes are not made at the Subscriber. How hierachyid columns work with one directional replication depends on the version of SQL Server the Subscriber is running.

- A SQL Server 2008 Publisher can replicate hierachyid columns to a SQL Server 2008 Subscriber without any special considerations.
- A SQL Server 2008 Publisher must convert hierarchyid columns to replicate them to a Subscriber that is running SQL Server Compact 3.5 SP2 or an earlier version of SQL Server. SQL Server Compact 3.5 SP2 and earlier versions of SQL Server do not support hierarchyid columns. If you are using one of these versions, you can still replicate data to a Subscriber. To do this, you must set a schema option or the publication compatibility level (for merge replication) so the column can be converted to a compatible data type. For more information, see Using Multiple Versions of SQL Server in a Replication Topology.

Column filtering is supported in both of these scenarios. This includes filtering out hierarchyid columns. Row filtering is supported as long as the filter does not include a hierarchyid column.

Bi-Directional Replication

Bi-directional replication includes transactional replication with updating subscriptions, peer-to-peer transactional replication, and merge replication in which changes are made at the Subscriber. Replication lets you configure a table with hierarchyid columns for bi-directional replication. Note the following requirements and considerations.

- The Publisher and all Subscribers must be running SQL Server 2008.
- Replication replicates the data as bytes and does not perform any validation to assure the integrity of the hierarchy.
- The hierarchy of the changes that were made at the source (Subscriber or Publisher) is not maintained when they replicate to the destination.
- The hash values for hierarchyid columns are specific to the database in which they are generated. Therefore, the same value could be generated on the Publisher and Subscriber, but it could be applied to different rows. Replication does not check for this condition, and there is no built-in way to partition hierarchyid column values as there is for IDENTITY columns. Applications must use constraints or other mechanisms to avoid such undetected conflicts.
- It is possible that rows that are inserted on the Subscriber will be orphaned. The parent row of the inserted row might have been deleted at the Publisher. This results in an undetected conflict when the row from the Subscriber is inserted at the Publisher.
- Column filters cannot filter out non-nullable hierarchyid columns: inserts from the Subscriber will fail because there is no default value for the hierarchyid column on the Publisher.
- Row filtering is supported as long as the filter does not include a hierarchyid column.

6.7.1. hierarchyid Data Type Method Reference

SQL Server 2008 supports a set of methods for the Transact-SQL **hierarchyid** data type and the common language runtime (CLR) SqlHierarchyId class. For more information, see Working with hierarchyid Data. The topics in this section provide the syntax and examples for each **hierarchyid** data type method.

In This Section

GetAncestor (Database Engine)

GetDescendant (Database Engine)
GetLevel (Database Engine)
GetRoot (Database Engine)
IsDescendantOf (Database Engine)
Parse (Database Engine)
Read (Database Engine)
GetReparentedValue (Database Engine)
ToString (Database Engine)
Write (Database Engine)

6.8. Numeric Types

SQL Server supports the following numeric types.

6.8.1. decimal and numeric

Numeric data types that have fixed precision and scale.

decimal[**(***p*[**,***s*] **)**] and numeric[**(***p*[**,***s*] **)**]

> Fixed precision and scale numbers. When maximum precision is used, valid values are from - 10^38 +1 through 10^38 - 1. The ISO synonyms for decimal are dec and dec(*p*, *s*). numeric is functionally equivalent to decimal.

p (precision)

> The maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point. The precision must be a value from 1 through the maximum precision of 38. The default precision is 18.

s (scale)

> The maximum number of decimal digits that can be stored to the right of the decimal point. Scale must be a value from 0 through *p*. Scale can be specified only if precision is specified. The default scale is 0; therefore, 0 <= *s* <= *p*. Maximum storage sizes vary, based on the precision.

Precision	Storage bytes
1 - 9	5
10-19	9
20-28	13
29-38	17

6.8.2. float and real

Approximate-number data types for use with floating point numeric data. Floating point data is approximate; therefore, not all values in the data type range can be represented exactly.

Note
The ISO synonym for real is float(24).

Data type	Range	Storage
`float`	- 1.79E+308 to -2.23E-308, 0 and 2.23E-308 to 1.79E+308	Depends on the value of *n*
`real`	- 3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 to 3.40E + 38	4 Bytes

Syntax

float [**(***n***)**]

Where *n* is the number of bits that are used to store the mantissa of the float number in scientific notation and, therefore, dictates the precision and storage size. If *n* is specified, it must be a value between **1** and **53**. The default value of *n* is **53**.

*n*value	Precision	Storage size
1-24	7 digits	4 bytes
25-53	15 digits	8 bytes

Note
SQL Server treats *n* as one of two possible values. If **1**<=n<=**24**, *n* is treated as **24**. If **25**<=n<=**53**, *n* is treated as **53**.

The SQL Server float[(n)] data type complies with the ISO standard for all values of *n* from **1** through **53**. The synonym for double precision is float(53).

6.8.3. int, bigint, smallint, and tinyint

Exact-number data types that use integer data.

Data type	Range	Storage
`bigint`	-2^63 (-9,223,372,036,854,775,808) to 2^63-1	8 Bytes

	(9,223,372,036,854,775,807)	
int	-2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	4 Bytes
smallint	-2^15 (-32,768) to 2^15-1 (32,767)	2 Bytes
tinyint	0 to 255	1 Byte

Remarks

The int data type is the primary integer data type in SQL Server. The bigint data type is intended for use when integer values might exceed the range that is supported by the int data type.

bigint fits between smallmoney and int in the data type precedence chart.

Functions return bigint only if the parameter expression is a bigint data type. SQL Server does not automatically promote other integer data types (tinyint, smallint, and int) to bigint.

Caution
When you use the +, -, *, /, or % arithmetic operators to perform implicit or explicit conversion of int, smallint, tinyint, or bigint constant values to the float, real, decimal or numeric data types, the rules that SQL Server applies when it calculates the data type and precision of the expression results differ depending on whether the query is autoparameterized or not.
Therefore, similar expressions in queries can sometimes produce different results. When a query is not autoparameterized, the constant value is first converted to numeric, whose precision is just large enough to hold the value of the constant, before converting to the specified data type. For example, the constant value 1 is converted to numeric (1, 0), and the constant value 250 is converted to numeric (3, 0).
When a query is autoparameterized, the constant value is always converted to numeric (10, 0) before converting to the final data type. When the / operator is involved, not only can the result type's precision differ among similar queries, but the result value can differ also. For example, the result value of an autoparameterized query that includes the expression SELECT CAST (1.0 / 7 AS float) will differ from the result value of the same query that is not autoparameterized, because the results of the autoparameterized query will be truncated to fit into the numeric (10, 0) data type. For more information about parameterized queries, see Simple Parameterization.

6.8.4. money and smallmoney

Data types that represent monetary or currency values.

Data type	Range	Storage
money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
smallmoney	- 214,748.3648 to 214,748.3647	4 bytes

Remarks

The money and smallmoney data types are accurate to a ten-thousandth of the monetary units that they represent.

6.9. rowversion

Is a data type that exposes automatically generated, unique binary numbers within a database. rowversion is generally used as a mechanism for version-stamping table rows. The storage size is 8 bytes. The rowversion data type is just an incrementing number and does not preserve a date or a time. To record a date or time, use a datetime2 data type.

Remarks

Each database has a counter that is incremented for each insert or update operation that is performed on a table that contains a rowversion column within the database. This counter is the database rowversion. This tracks a relative time within a database, not an actual time that can be associated with a clock. A table can have only one rowversion column. Every time that a row with a rowversion column is modified or inserted, the incremented database rowversion value is inserted in the rowversion column. This property makes a rowversion column a poor candidate for keys, especially primary keys. Any update made to the row changes the rowversion value and, therefore, changes the key value. If the column is in a primary key, the old key value is no longer valid, and foreign keys referencing the old value are no longer valid. If the table is referenced in a dynamic cursor, all updates change the position of the rows in the cursor. If the column is in an index key, all updates to the data row also generate updates of the index.

timestamp is the synonym for the rowversion data type and is subject to the behavior of data type synonyms. In DDL statements, use rowversion instead of timestamp wherever possible. For more information, see Data Type Synonyms.

The Transact-SQL timestamp data type is different from the timestamp data type defined in the ISO standard.

Note

The timestamp syntax is deprecated. This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

In a CREATE TABLE or ALTER TABLE statement, you do not have to specify a column name for the timestamp data type, for example:

```
CREATE TABLE ExampleTable (PriKey int PRIMARY KEY, timestamp);
```

If you do not specify a column name, the SQL Server Database Engine generates the timestamp column name; however, the rowversion synonym does not follow this behavior. When you use rowversion, you must specify a column name, for example:

```
CREATE TABLE ExampleTable2 (PriKey int PRIMARY KEY, VerCol rowversion) ;
```

Note

Duplicate rowversion values can be generated by using the SELECT INTO statement in which a rowversion column is in the SELECT list. We do not recommend using rowversion in this manner.

A nonnullable rowversion column is semantically equivalent to a binary(8) column. A nullable rowversion column is semantically equivalent to a varbinary(8) column.

You can use the rowversion column of a row to easily determine whether any value in the row has changed since the last time it was read. If any change is made to the row, the rowversion value is updated. If no change is made to the row, the rowversion value is the same as when it was previously read. To return the current rowversion value for a database, use @@DBTS.

You can add a rowversion column to a table to help maintain the integrity of the database when multiple users are updating rows at the same time. You may also want to know how many rows and which rows were updated without re-querying the table.

For example, assume that you create a table named MyTest. You populate some data in the table by running the following Transact-SQL statements.

```
CREATE TABLE MyTest (myKey int PRIMARY KEY
    ,myValue int, RV rowversion);
GO
INSERT INTO MyTest (myKey, myValue) VALUES (1, 0);
GO
INSERT INTO MyTest (myKey, myValue) VALUES (2, 0);
GO
```

You can then use the following sample Transact-SQL statements to implement optimistic concurrency control on the MyTest table during the update.

```
DECLARE @t TABLE (myKey int);
UPDATE MyTest
SET myValue = 2
    OUTPUT inserted.myKey INTO @t(myKey)
WHERE myKey = 1
    AND RV = myValue;
IF (SELECT COUNT(*) FROM @t) = 0
    BEGIN
        RAISERROR ('error changing row with myKey = %d'
```

```
            ,16 -- Severity.
            ,1 -- State
            ,1) -- myKey that was changed
    END;
```

myValue is the rowversion column value for the row that indicates the last time that you read the row. This value must be replaced by the actual rowversion value. An example of the actual rowversion value is 0x00000000000007D3.

You can also put the sample Transact-SQL statements into a transaction. By querying the @t variable in the scope of the transaction, you can retrieve the updated myKey column of the table without requerying the MyTest table.

The following is the same example using the timestamp syntax:

```
CREATE TABLE MyTest2 (myKey int PRIMARY KEY
    ,myValue int, TS timestamp);
GO
INSERT INTO MyTest2 (myKey, myValue) VALUES (1, 0);
GO
INSERT INTO MyTest2 (myKey, myValue) VALUES (2, 0);
GO
DECLARE @t TABLE (myKey int);
UPDATE MyTest2
SET myValue = 2
    OUTPUT inserted.myKey INTO @t(myKey)
WHERE myKey = 1
    AND TS = myValue;
IF (SELECT COUNT(*) FROM @t) = 0
    BEGIN
        RAISERROR ('error changing row with myKey = %d'
            ,16 -- Severity.
            ,1 -- State
            ,1) -- myKey that was changed
    END;
```

6.10. Spatial Types

SQL Server supports the following spatial types.

6.10.1. geography

The geography spatial data type, geography, is implemented as a .NET common language runtime (CLR) data type in SQL Server. This type represents data in a round-earth coordinate system. The SQL Server geography data type stores ellipsoidal (round-earth) data, such as GPS latitude and longitude coordinates.

SQL Server 2008 supports a set of methods for the **geography** spatial data type. These methods include methods on geography that are defined by the Open Geospatial Consortium (OGC) standard and a set of Microsoft extensions to that standard.

For more information on geography spatial Data Type methods, see the geography Data Type Method Reference.

Registering the geography Type

The geography type is predefined and available in each database. You can create table columns of type geography and operate on geography data in the same manner as you would use other system-supplied types.

Examples

A. Showing how to add and query geography data

The following examples show how to add and query geography data. The first example creates a table with an identity column and a geography column, GeogCol1. A third column renders the geography column into its Open Geospatial Consortium (OGC) Well-Known Text (WKT) representation, and uses the STAsText() method. Two rows are then inserted: one row contains a LineString instance of geography, and one row contains a Polygon instance.

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

CREATE TABLE SpatialTable
    ( id int IDENTITY (1,1),
    GeogCol1 geography,
    GeogCol2 AS GeogCol1.STAsText() );
GO

INSERT INTO SpatialTable (GeogCol1)
VALUES (geography::STGeomFromText('LINESTRING(-122.360 47.656, -122.343 47.65
6 )', 4326));

INSERT INTO SpatialTable (GeogCol1)
VALUES (geography::STGeomFromText('POLYGON((-122.358 47.653 , -122.348 47.649
, -122.348 47.658, -122.358 47.658, -122.358 47.653))', 4326));
GO
```

B. Returning the intersection of two geography instances

The following example uses the STIntersection() method to return the points where the two previously inserted geography instances intersect.

```
DECLARE @geog1 geography;
DECLARE @geog2 geography;
DECLARE @result geography;
```

```
SELECT @geog1 = GeogCol1 FROM SpatialTable WHERE id = 1;
SELECT @geog2 = GeogCol1 FROM SpatialTable WHERE id = 2;
SELECT @result = @geog1.STIntersection(@geog2);
SELECT @result.STAsText();
```

6.10.2. geometry

The planar spatial data type, geometry, is implemented as a common language runtime (CLR) data type in SQL Server. This type represents data in a Euclidean (flat) coordinate system.

SQL Server 2008 supports a set of methods for the **geometry** spatial data type. These methods include methods on geometry that are defined by the Open Geospatial Consortium (OGC) standard and a set of Microsoft extensions to that standard.

For more information on geometry spatial Data Type methods, see the geometry Data Type Method Reference.

Registering the geometry Type

The geometry type is predefined and available in each database. You can create table columns of type geometry and operate on geometry data in the same manner as you would use other CLR types.

Examples

The following two examples show how to add and query geometry data. The first example creates a table with an identity column and a geometry column, GeomCol1. A third column renders the geometry column into its Open Geospatial Consortium (OGC) Well-Known Text (WKT) representation, and uses the STAsText() method. Two rows are then inserted: one row contains a LineString instance of geometry, and one row contains a Polygon instance.

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

CREATE TABLE SpatialTable
    ( id int IDENTITY (1,1),
    GeomCol1 geometry,
    GeomCol2 AS GeomCol1.STAsText() );
GO

INSERT INTO SpatialTable (GeomCol1)
VALUES (geometry::STGeomFromText('LINESTRING (100 100, 20 180, 180 180)', 0))
;

INSERT INTO SpatialTable (GeomCol1)
VALUES (geometry::STGeomFromText('POLYGON ((0 0, 150 0, 150 150, 0 150, 0 0))
', 0));
GO
```

The second example uses the STIntersection() method to return the points where the two previously inserted geometry instances intersect.

```
DECLARE @geom1 geometry;
DECLARE @geom2 geometry;
DECLARE @result geometry;

SELECT @geom1 = GeomCol1 FROM SpatialTable WHERE id = 1;
SELECT @geom2 = GeomCol1 FROM SpatialTable WHERE id = 2;
SELECT @result = @geom1.STIntersection(@geom2);
SELECT @result.STAsText();
```

6.11. String and Binary Types

SQL Server supports the following string and binary types.

6.11.1. binary and varbinary

Binary data types of either fixed length or variable length.

binary [(*n*)]

> Fixed-length binary data with a length of *n* bytes, where *n* is a value from 1 through 8,000. The storage size is *n* bytes.

varbinary [(*n* | max)]

> Variable-length binary data. *n* can be a value from 1 through 8,000. max indicates that the maximum storage size is 2^31-1 bytes. The storage size is the actual length of the data entered + 2 bytes. The data that is entered can be 0 bytes in length. The ANSI SQL synonym for varbinary is **binary varying**.

Remarks

When *n* is not specified in a data definition or variable declaration statement, the default length is 1. When *n*is not specified with the CAST function, the default length is 30.

Use binary when the sizes of the column data entries are consistent.

Use varbinary when the sizes of the column data entries vary considerably.

Use varbinary(max) when the column data entries exceed 8,000 bytes.

6.11.2. char and varchar

Are character data types of either fixed length or variable length.

char [(*n*)]

> Fixed-length, non-Unicode character data with a length of *n* bytes. *n* must be a value from 1 through 8,000. The storage size is *n* bytes. The ISO synonym for char is character.

varchar [(*n* | max)]

> Variable-length, non-Unicode character data. *n* can be a value from 1 through 8,000. max indicates that the maximum storage size is 2^31-1 bytes. The storage size is the actual length of data entered + 2 bytes. The data entered can be 0 characters in length. The ISO synonyms for varchar are char varying or character varying.

Remarks

When *n* is not specified in a data definition or variable declaration statement, the default length is 1. When *n* is not specified when using the CAST and CONVERT functions, the default length is 30.

Objects that use char or varchar are assigned the default collation of the database, unless a specific collation is assigned using the COLLATE clause. The collation controls the code page that is used to store the character data.

If you have sites that support multiple languages, consider using the Unicode nchar or nvarchar data types to minimize character conversion issues. If you use char or varchar, we recommend the following:

- Use char when the sizes of the column data entries are consistent.
- Use varchar when the sizes of the column data entries vary considerably.
- Use varchar(max) when the sizes of the column data entries vary considerably, and the size might exceed 8,000 bytes.

If SET ANSI_PADDING is OFF when either CREATE TABLE or ALTER TABLE is executed, a char column that is defined as NULL is handled as varchar.

When the collation code page uses double-byte characters, the storage size is still *n* bytes. Depending on the character string, the storage size of *n* bytes can be less than *n* characters.

Examples

A. Showing the default value of n when used in variable declaration.

The following example shows the default value of *n* is 1 for the char and varchar data types when they are used in variable declaration.

```
DECLARE @myVariable AS varchar
DECLARE @myNextVariable AS char
SET @myVariable = 'abc'
SET @myNextVariable = 'abc'
--The following returns 1
```

```
SELECT DATALENGTH(@myVariable), DATALENGTH(@myNextVariable);
GO
```

B. Showing the default value of n when varchar is used with CAST and CONVERT.

The following example shows that the default value of *n* is 30 when the char or varchar data types are used with the CAST and CONVERT functions.

```
DECLARE @myVariable AS varchar(40)
SET @myVariable = 'This string is longer than thirty characters'
SELECT CAST(@myVariable AS varchar)
SELECT DATALENGTH(CAST(@myVariable AS varchar)) AS 'VarcharDefaultLength';
SELECT CONVERT(char, @myVariable)
SELECT DATALENGTH(CONVERT(char, @myVariable)) AS 'VarcharDefaultLength';
```

6.11.3. nchar and nvarchar

Character data types that are either fixed-length, nchar, or variable-length, nvarchar, Unicode data and use the UNICODE UCS-2 character set.

nchar [(n)]

Fixed-length Unicode character data of *n* characters. *n* must be a value from 1 through 4,000. The storage size is two times *n* bytes. The ISO synonyms for nchar are national char and national character.

nvarchar [(n | max)]

Variable-length Unicode character data. *n*can be a value from 1 through 4,000. max indicates that the maximum storage size is 2^31-1 bytes. The storage size, in bytes, is two times the number of characters entered + 2 bytes. The data entered can be 0 characters in length. The ISO synonyms for nvarchar are national char varying and national character varying.

Remarks

When *n* is not specified in a data definition or variable declaration statement, the default length is 1. When *n* is not specified with the CAST function, the default length is 30.

Use nchar when the sizes of the column data entries are probably going to be similar.

Use nvarchar when the sizes of the column data entries are probably going to vary considerably.

sysname is a system-supplied user-defined data type that is functionally equivalent to nvarchar(128), except that it is not nullable. sysname is used to reference database object names.

Objects that use nchar or nvarchar are assigned the default collation of the database unless a specific collation is assigned using the COLLATE clause.

SET ANSI_PADDING is always ON for nchar and nvarchar. SET ANSI_PADDING OFF does not apply to the nchar or nvarchar data types.

6.11.4. ntext, text, and image

Important

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

Fixed and variable-length data types for storing large non-Unicode and Unicode character and binary data. Unicode data uses the UNICODE UCS-2 character set.

ntext

Variable-length Unicode data with a maximum length of 2^30 - 1 (1,073,741,823) characters. Storage size, in bytes, is two times the number of characters entered. The ISO synonym for ntext is national text.

text

Variable-length non-Unicode data in the code page of the server and with a maximum length of 2^31-1 (2,147,483,647) characters. When the server code page uses double-byte characters, the storage is still 2,147,483,647 bytes. Depending on the character string, the storage size may be less than 2,147,483,647 bytes.

image

Variable-length binary data from 0 through 2^31-1 (2,147,483,647) bytes.

Remarks

The following functions and statements can be used with ntext, text, or image data.

Functions	Statements
DATALENGTH	**READTEXT**
PATINDEX	**SET TEXTSIZE**
SUBSTRING	**UPDATETEXT**
TEXTPTR	**WRITETEXT**
TEXTVALID	

6.12. sql_variant

A data type that stores values of various SQL Server-supported data types.

Syntax

```
sql_variant
```

Remarks

sql_variant can be used in columns, parameters, variables, and the return values of user-defined functions. sql_variant enables these database objects to support values of other data types.

A column of type sql_variant may contain rows of different data types. For example, a column defined as sql_variant can store int, binary, and char values. The following table lists the types of values that cannot be stored by using sql_variant:

`varchar(max)`	`varbinary(max)`
`nvarchar(max)`	`xml`
`text`	`ntext`
`image`	`timestamp`
`sql_variant`	`geography`
`hierarchyid`	`geometry`
`User-defined types`	

sql_variant can have a maximum length of 8016 bytes. This includes both the base type information and the base type value. The maximum length of the actual base type value is 8,000 bytes.

A sql_variant data type must first be cast to its base data type value before participating in operations such as addition and subtraction.

sql_variant can be assigned a default value. This data type can also have NULL as its underlying value, but the NULL values will not have an associated base type. Also, sql_variant cannot have another sql_variant as its base type.

A unique, primary, or foreign key may include columns of type sql_variant, but the total length of the data values that make up the key of a specific row should not be more than the maximum length of an index. This is 900 bytes.

A table can have any number of sql_variant columns.

sql_variant cannot be used in CONTAINSTABLE and FREETEXTTABLE.

ODBC does not fully support sql_variant. Therefore, queries of sql_variant columns are returned as binary data when you use Microsoft OLE DB Provider for ODBC (MSDASQL). For example, a sql_variant column that contains the character string data 'PS2091' is returned as 0x505332303931.

Comparing sql_variant Values

The sql_variant data type belongs to the top of the data type hierarchy list for conversion. For sql_variant comparisons, the SQL Server data type hierarchy order is grouped into data type families.

Data type hierarchy	Data type family
`sql_variant`	sql_variant
`datetime2`	Date and time
`datetimeoffset`	Date and time
`datetime`	Date and time
`smalldatetime`	Date and time
`date`	Date and time
`time`	Date and time
`float`	Approximate numeric
`real`	Approximate numeric
`decimal`	Exact numeric
`money`	Exact numeric
`smallmoney`	Exact numeric
`bigint`	Exact numeric
`int`	Exact numeric
`smallint`	Exact numeric
`tinyint`	Exact numeric

bit	Exact numeric
nvarchar	Unicode
nchar	Unicode
varchar	Unicode
char	Unicode
varbinary	Binary
binary	Binary
uniqueidentifier	Uniqueidentifier

The following rules apply to sql_variant comparisons:

- When sql_variant values of different base data types are compared and the base data types are in different data type families, the value whose data type family is higher in the hierarchy chart is considered the greater of the two values.
- When sql_variant values of different base data types are compared and the base data types are in the same data type family, the value whose base data type is lower in the hierarchy chart is implicitly converted to the other data type and the comparison is then made.
- When sql_variant values of the char, varchar, nchar, or nvarchar data types are compared, their collations are first compared based on the following criteria: LCID, LCID version, comparison flags, and sort ID. Each of these criteria are compared as integer values, and in the order listed. If all of these criteria are equal, then the actual string values are compared according to the collation.

6.13. table

Is a special data type that can be used to store a result set for processing at a later time. table is primarily used for temporary storage of a set of rows returned as the result set of a table-valued function.

Note

To declare variables of type table, use DECLARE @local_variable.

Syntax

```
table_type_definition ::=
    TABLE ( { column_definition | table_constraint } [ ,...n ] )

column_definition ::=
```

```
    column_name scalar_data_type
    [ COLLATE collation_definition ]
    [ [ DEFAULT constant_expression ] | IDENTITY [ ( seed , increment ) ] ]
    [ ROWGUIDCOL ]
    [ column_constraint ] [ ...n ]

column_constraint ::=
    { [ NULL | NOT NULL ]
    | [ PRIMARY KEY | UNIQUE ]
    | CHECK ( logical_expression )
    }

table_constraint ::=
     { { PRIMARY KEY | UNIQUE } ( column_name [ ,...n ] )
     | CHECK ( logical_expression )
     }
```

Arguments

table_type_definition

Is the same subset of information that is used to define a table in CREATE TABLE. The table declaration includes column definitions, names, data types, and constraints. The only constraint types allowed are PRIMARY KEY, UNIQUE KEY, and NULL.

For more information about the syntax, see CREATE TABLE, CREATE FUNCTION, and DECLARE @local_variable.

collation_definition

Is the collation of the column that is made up of a Microsoft Windows locale and a comparison style, a Windows locale and the binary notation, or a Microsoft SQL Server collation. If *collation_definition* is not specified, the column inherits the collation of the current database. Or if the column is defined as a common language runtime (CLR) user-defined type, the column inherits the collation of the user-defined type.

Remarks

Functions and variables can be declared to be of type table. table variables can be used in functions, stored procedures, and batches.

Important

Queries that modify table variables do not generate parallel query execution plans. Performance can be affected when very large table variables, or table variables in complex queries, are modified. In these situations, consider using temporary tables instead. For more information, see CREATE TABLE. Queries that read table variables without modifying them can still be parallelized.

table variables provide the following benefits:

- A table variable behaves like a local variable. It has a well-defined scope. This is the function, stored procedure, or batch that it is declared in.

  Within its scope, a table variable can be used like a regular table. It may be applied anywhere a table or table expression is used in SELECT, INSERT, UPDATE, and DELETE statements. However, table cannot be used in the following statement:

  ```
  SELECT select_list INTO table_variable
  ```

  table variables are automatically cleaned up at the end of the function, stored procedure, or batch in which they are defined.

- CHECK constraints, DEFAULT values and computed columns in the table type declaration cannot call user-defined functions.
- table variables used in stored procedures cause fewer recompilations of the stored procedures than when temporary tables are used.
- Transactions involving table variables last only for the duration of an update on the table variable. Therefore, table variables require less locking and logging resources.

Indexes cannot be created explicitly on table variables, and no statistics are kept on table variables. In some cases, performance may improve by using temporary tables instead, which support indexes and statistics. For more information about temporary tables, see CREATE TABLE.

table variables can be referenced by name in the FROM clause of a batch, as shown the following example:

```
SELECT Employee_ID, Department_ID FROM @MyTableVar
```

Outside a FROM clause, table variables must be referenced by using an alias, as shown in the following example:

```
SELECT EmployeeID, DepartmentID
FROM @MyTableVar m
JOIN Employee on (m.EmployeeID =Employee.EmployeeID AND
   m.DepartmentID = Employee.DepartmentID)
```

Assignment operation between table variables is not supported. Also, because table variables have limited scope and are not part of the persistent database, they are not affected by transaction rollbacks.

6.14. uniqueidentifier

Is a 16-byte GUID.

Remarks

A column or local variable of uniqueidentifier data type can be initialized to a value in the following ways:

- By using the NEWID function.

- By converting from a string constant in the form *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*, in which each *x* is a hexadecimal digit in the range 0-9 or a-f. For example, 6F9619FF-8B86-D011-B42D-00C04FC964FF is a valid uniqueidentifier value.

Comparison operators can be used with uniqueidentifier values. However, ordering is not implemented by comparing the bit patterns of the two values. The only operations that can be performed against a uniqueidentifier value are comparisons (=, <>, <, >, <=, >=) and checking for NULL (IS NULL and IS NOT NULL). No other arithmetic operators can be used. All column constraints and properties, except IDENTITY, can be used on the uniqueidentifier data type.

Merge replication and transactional replication with updating subscriptions use uniqueidentifier columns to guarantee that rows are uniquely identified across multiple copies of the table.

Converting uniqueidentifier Data

The uniqueidentifier type is considered a character type for the purposes of conversion from a character expression, and therefore is subject to the truncation rules for converting to a character type. That is, when character expressions are converted to a character data type of a different size, values that are too long for the new data type are truncated. See the Examples section.

Examples

The following example converts a uniqueidentifier value to a char data type.

```
DECLARE @myid uniqueidentifier = NEWID();
SELECT CONVERT(char(255), @myid) AS 'char';
```

The following example demonstrates the truncation of data when the value is too long for the data type being converted to. Because the uniqueidentifier type is limited to 36 characters, the characters that exceed that length are truncated.

```
DECLARE @ID nvarchar(max) = N'0E984725-C51C-4BF4-9960-E1C80E27ABA0wrong';
SELECT @ID, CONVERT(uniqueidentifier, @ID) AS TruncatedValue;
```

Here is the result set.

```
String                                       TruncatedValue
-------------------------------------------- ------------------------------------
0E984725-C51C-4BF4-9960-E1C80E27ABA0wrong    0E984725-C51C-4BF4-9960-E1C80E27ABA0

(1 row(s) affected)
```

6.15. xml

Is the data type that stores XML data. You can store xml instances in a column, or a variable of xml type. For more information, see Implementing XML in SQL Server.

Syntax

```
xml ( [ CONTENT | DOCUMENT ] xml_schema_collection )
```

Arguments

CONTENT

Restricts the xml instance to be a well-formed XML fragment. The XML data can contain multiple zero or more elements at the top level. Text nodes are also allowed at the top level.

This is the default behavior.

DOCUMENT

Restricts the xml instance to be a well-formed XML document. The XML data must have one and only one root element. Text nodes are not allowed at the top level.

xml_schema_collection

Is the name of an XML schema collection. To create a typed xml column or variable, you can optionally specify the XML schema collection name. For more information about typed and untyped XML, see Typed XML Compared to Untyped XML.

Remarks

The stored representation of xml data type instances cannot exceed 2 gigabytes (GB) in size. For more information, see Implementing XML in SQL Server.

The CONTENT and DOCUMENT facets apply only to typed XML. For more information see Typed XML Compared to Untyped XML.

Examples

```
USE AdventureWorks2008R2;
GO
DECLARE @y xml (Sales.IndividualSurveySchemaCollection)
SET @y =  (SELECT TOP 1 Demographics FROM Sales.Individual);
SELECT @y;
GO
```

7. Expressions

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.

Syntax

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

Arguments

Term	Definition
constant	Is a symbol that represents a single, specific data value. For more information, see Constants.
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.
(*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* }	Is an operator that has only one numeric operand: • + indicates a positive number. • - indicates a negative number. • ~ indicates the one's complement operator. Unary operators can be applied only to expressions that evaluate to any one of the data types of the numeric data type category.
{ *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.
ranking_windowed_function	Is any Transact-SQL ranking function. For more information, see Ranking Functions.
aggregate_windowed_function	Is any Transact-SQL aggregate function with the OVER clause. For more information, see OVER Clause.

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.

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.

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 AdventureWorks2008R2;
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.

7.1. CASE

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.

Syntax

```
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
```

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.

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.
- Evaluates *input_expression*, and then 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.

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 AdventureWorks2008R2;
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 AdventureWorks2008R2;
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 to replace the IIf function that is used in Microsoft Access

CASE provides functionality that is similar to the IIf function in Microsoft Access. The following example shows a simple query that uses IIf to provide an output value for the TelephoneInstructions column in an Access table that is named db1.ContactInfo.

```
SELECT FirstName, LastName, TelephoneNumber,
     IIf(IsNull(TelephoneInstructions),"Any time",
     TelephoneInstructions) AS [When to Contact]
FROM db1.ContactInfo;
```

The following example uses CASE to provide an output value for the TelephoneSpecialInstructions column in the AdventureWorks2008R2 view Person.vAdditionalContactInfo.

```
USE AdventureWorks2008R2;
GO
SELECT FirstName, LastName, TelephoneNumber, 'When to Contact' =
     CASE
          WHEN TelephoneSpecialInstructions IS NULL THEN 'Any time'
          ELSE TelephoneSpecialInstructions
     END
FROM Person.vAdditionalContactInfo;
```

D. 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 EmployeeID in descending order. Employees that have the SalariedFlag set to 0 are returned in order by the EmployeeID 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;
```

E. 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 AdventureWorks2008R2;
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;
```

F. 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 AdventureWorks2008R2 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 AdventureWorks2008R2;
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 conta
ct.
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);
```

G. 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 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 AdventureWorks2008R2;
GO
SELECT JobTitle, MAX(ph1.Rate)AS MaximumRate
FROM HumanResources.Employee AS e
JOIN HumanResources.EmployeePayHistory AS ph1 ON e.BusinessEntityID = ph1.Bus
inessEntityID
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;
```

7.2. COALESCE

Returns the first nonnull expression among its arguments.

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.

Note
At least one of the null values must be a typed NULL.

```
COALESCE(expression1,...n) is equivalent to the following CASE expression:
CASE
WHEN (expression1 IS NOT NULL) THEN expression1
WHEN (expression2 IS NOT NULL) THEN expression2
...
ELSE expressionN
END
```

ISNULL and COALESCE though equivalent, can behave differently. An expression involving ISNULL with non-null parameters is considered to be NOT NULL, while expressions involving COALESCE with non-null parameters is considered to be NULL. In SQL Server, to index expressions involving COALESCE with non-null parameters, the computed column can be persisted using the PERSISTED column attribute as in the following statement:

```
CREATE TABLE #CheckSumTest
    (
        ID int identity ,
        Num int DEFAULT ( RAND() * 100 ) ,
        RowCheckSum AS COALESCE( CHECKSUM( id , num ) , 0 ) PERSISTED PRIMARY
KEY
    );
```

Examples

A. Running a simple example

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

```
USE AdventureWorks2008R2;
GO
SELECT Name, Class, Color, ProductNumber,
COALESCE(Class, Color, ProductNumber) AS FirstNotNull
FROM Production.Product ;
GO
```

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
------------
20800.0000
41600.0000
62400.0000
```

83200.0000
10000.0000
20000.0000
30000.0000
40000.0000
45000.0000
50000.0000
120000.0000
56000.0000
(12 row(s) affected)

7.3. NULLIF

Returns a null value if the two specified expressions are equal.

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 evaluted 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.

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID ('dbo.budgets','U') IS NOT NULL
   DROP TABLE budgets;
GO
SET NOCOUNT ON;
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 AdventureWorks2008R2;
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
```

8. Language Elements

SQL Server supports the following language elements.

8.1. -- (Comment)

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.

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	SQL Server 2000
Make the selected text a comment	CTRL+K, CTRL+C	CTRL+SHIFT+C
Uncomment the selected text	CTRL+K, CTRL+U	CTRL+SHIFT+R

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

For multiline comments, see /*...*/ (Comment).

Examples

The following example uses the -- commenting characters.

```
-- Choose the AdventureWorks2008R2 database.
USE AdventureWorks2008R2;
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
```

8.2. /*...*/ (Comment)

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

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 AdventureWorks2008R2;
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 AdventureWorks2008R2 database
and their contact information.
*/
SELECT p.FirstName, p.LastName, a.AddressLine1, a.AddressLine2, a.City, a.Pos
talCode
FROM Person.Person AS p
JOIN HumanResources.Employee AS e ON p.BusinessEntityID = e.BusinessEntityID
JOIN Person.BusinessEntityAddress AS ea ON e.BusinessEntityID = ea.BusinessEn
tityID
JOIN Person.Address AS a ON ea.AddressID = a.AddressID;
GO
```

8.3. USE

Changes the database context to the specified database or database snapshot.

Syntax

```
USE { database }
```

Arguments

database

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.

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 AdventureWorks2008R2 database.

```
USE AdventureWorks2008R2;
GO
```

9. Operators

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

9.1. Arithmetic Operators

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: Adds two numbers. This addition arithmetic operator can also add a number, in days, to a date. Syntax: `expression + expression`
+= (Add Equals)	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). Syntax: `expression += expression`
- (Subtract)	Subtraction: Subtracts two numbers (an arithmetic subtraction operator). Can also subtract a number, in days, from a date. Syntax: `expression - expression`
-= (Subtract EQUALS)	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`

* (Multiply)	Multiplication: Multiplies two expressions (an arithmetic multiplication operator). Syntax: `expression * expression`
*= (Multiply EQUALS)	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). Syntax: `expression *= expression`
/ (Divide)	Division: Divides one number by another (an arithmetic division operator). Syntax: `dividend / divisor`
/= (Divide EQUALS)	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). 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).
% (Modulo)	Returns the integer remainder of a division. For example, 12 % 5 = 2 because the remainder of 12 divided by 5 is 2. Syntax: `dividend % divisor`
%= (Modulo EQUALS)	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). Syntax: `expression %= expression`

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.

9.2. Assignment Operator

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.

```
USE AdventureWorks2008R2;
GO
SELECT FirstColumnHeading = 'xyz',
       SecondColumnHeading = ProductID
FROM Production.Product;
GO
```

9.3. Bitwise Operators

Bitwise operators perform bit manipulations between two expressions of any of the data types of the integer data type category.

Operator	Meaning
~ (Bitwise NOT)	Performs a bitwise logical NOT operation on an integer value. Syntax: `~ expression`
& (Bitwise AND)	Bitwise AND (two operands). Performs a bitwise logical AND operation between two integer values. Syntax: `expression & expression`
&= (Bitwise AND EQUALS)	Performs a bitwise logical AND operation between two integer values, and sets a value to the result of the operation. Syntax: `expression &= expression`
^ (Bitwise Exclusive OR)	Bitwise exclusive OR (two operands). Performs a bitwise exclusive OR operation between two integer values. Syntax: `expression ^ expression`
^= (Bitwise Exclusive OR EQUALS)	Performs a bitwise exclusive OR operation between two integer values, and sets a value to the result of the operation. Syntax: `expression ^= expression`
| (Bitwise OR)	Bitwise OR (two operands). Performs a bitwise logical OR operation between two specified integer values as translated to binary expressions within Transact-SQL

	statements. Syntax: `expression | expression`
|= (Bitwise OR EQUALS)	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. Syntax: `expression |= expression`

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
smallint	int, smallint, tinyint, binary, or varbinary
tinyint	int, smallint, tinyint, binary, or varbinary
varbinary	int, smallint, or tinyint

9.4. Comparison Operators

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 an equals (=) operator returns TRUE if both expressions are NULL. For example, NULL = NULL returns TRUE when SET ANSI_NULLS is OFF.

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:

```
USE AdventureWorks2008R2;
GO
DECLARE @MyProduct int;
SET @MyProduct = 750;
IF (@MyProduct <> 0)
   SELECT ProductID, Name, ProductNumber
   FROM Production.Product
   WHERE ProductID = @MyProduct;
GO
```

9.5. Compound Operators

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 EQUALS)	Adds some amount to the original value and sets the original value to the result.
-=	-= (Subtract EQUALS)	Subtracts some amount from the original value and sets the original value to the result.
*=	*= (Multiply EQUALS)	Multiplies by an amount and sets the original value to the result.
/=	/= (Divide EQUALS)	Divides by an amount and sets the original value to the result.
%=	%= (Modulo EQUALS) d	Divides by an amount and sets the original value to the modulo.
&=	&= (Bitwise AND EQUALS)	Performs a bitwise AND and sets the original value to the result.
^=	^= (Bitwise Exclusive OR EQUALS)	Performs a bitwise exclusive OR and sets the original value to the result.
|=	|= (Bitwise OR EQUALS)	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.

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;
```

9.6. Logical Operators

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.

9.7. Set Operators

SQL Server provides the following set operators. Set operators combine results from two or more queries into a single result set.

9.7.1. EXCEPT and INTERSECT

Returns distinct values by comparing the results of two queries.

EXCEPT returns any distinct values from the left query that are not also found on the right query.

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

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.

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.

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 operand 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 operand.

Remarks

When the data types of comparable columns that are returned by the queries to the left and right of the EXCEPT or INTERSECT operands 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 you compare rows for determining distinct values, 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 operand.

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 operand.

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 operand
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, or together with COMPUTE and COMPUTE BY clauses.

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 operands. The first query returns all values from the Production.Product table for comparison to the results with INTERSECT and EXCEPT.

```
USE AdventureWorks2008R2;
GO
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 operand.

```
USE AdventureWorks2008R2;
GO
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 operand that are not also found on the right query.

```
USE AdventureWorks2008R2;
GO
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 operand that are not also found on the right query. The tables are reversed from the previous example.

```
USE AdventureWorks2008R2;
GO
SELECT ProductID
FROM Production.WorkOrder
```

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

9.7.2. UNION

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.

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.

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.

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID ('dbo.Gloves', 'U') IS NOT NULL
DROP TABLE dbo.Gloves;
GO
-- Create Gloves table.
SELECT ProductModelID, Name
INTO dbo.Gloves
FROM Production.ProductModel
WHERE ProductModelID IN (3, 4);
GO

-- Here is the simple union.
USE AdventureWorks2008R2;
GO
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.

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID ('dbo.ProductResults', 'U') IS NOT NULL
DROP TABLE dbo.ProductResults;
GO
IF OBJECT_ID ('dbo.Gloves', 'U') IS NOT NULL
DROP TABLE dbo.Gloves;
GO
-- Create Gloves table.
SELECT ProductModelID, Name
INTO dbo.Gloves
FROM Production.ProductModel
WHERE ProductModelID IN (3, 4);
GO

USE AdventureWorks2008R2;
```

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

SELECT *
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.

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID ('dbo.Gloves', 'U') IS NOT NULL
DROP TABLE dbo.Gloves;
GO
-- Create Gloves table.
SELECT ProductModelID, Name
INTO dbo.Gloves
FROM Production.ProductModel
WHERE ProductModelID IN (3, 4);
GO

/* INCORRECT */
USE AdventureWorks2008R2;
GO
SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE ProductModelID NOT IN (3, 4)
ORDER BY Name
UNION
SELECT ProductModelID, Name
FROM dbo.Gloves;
GO

/* CORRECT */
USE AdventureWorks2008R2;
GO
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.

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID ('dbo.EmployeeOne', 'U') IS NOT NULL
DROP TABLE dbo.EmployeeOne;
GO
IF OBJECT_ID ('dbo.EmployeeTwo', 'U') IS NOT NULL
DROP TABLE dbo.EmployeeTwo;
GO
IF OBJECT_ID ('dbo.EmployeeThree', 'U') IS NOT NULL
DROP TABLE dbo.EmployeeThree;
GO

SELECT pp.LastName, pp.FirstName, e.JobTitle
INTO dbo.EmployeeOne
FROM Person.Person AS pp JOIN HumanResources.Employee AS e
ON e.BusinessEntityID = pp.BusinessEntityID
WHERE LastName = 'Johnson';
GO
SELECT pp.LastName, pp.FirstName, e.JobTitle
INTO dbo.EmployeeTwo
FROM Person.Person AS pp JOIN HumanResources.Employee AS e
ON e.BusinessEntityID = pp.BusinessEntityID
WHERE LastName = 'Johnson';
GO
SELECT pp.LastName, pp.FirstName, e.JobTitle
INTO dbo.EmployeeThree
FROM Person.Person AS pp JOIN HumanResources.Employee AS e
ON e.BusinessEntityID = pp.BusinessEntityID
WHERE LastName = 'Johnson';
GO
```

```
-- Union ALL
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeOne
UNION ALL
SELECT LastName, FirstName ,JobTitle
FROM dbo.EmployeeTwo
UNION ALL
SELECT LastName, FirstName,JobTitle
FROM dbo.EmployeeThree;
GO

SELECT LastName, FirstName,JobTitle
FROM dbo.EmployeeOne
UNION
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeTwo
UNION
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeThree;
GO

SELECT LastName, FirstName,JobTitle
FROM dbo.EmployeeOne
UNION ALL
(
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeTwo
UNION
SELECT LastName, FirstName, JobTitle
FROM dbo.EmployeeThree
);
GO
```

9.8. String Concatenation Operator

The plus sign (+) is the string concatenation operator that enables string concatenation. All other string manipulation is handled by using string functions such as SUBSTRING.

By default, an empty string is interpreted as an empty string in INSERT or assignment statements on data of the **varchar** data type. In concatenating data of the **varchar**, **char**, or **text** data types, the empty string is interpreted as an empty string. For example, 'abc' + '' + 'def' is stored as 'abcdef'. However, if the compatibility level setting is 65, empty constants are treated as a single blank character and 'abc' + '' + 'def' is stored as 'abc def'. For more information about setting compatibility levels and interpreting empty strings, see sp_dbcmptlevel.

When two character strings are concatenated, the collation of the result expression is set following the rules of collation precedence. For more information, see Collation Precedence.

9.8.1. + (String Concatenation)

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).

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.

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.

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.

```
USE AdventureWorks2008R2;
GO
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.

```
USE AdventureWorks2008R2;
GO
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.

```
USE AdventureWorks2008R2;
GO
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)

9.8.2. += (String Concatenation)

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'.

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'.

The += operator cannot be used without a variable. For example, the following code will cause an error:

```
SELECT 'Adventure' += 'Works'
```

Examples

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.

9.8.3. % (Wildcard - Character(s) to Match)

Matches any string of zero or more characters. This wildcard character can be used as either a prefix or a suffix. For more information, see Pattern Matching in Search Conditions.

Examples

The following example returns all the first names of people in the Person table of AdventureWorks2008R2 that start with Dan.

```
USE AdventureWorks2008R2;
GO
SELECT FirstName, LastName
FROM Person.Person
WHERE FirstName LIKE 'Dan%';
GO
```

9.8.4. [ ] (Wildcard - Character(s) to Match)

Matches any single character within the specified range or set that is specified between the brackets. These wildcard characters can be used in string comparisons that involve pattern matching, such as LIKE and PATINDEX. For more information, see Pattern Matching in Search Conditions.

Examples

The following example uses the [] operator to return all Adventure Works employees who have addresses with a four-digit postal code.

```
USE AdventureWorks2008R2;
GO
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.Busi
nessEntityID
INNER JOIN Person.Address AS a ON a.AddressID = ea.AddressID
WHERE a.PostalCode LIKE '[0-9][0-9][0-9][0-9]';
GO
```

Here is the result set:

```
EmployeeID      FirstName      LastName      PostalCode
----------      ---------      ---------     ----------
290             Lynn           Tsoflias      3000
```

9.8.5. [^] (Wildcard - Character(s) Not to Match)

Matches any single character that is not within the range or set specified between the square brackets. For more information, see Pattern Matching in Search Conditions.

Examples

The following example uses the [^] operator to find all the people in the Person table who have first names that start with Al and have a third letter that is not the letter a.

```
USE AdventureWorks2008R2;
GO
SELECT FirstName, LastName
FROM Person.Person
WHERE FirstName LIKE 'Al[^a]%'
ORDER BY FirstName;
```

9.8.6. _ (Wildcard - Match One Character)

Matches any single character in a string comparison operation that involves pattern matching, such as LIKE and PATINDEX. For more information, see Pattern Matching in Search Conditions.

Examples

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 AdventureWorks2008R2;
GO
SELECT FirstName, LastName
FROM Person.Person
WHERE FirstName LIKE '_an'
ORDER BY FirstName;
```

9.9. Unary Operators

Unary operators perform an operation on only one expression of any one of the data types of the numeric data type category. For more information about data type categories, see Transact-SQL Syntax Conventions.

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.

9.10. Operator Precedence

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	* (Multiply), / (Division), % (Modulo)
3	+ (Positive), - (Negative), + (Add), (+ Concatenate), - (Subtract), & (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
```

10. Predicates

Is an expression that evaluates to TRUE, FALSE, or UNKNOWN. Predicates are used in the search condition of WHERE clauses and HAVING clauses, the join conditions of FROM clauses, and other constructs where a Boolean value is required.

10.1. IS [NOT] NULL

Determines whether a specified expression is NULL.

Syntax

```
expression IS [ NOT ] NULL
```

Arguments

expression

Is any valid expression.

NOT

Specifies that the Boolean result be negated. The predicate reverses its return values, returning TRUE if the value is not NULL, and FALSE if the value is NULL.

Result Types

Boolean

Return Code Values

If the value of *expression* is NULL, IS NULL returns TRUE; otherwise, it returns FALSE.

If the value of *expression* is NULL, IS NOT NULL returns FALSE; otherwise, it returns TRUE.

Remarks

To determine whether an expression is NULL, use IS NULL or IS NOT NULL instead of comparison operators (such as = or !=). Comparison operators return UNKNOWN when either or both arguments are NULL.

Examples

The following example returns the name and the weight for all products for which either the weight is less than 10 pounds or the color is unknown, or NULL.

```
USE AdventureWorks2008R2;
GO
```

```
SELECT Name, Weight, Color
FROM Production.Product
WHERE Weight < 10.00 OR Color IS NULL
ORDER BY Name;
GO
```

10.2. CONTAINS

Is a predicate used in a WHERE clause to search columns containing character-based data types for **precise** or **fuzzy** (less precise) matches to single words and phrases, the proximity of words within a certain distance of one another, or weighted matches.

In SQL Server, you can use four-part names in CONTAINS or FREETEXT full-text predicates to execute queries against linked servers.

CONTAINS can search for:

- A word or phrase.
- The prefix of a word or phrase.
- A word near another word.
- A word inflectionally generated from another (for example, the word **drive** is the inflectional stem of **drives**, **drove**, **driving**, and **driven**).
- A word that is a synonym of another word using a thesaurus (for example, the word **metal** can have synonyms such as **aluminum** and **steel**).

Syntax

```
CONTAINS
   ( { column_name | ( column_list ) | * }
     , '<contains_search_condition>'
   [ , LANGUAGE language_term ]
   )

<contains_search_condition> ::=
    { <simple_term>
    | <prefix_term>
    | <generation_term>
    | <proximity_term>
    | <weighted_term>
    }
    | { ( <contains_search_condition> )
    [ { <AND> | <AND NOT> | <OR> } ]
    <contains_search_condition> [ ...n ]
    }

<simple_term> ::=
     word | "phrase"
```

```
<prefix term> ::=
     { "word *" | "phrase *" }

<generation_term> ::=
     FORMSOF ( { INFLECTIONAL | THESAURUS } , <simple_term> [ ,...n ] )

<proximity_term> ::=
     { <simple_term> | <prefix_term> }
     { { NEAR | ~ }
     { <simple_term> | <prefix_term> }
     } [ ...n ]

<weighted_term> ::=
     ISABOUT
    ( { {
  <simple_term>
  | <prefix_term>
  | <generation_term>
  | <proximity_term>
  }
   [ WEIGHT ( weight_value ) ]
   } [ ,...n ]
    )

<AND> ::=
     { AND | & }

<AND NOT> ::=
     { AND NOT | &! }

<OR> ::=
     { OR | | }
```

Arguments

column_name

Is the name of a full-text indexed column of the table specified in the FROM clause. The columns can be of type char, varchar, nchar, nvarchar, text, ntext, image, xml, varbinary, or varbinary(max).

column_list

Specifies two or more columns, separated by commas. *column_list* must be enclosed in parentheses. Unless *language_term* is specified, the language of all columns of *column_list* must be the same.

*

Specifies that the query will search all full-text indexed columns in the table specified in the FROM clause for the given search condition. The columns in the CONTAINS clause must come

from a single table that has a full-text index. Unless *language_term* is specified, the language of all columns of the table must be the same.

LANGUAGE *language_term*

Is the language to use for word breaking, stemming, thesaurus expansions and replacements, and noise-word (or stopword) removal as part of the query. This parameter is optional.

If documents of different languages are stored together as binary large objects (BLOBs) in a single column, the locale identifier (LCID) of a given document determines what language to use to index its content. When querying such a column, specifying LANGUAGE *language_term* can increase the probability of a good match.

language_term can be specified as a string, integer, or hexadecimal value corresponding to the LCID of a language. If *language_term* is specified, the language it represents will be applied to all elements of the search condition. If no value is specified, the column full-text language is used.

When specified as a string, *language_term* corresponds to the **alias** column value in he sys.syslanguages compatibility view. The string must be enclosed in single quotation marks, as in '*language_term*'. When specified as an integer, *language_term* is the actual LCID that identifies the language. When specified as a hexadecimal value, *language_term* is 0x followed by the hexadecimal value of the LCID. The hexadecimal value must not exceed eight digits, including leading zeros.

If the value is in double-byte character set (DBCS) format, SQL Server will convert it to Unicode.

If the language specified is not valid or there are no resources installed that correspond to that language, SQL Server returns an error. To use the neutral language resources, specify 0x0 as *language_term*.

<*contains_search_condition*>

Specifies the text to search for in *column_name* and the conditions for a match.

<*contains_search_condition*> is nvarchar. An implicit conversion occurs when another character data type is used as input. In the following example, the @SearchWord variable, which is defined as varchar(30), causes an implicit conversion in the CONTAINS predicate.

```
USE AdventureWorks2008R2;
GO
DECLARE @SearchWord varchar(30)
SET @SearchWord ='performance'
SELECT Description
FROM Production.ProductDescription
WHERE CONTAINS(Description, @SearchWord);
```

Because "parameter sniffing" does not work across conversion, use nvarchar for better performance. In the example, declare @SearchWord as nvarchar(30).

```
USE AdventureWorks2008R2;
GO
DECLARE @SearchWord nvarchar(30)
SET @SearchWord = N'performance'
SELECT Description
FROM Production.ProductDescription
WHERE CONTAINS(Description, @SearchWord);
```

You can also use the OPTIMIZE FOR query hint for cases in which a non optimal plan is generated.

word

Is a string of characters without spaces or punctuation.

phrase

Is one or more words with spaces between each word.

> **Note**
>
> Some languages, such as those written in some parts of Asia, can have phrases that consist of one or more words without spaces between them.

<simple_term>

Specifies a match for an exact word or a phrase. Examples of valid simple terms are "blue berry", blueberry, and "Microsoft SQL Server". Phrases should be enclosed in double quotation marks (""). Words in a phrase must appear in the same order as specified in *<contains_search_condition>* as they appear in the database column. The search for characters in the word or phrase is not case-sensitive. Noise words (or stopwords) (such as a, and, or the) in full-text indexed columns are not stored in the full-text index. If a noise word is used in a single word search, SQL Server returns an error message indicating that the query contains only noise words. SQL Server includes a standard list of noise words in the directory \Mssql\Binn\FTERef of each instance of SQL Server.

Punctuation is ignored. Therefore, CONTAINS(testing, "computer failure") matches a row with the value, "Where is my computer? Failure to find it would be expensive." For more information on word-breaker behavior, see Word Breakers and Stemmers.

<prefix_term>

Specifies a match of words or phrases beginning with the specified text. Enclose a prefix term in double quotation marks ("") and add an asterisk (*) before the ending quotation mark, so that all text starting with the simple term specified before the asterisk is matched. The clause should be specified this way: CONTAINS (column, '"text*"'). The asterisk matches zero, one, or more characters (of the root word or words in the word or phrase). If the text and asterisk are not delimited by double quotation marks, so the predicate reads CONTAINS (column, 'text*'), full-text search considers the asterisk as a character and searches for exact matches to text*. The full-text

engine will not find words with the asterisk (*) character because word breakers typically ignore such characters.

When <*prefix_term*> is a phrase, each word contained in the phrase is considered to be a separate prefix. Therefore, a query specifying a prefix term of "local wine*" matches any rows with the text of "local winery", "locally wined and dined", and so on.

<generation_term>

Specifies a match of words when the included simple terms include variants of the original word for which to search.

INFLECTIONAL

Specifies that the language-dependent stemmer is to be used on the specified simple term. Stemmer behavior is defined based on stemming rules of each specific language. The neutral language does not have an associated stemmer. The column language of the columns being queried is used to refer to the desired stemmer. If *language_term* is specified, the stemmer corresponding to that language is used.

A given <*simple_term*> within a <*generation_term*> will not match both nouns and verbs.

THESAURUS

Specifies that the thesaurus corresponding to the column full-text language, or the language specified in the query is used. The longest pattern or patterns from the <*simple_term*> are matched against the thesaurus and additional terms are generated to expand or replace the original pattern. If a match is not found for all or part of the <*simple_term*>, the non-matching portion is treated as a *simple_term*. For more information on the full-text search thesaurus, see Thesaurus Configuration.

<proximity_term>

Specifies a match of words or phrases that must be in the document that is being searched. Like the AND operator, <*proximity_term*> requires both the search terms to exist in the document being searched.

NEAR | ~

Indicates that the word or phrase on each side of the NEAR or ~ operator must occur in a document for a match to be returned. Several proximity terms can be chained, as in a NEAR b NEAR c or a ~ b ~ c. Chained proximity terms must all be in the document for a match to be returned.

When used in the CONTAINSTABLE function, the proximity of the search terms affects the ranking of each document. The nearer the matched search terms are in a document, the higher the ranking of the document. If matched search terms are >50 terms apart, the rank returned on the document is 0.

For example, CONTAINS (*column_name*, 'fox NEAR chicken') and CONTAINSTABLE (*table_name*, *column_name*, 'fox ~ chicken') would both return any documents in the specified column that

contain both "fox" and "chicken". In addition, CONTAINSTABLE returns a rank for each document based on the proximity of "fox" and "chicken". For example, if a document contains the sentence, "The fox ate the chicken," its ranking would be high.

NEAR indicates the logical distance between terms, rather than the absolute distance between them. For example, terms within different phrases or sentences within a paragraph are treated as farther apart than terms in the same phrase or sentence, regardless of their actual proximity, on the assumption that they are less related. Likewise, terms in different paragraphs are treated as being even farther apart.

<weighted_term>

Specifies that the matching rows (returned by the query) match a list of words and phrases, each optionally given a weighting value.

ISABOUT

Specifies the *<weighted_term>* keyword.

WEIGHT(*weight_value*)

Specifies a weight value, which is a number from 0.0 through 1.0. Each component in *<weighted_term>* may include a *weight_value*. *weight_value* is a way to change how various portions of a query affect the rank value assigned to each row matching the query. WEIGHT does not affect the results of CONTAINS queries, but WEIGHT impacts rank in CONTAINSTABLE queries.

Note
The decimal separator is always a period, regardless of the operating system locale.

{ AND | & } | { AND NOT | &! } | { OR | | }

Specifies a logical operation between two contains search conditions.

AND | &

Indicates that the two contains search conditions must be met for a match. The ampersand symbol (&) may be used instead of the AND keyword to represent the AND operator.

AND NOT | &!

Indicates that the second search condition must not be present for a match. The ampersand followed by the exclamation mark symbol (&!) may be used instead of the AND NOT keyword to represent the AND NOT operator.

OR | |

Indicates that either of the two contains search conditions must be met for a match. The bar symbol (|) may be used instead of the OR keyword to represent the OR operator.

When <*contains_search_condition*> contains parenthesized groups, these parenthesized groups are evaluated first. After evaluating parenthesized groups, these rules apply when using these logical operators with contains search conditions:

- NOT is applied before AND.
- NOT can only occur after AND, as in AND NOT. The OR NOT operator is not allowed. NOT cannot be specified before the first term. For example, CONTAINS (mycolumn, 'NOT "phrase_to_search_for" ') is not valid.
- AND is applied before OR.
- Boolean operators of the same type (AND, OR) are associative and can therefore be applied in any order.

n

Is a placeholder indicating that multiple CONTAINS search conditions and terms within them can be specified.

Remarks

Full-text predicates and functions work on a single table, which is implied in the FROM predicate. To search on multiple tables, use a joined table in your FROM clause to search on a result set that is the product of two or more tables.

CONTAINS is not recognized as a keyword if the compatibility level is less than 70. For more information, see sp_dbcmptlevel.

Full-text predicates are not allowed in the OUTPUT clause when the database compatibility level is set to 100.

Comparison of LIKE to Full-Text Search

In contrast to full-text search, the LIKE Transact-SQL predicate works on character patterns only. Also, you cannot use the LIKE predicate to query formatted binary data. Furthermore, a LIKE query against a large amount of unstructured text data is much slower than an equivalent full-text query against the same data. A LIKE query against millions of rows of text data can take minutes to return; whereas a full-text query can take only seconds or less against the same data, depending on the number of rows that are returned.

Examples

A. Using CONTAINS with <simple_term>

The following example finds all products with a price of $80.99 that contain the word "Mountain".

```
USE AdventureWorks2008R2;
GO
SELECT Name, ListPrice
FROM Production.Product
WHERE ListPrice = 80.99
   AND CONTAINS(Name, 'Mountain');
GO
```

B. Using CONTAINS and phrase in <simple_term>

The following example returns all products that contain either the phrase "Mountain" or "Road".

```
USE AdventureWorks2008R2;
GO
SELECT Name
FROM Production.Product
WHERE CONTAINS(Name, ' "Mountain" OR "Road" ')
GO
```

C. Using CONTAINS with <prefix_term>

The following example returns all product names with at least one word starting with the prefix chain in the Name column.

```
USE AdventureWorks2008R2;
GO
SELECT Name
FROM Production.Product
WHERE CONTAINS(Name, ' "Chain*" ');
GO
```

D. Using CONTAINS and OR with <prefix_term>

The following example returns all category descriptions containing strings with prefixes of either "chain" or "full".

```
USE AdventureWorks2008R2;
GO
SELECT Name
FROM Production.Product
WHERE CONTAINS(Name, '"chain*" OR "full*"');
GO
```

E. Using CONTAINS with <proximity_term>

The following example returns all product names that have the word bike near the word performance.

```
USE AdventureWorks2008R2;
GO
SELECT Description
FROM Production.ProductDescription
WHERE CONTAINS(Description, 'bike NEAR performance');
GO
```

F. Using CONTAINS with <generation_term>

The following example searches for all products with words of the form ride: riding, ridden, and so on.

```
USE AdventureWorks2008R2;
GO
SELECT Description
FROM Production.ProductDescription
WHERE CONTAINS(Description, ' FORMSOF (INFLECTIONAL, ride) ');
GO
```

G. Using CONTAINS with <weighted_term>

The following example searches for all product names containing the words performance, comfortable, or smooth, and different weightings are given to each word.

```
USE AdventureWorks2008R2;
GO
SELECT Description
FROM Production.ProductDescription
WHERE CONTAINS(Description, 'ISABOUT (performance weight (.8),
comfortable weight (.4), smooth weight (.2) )' );
GO
```

H. Using CONTAINS with variables

The following example uses a variable instead of a specific search term.

```
USE AdventureWorks2008R2;
GO
DECLARE @SearchWord nvarchar(30)
SET @SearchWord = N'Performance'
SELECT Description
FROM Production.ProductDescription
WHERE CONTAINS(Description, @SearchWord);
GO
```

I. Using CONTAINS with a logical operator (AND)

The following example uses the ProductDescription table of the AdventureWorks2008R2 database. The query uses the CONTAINS predicate to search for descriptions in which the description ID is not equal to 5 and the description contains both the word "Aluminum" and the word "spindle." The search condition uses the AND Boolean operator.

```
USE AdventureWorks2008R2;
GO
SELECT Description
FROM Production.ProductDescription
WHERE ProductDescriptionID <> 5 AND
  CONTAINS(Description, ' Aluminum AND spindle');
GO
```

J. Using CONTAINS to verify a row insertion

The following example uses CONTAINS within a SELECT subquery. Using the AdventureWorks2008R2 database, the query obtains the comment value of all the comments in the ProductReview table for a particular cycle. The search condition uses the AND Boolean operator.

```
USE AdventureWorks2008R2;
GO
INSERT INTO Production.ProductReview
(ProductID, ReviewerName, EmailAddress, Rating, Comments)
VALUES
(780, 'John Smith', 'john@fourthcoffee.com', 5,
'The Mountain-200 Silver from Adventure Works Cycles meets and exceeds expect
ations. I enjoyed the smooth ride down the roads of Redmond')

-- Given the full-text catalog for these tables is Adv_ft_ctlg,
-- with change_tracking on so that the full-text indexes are updated automati
cally.
WAITFOR DELAY '00:00:30'
-- Wait 30 seconds to make sure that the full-text index gets updated.

SELECT r.Comments, p.Name
FROM Production.ProductReview r
JOIN Production.Product p
ON
 r.ProductID = p.ProductID

AND r.ProductID = (SELECT ProductID FROM Production.ProductReview
                   WHERE CONTAINS (Comments, ' Adventure Works AND
                                     Redmond AND "Mountain-200 Silver" '))
GO
```

10.3. FREETEXT

Is a predicate used in a WHERE clause to search columns containing character-based data types for values that match the **meaning** and not just the exact wording of the words in the search condition. When FREETEXT is used, the full-text query engine internally performs the following actions on the *freetext_string*, assigns each term a weight, and then finds the matches.

- Separates the string into individual words based on word boundaries (word-breaking).
- Generates inflectional forms of the words (stemming).
- Identifies a list of expansions or replacements for the terms based on matches in the thesaurus.

Syntax

```
FREETEXT ( { column_name | (column_list) | * }
          , 'freetext_string' [ , LANGUAGE language_term ] )
```

Arguments

column_name

Is the name of one or more full-text indexed columns of the table specified in the FROM clause. The columns can be of type char, varchar, nchar, nvarchar, text, ntext, image, xml, varbinary, or varbinary(max).

column_list

Indicates that several columns, separated by a comma, can be specified. *column_list* must be enclosed in parentheses. Unless *language_term* is specified, the language of all columns of *column_list* must be the same.

*

Specifies that all columns that have been registered for full-text searching should be used to search for the given *freetext_string*. If more than one table is in the FROM clause, * must be qualified by the table name. Unless *language_term* is specified, the language of all columns of the table must be the same.

freetext_string

Is text to search for in the *column_name*. Any text, including words, phrases or sentences, can be entered. Matches are generated if any term or the forms of any term is found in the full-text index.

Unlike in the CONTAINS and CONTAINSTABLE search condition where AND is a keyword, when used in *freetext_string* the word 'and' is considered a noise word, or stopword, and will be discarded.

Use of WEIGHT, FORMSOF, wildcards, NEAR and other syntax is not allowed. *freetext_string* is wordbroken, stemmed, and passed through the thesaurus. If *freetext_string* is enclosed in double quotation marks, a phrase match is instead performed; stemming and thesaurus are not performed.

freetext_string is nvarchar. An implicit conversion occurs when another character data type is used as input. In the following example, the @SearchWord variable, which is defined as varchar(30), causes an implicit conversion in the FREETEXT predicate.

```
USE AdventureWorks2008R2;
GO
DECLARE @SearchWord varchar(30)
SET @SearchWord ='performance'
SELECT Description
FROM Production.ProductDescription
WHERE FREETEXT(Description, @SearchWord);
```

Because "parameter sniffing" does not work across conversion, use nvarchar for better performance. In the example, declare @SearchWord as nvarchar(30).

```
USE AdventureWorks2008R2;
```

```
GO
DECLARE @SearchWord nvarchar(30)
SET @SearchWord = N'performance'
SELECT Description
FROM Production.ProductDescription
WHERE FREETEXT(Description, @SearchWord);
```

You can also use the OPTIMIZE FOR query hint for cases in which a non-optimal plan is generated.

LANGUAGE *language_term*

Is the language whose resources will be used for word breaking, stemming, and thesaurus and stopword removal as part of the query. This parameter is optional and can be specified as a string, integer, or hexadecimal value corresponding to the locale identifier (LCID) of a language. If *language_term* is specified, the language it represents will be applied to all elements of the search condition. If no value is specified, the column full-text language is used.

If documents of different languages are stored together as binary large objects (BLOBs) in a single column, the locale identifier (LCID) of a given document determines what language is used to index its content. When querying such a column, specifying *LANGUAGE language_term* can increase the probability of a good match.

When specified as a string, *language_term* corresponds to the **alias** column value in he sys.syslanguages compatibility view. The string must be enclosed in single quotation marks, as in '*language_term*'. When specified as an integer, *language_term* is the actual LCID that identifies the language. When specified as a hexadecimal value, *language_term* is 0x followed by the hexadecimal value of the LCID. The hexadecimal value must not exceed eight digits, including leading zeros.

If the value is in double-byte character set (DBCS) format, Microsoft SQL Server will convert it to Unicode.

If the language specified is not valid or there are no resources installed that correspond to that language, Microsoft SQL Server returns an error. To use the neutral language resources, specify 0x0 as *language_term*.

Remarks

Full-text predicates and functions work on a single table, which is implied in the FROM predicate. To search on multiple tables, use a joined table in your FROM clause to search on a result set that is the product of two or more tables.

Full-text queries using FREETEXT are less precise than those full-text queries using CONTAINS. The SQL Server full-text search engine identifies important words and phrases. No special meaning is given to any of the reserved keywords or wildcard characters that typically have meaning when specified in the <contains_search_condition> parameter of the CONTAINS predicate.

FREETEXT is not recognized as a keyword if the compatibility level is less than 70. For more information, see sp_dbcmptlevel.

Full-text predicates are not allowed in the OUTPUT clause when the database compatibility level is set to 100.

Comparison of LIKE with Full-Text Search

In contrast to full-text search, the LIKE Transact-SQL predicate works on character patterns only. Also, you cannot use the LIKE predicate to query formatted binary data. Furthermore, a LIKE query against a large amount of unstructured text data is much slower than an equivalent full-text query against the same data. A LIKE query against millions of rows of text data can take minutes to return; whereas a full-text query can take only seconds or less against the same data, depending on the number of rows that are returned.

Examples

A. Using FREETEXT to search for words containing specified character values

The following example searches for all documents containing the words related to "vital", "safety", and "components".

```
USE AdventureWorks2008R2;
GO
SELECT Title
FROM Production.Document
WHERE FREETEXT (Document, 'vital safety components' );
GO
```

B. Using FREETEXT with variables

The following example uses a variable instead of a specific search term.

```
USE AdventureWorks2008R2;
GO
DECLARE @SearchWord nvarchar(30);
SET @SearchWord = N'high-performance';
SELECT Description
FROM Production.ProductDescription
WHERE FREETEXT(Description, @SearchWord);
GO
```

11. PRINT

Returns a user-defined message to the client.

Syntax

```
PRINT msg_str | @local_variable | string_expr
```

Arguments

msg_str

Is a character string or Unicode string constant. For more information, see Constants.

@*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.

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).

For information about how applications process the messages returned by the PRINT statement, see Handling Errors and Messages in Applications.

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
```

12. RAISERROR

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.

Syntax

```
RAISERROR ( { msg_id | 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
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.

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.

Note

Severity levels less than 0 are interpreted as 0. Severity levels greater than 25 are interpreted

as 25.

state

Is an integer from 0 through 255. Negative values or values larger than 255 generate an error.

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.
NOWAIT	Sends messages immediately to the client.
SETERROR	Sets the @@ERROR and ERROR_NUMBER values to *msg_id* or 50000, regardless of the severity level.

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. For more information, see Using TRY...CATCH in Transact-SQL.

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
```

13. SET Statements

The Transact-SQL programming language provides several SET statements that change the current session handling of specific information. The SET statements are grouped into the categories shown in the following table.

For information about setting local variables with the SET statement, see SET @local_variable.

Category	Statements
Date and time statements	SET DATEFIRST SET DATEFORMAT
Locking statements	SET DEADLOCK_PRIORITY SET LOCK_TIMEOUT
Miscellaneous statements	SET CONCAT_NULL_YIELDS_NULL SET CURSOR_CLOSE_ON_COMMIT SET FIPS_FLAGGER SET IDENTITY_INSERT SET LANGUAGE SET OFFSETS SET QUOTED_IDENTIFIER
Query Execution Statements	SET ARITHABORT SET ARITHIGNORE SET FMTONLY SET NOCOUNT SET NOEXEC SET NUMERIC_ROUNDABORT SET PARSEONLY SET QUERY_GOVERNOR_COST_LIMIT SET ROWCOUNT SET TEXTSIZE
ISO Settings statements	SET ANSI_DEFAULTS SET ANSI_NULL_DFLT_OFF SET ANSI_NULL_DFLT_ON SET ANSI_NULLS SET ANSI_PADDING SET ANSI_WARNINGS
Statistics statements	SET FORCEPLAN SET SHOWPLAN_ALL SET SHOWPLAN_TEXT SET SHOWPLAN_XML SET STATISTICS IO SET STATISTICS XML

	SET STATISTICS PROFILE SET STATISTICS TIME
Transactions statements	SET IMPLICIT_TRANSACTIONS SET REMOTE_PROC_TRANSACTIONS SET TRANSACTION ISOLATION LEVEL SET XACT_ABORT

Considerations When You Use the SET Statements

- All SET statements are implemented at execute or run time, except for SET FIPS_FLAGGER, SET OFFSETS, SET PARSEONLY, and SET QUOTED_IDENTIFIER. These statements are implemented at parse time.
- If a SET statement is run in a stored procedure or trigger, the value of the SET option is restored after control is returned from the stored procedure or trigger. Also, if a SET statement is specified in a dynamic SQL string that is run by using either **sp_executesql** or EXECUTE, the value of the SET option is restored after control is returned from the batch specified in the dynamic SQL string.
- Stored procedures execute with the SET settings specified at execute time except for SET ANSI_NULLS and SET QUOTED_IDENTIFIER. Stored procedures specifying SET ANSI_NULLS or SET QUOTED_IDENTIFIER use the setting specified at stored procedure creation time. If used inside a stored procedure, any SET setting is ignored.
- The **user options** setting of **sp_configure** allows for server-wide settings and works across multiple databases. This setting also behaves like an explicit SET statement, except that it occurs at login time.
- Database settings set by using ALTER DATABASE are valid only at the database level and take effect only if explicitly set. Database settings override instance option settings that are set by using **sp_configure**.
- For any one of the SET statements with ON and OFF settings, you can specify either an ON or OFF setting for multiple SET options.

> **Note**
> This does not apply to the statistics related SET options.

- For example, SET QUOTED_IDENTIFIER, ANSI_NULLS ON sets both QUOTED_IDENTIFIER and ANSI_NULLS to ON.
- SET statement settings override equivalent database option settings that are set by using ALTER DATABASE. For example, the value specified in a SET ANSI_NULLS statement will override the database setting for ANSI_NULLs. Additionally, some connection settings are automatically set ON when a user connects to a database based on the values put into effect by the previous use of the **sp_configure user options** setting, or the values that apply to all ODBC and OLE/DB connections.
- ALTER, CREATE and DROP DATABASE statements do not honor the SET LOCK_TIMEOUT setting.
- When a global or shortcut SET statement, such as SET ANSI_DEFAULTS, sets several settings, issuing the shortcut SET statement resets the previous settings for all those options affected by the shortcut SET statement. If an individual SET option that is affected by a shortcut SET statement is explicitly set after the shortcut SET statement is issued, the individual SET statement overrides the corresponding shortcut settings.
- When batches are used, the database context is determined by the batch established by using the USE statement. Ad hoc queries and all other statements that are executed outside the stored

procedure and that are in batches inherit the option settings of the database and connection established by the USE statement.

- Multiple Active Result Set (MARS) requests share a global state that contains the most recent session SET option settings. When each request executes it can modify the SET options. The changes are specific to the request context in which they are set, and do not affect other concurrent MARS requests. However, after the request execution is completed, the new SET options are copied to the global session state. New requests that execute under the same session after this change will use these new SET option settings.
- When a stored procedure is executed, either from a batch or from another stored procedure, it is executed under the option values that are currently set in the database that contains the stored procedure. For example, when stored procedure **db1.dbo.sp1** calls stored procedure **db2.dbo.sp2**, stored procedure **sp1** is executed under the current compatibility level setting of database **db1**, and stored procedure **sp2** is executed under the current compatibility level setting of database **db2**.
- When a Transact-SQL statement refers to objects that reside in multiple databases, the current database context and the current connection context applies to that statement. In this case, if Transact-SQL statement is in a batch, the current connection context is the database defined by the USE statement; if the Transact-SQL statement is in a stored procedure, the connection context is the database that contains the stored procedure.
- When you are creating and manipulating indexes on computed columns or indexed views, the SET options ARITHABORT, CONCAT_NULL_YIELDS_NULL, QUOTED_IDENTIFIER, ANSI_NULLS, ANSI_PADDING, and ANSI_WARNINGS must be set to ON. The option NUMERIC_ROUNDABORT must be set to OFF.

  If any one of these options is not set to the required values, INSERT, UPDATE, DELETE, DBCC CHECKDB and DBCC CHECKTABLE actions on indexed views or tables with indexes on computed columns will fail. SQL Server will raise an error listing all the options that are incorrectly set. Also, SQL Server will process SELECT statements on these tables or indexed views as if the indexes on computed columns or on the views do not exist.

14. Variables

SQL Server provides the following statements to declare and set local variables.

14.1. DECLARE @local_variable

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.

Syntax

```
DECLARE
     {
{{ @local_variable [AS] data_type } | [ = value ] }
    | { @cursor_variable_name CURSOR }
} [,...n]
    | { @table_variable_name [AS] <table_type_definition> | <user-defined tab
le type> }

<table_type_definition> ::=
     TABLE ( { <column_definition> | <table_constraint> } [ ,... ]
   )

<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 [ ,... ] )
     | CHECK ( search_condition )
     }

<index_option> ::=
See CREATE TABLE for index option syntax.
```

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. For more information about CLR user-defined types or alias data types, see CREATE TYPE.

=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.

@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_definition> 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.

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.

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

Are keywords that determine whether null values are allowed in the column.

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.

<user-defined table type>

Specifies that the variable is a user-defined table type.

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 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.
- 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. For more information about cursor variables, see Transact-SQL Cursors.

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 AdventureWorks2008R2;
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 p
JOIN Person.PersonPhone ph
ON p.BusinessEntityID = ph.BusinessEntityID
WHERE LastName LIKE 'Man%';
```

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 AdventureWorks2008R2;
```

```
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.

```
USE AdventureWorks2008R2;
GO
DECLARE @MyTableVar table(
    EmpID int NOT NULL,
    OldVacationHours int,
    NewVacationHours int,
    ModifiedDate datetime);
UPDATE TOP (10) HumanResources.Employee
SET VacationHours = VacationHours * 1.25,
    ModifiedDate = GETDATE()
OUTPUT inserted.BusinessEntityID,
       deleted.VacationHours,
       inserted.VacationHours,
       inserted.ModifiedDate
INTO @MyTableVar;
--Display the result set of the table variable.
SELECT EmpID, OldVacationHours, NewVacationHours, ModifiedDate
FROM @MyTableVar;
GO
--Display the result set of the table.
SELECT TOP (10) BusinessEntityID, VacationHours, ModifiedDate
FROM HumanResources.Employee;
GO
```

D. 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. For more information about table-valued parameters, see Table-Valued Parameters (Database Engine).

```
DECLARE @LocationTVP
AS LocationTableType;
```

14.2. SET @local_variable

Sets the specified local variable, previously created by using the DECLARE @*local_variable* statement, to the specified value.

Syntax

```
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 ] ] } ]
      }
    }
}
```

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.

Note

In SQL Server 2000, FAST_FORWARD and FORWARD_ONLY cursor options are mutually exclusive. If one is specified, the other cannot be, and an error is raised. Both keywords can be used in the same DECLARE CURSOR statement.

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.

Note
In SQL Server 2000, FAST_FORWARD and FORWARD_ONLY cursor options are mutually exclusive. If one is specified, the other cannot be, and an error is raised. Both keywords can be used in the same DECLARE CURSOR statement.

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 COMPUTE, COMPUTE BY, 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.

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.

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 AdventureWorks2008R2;
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 AdventureWorks2008R2.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 AdventureWorks2008R2;
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);
```

14.3. SELECT @local_variable

Specifies that the specified local variable that is created by using DECLARE @*local_variable* should be set to the specified expression.

For assigning variables, we recommend that you use SET @*local_variable* instead of SELECT @*local_variable*. For more information, see SET @local_variable.

Syntax

```
SELECT { @local_variable { = | += | -= | *= | /= | %= | &= | ^= | |= } expres
sion } [ ,...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:

+= 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. Using 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.

```
USE AdventureWorks2008R2 ;
GO
DECLARE @var1 nvarchar(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. Using SELECT @local_variable to no result set returns 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.

```
USE AdventureWorks2008R2 ;
GO
DECLARE @var1 nvarchar(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
```

15. CREATE FUNCTION

Creates a user-defined function in SQL Server 2008 R2. A user-defined function is a Transact-SQL or common language runtime (CLR) routine that accepts parameters, performs an action, such as a complex calculation, and returns the result of that action as a value. The return value can either be a scalar (single) value or a table. Use this statement to create a reusable routine that can be used in these ways:

- In Transact-SQL statements such as SELECT
- In applications calling the function
- In the definition of another user-defined function
- To parameterize a view or improve the functionality of an indexed view
- To define a column in a table
- To define a CHECK constraint on a column
- To replace a stored procedure

Syntax

```
--Transact-SQL Scalar Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN scalar_expression
    END
[ ; ]

--Transact-SQL Inline Table-Valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS TABLE
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    RETURN [ ( ] select_stmt [ ) ]
[ ; ]

--Transact-SQL Multistatement Table-valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [READONLY] }
```

```
    [ ,...n ]
  ]
)
RETURNS @return_variable TABLE <table_type_definition>
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN
    END
[ ; ]

--Transact-SQL Function Clauses
<function_option>::=
{
    [ ENCRYPTION ]
  | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}

<table_type_definition>:: =
( { <column_definition> <column_constraint>
  | <computed_column_definition> }
    [ <table_constraint> ] [ ,...n ]
)
<column_definition>::=
{
    { column_name data_type }
    [ [ DEFAULT constant_expression ]
      [ COLLATE collation_name ] | [ ROWGUIDCOL ]
    ]
    | [ IDENTITY [ (seed , increment ) ] ]
    [ <column_constraint> [ ...n ] ]
}

<column_constraint>::=
{
    [ NULL | NOT NULL ]
    { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      [ WITH FILLFACTOR = fillfactor
        | WITH ( < index_option > [ , ...n ] )
      [ ON { filegroup | "default" } ]
  | [ CHECK ( logical_expression ) ] [ ,...n ]
}

<computed_column_definition>::=
column_name AS computed_column_expression
```

```
<table_constraint>::=
{
    { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      ( column_name [ ASC | DESC ] [ ,...n ] )
        [ WITH FILLFACTOR = fillfactor
        | WITH ( <index_option> [ , ...n ] )
  | [ CHECK ( logical_expression ) ] [ ,...n ]
}

<index_option>::=
{
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS ={ ON | OFF }
}

--CLR Scalar Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS { return_data_type }
    [ WITH <clr_function_option> [ ,...n ] ]
    [ AS ] EXTERNAL NAME <method_specifier>
[ ; ]

--CLR Table-Valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS TABLE <clr_table_type_definition>
    [ WITH <clr_function_option> [ ,...n ] ]
    [ ORDER ( <order_clause> ) ]
    [ AS ] EXTERNAL NAME <method_specifier>
[ ; ]

--CLR Function Clauses
<order_clause> ::=
{
   <column_name_in_clr_table_type_definition>
   [ ASC | DESC ]
```

```
} [ ,...n]

<method_specifier>::=
    assembly_name.class_name.method_name

<clr_function_option>::=
}
    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}

<clr_table_type_definition>::=
( { column_name data_type } [ ,...n ] )
```

Arguments

schema_name

Is the name of the schema to which the user-defined function belongs.

function_name

Is the name of the user-defined function. Function names must comply with the rules for identifiers and must be unique within the database and to its schema.

> **Note**
> Parentheses are required after the function name even if a parameter is not specified.

@parameter_name

Is a parameter in the user-defined function. One or more parameters can be declared.

A function can have a maximum of 2,100 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the parameter is defined.

Specify a parameter name by using an at sign (@) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the function; the same parameter names can be used in other functions. Parameters can take the place only of constants; they cannot be used instead of table names, column names, or the names of other database objects.

> **Note**
> ANSI_WARNINGS is not honored when you pass parameters in a stored procedure, user-defined function, or when you declare and set variables in a batch statement. For example, if a variable is defined as char(3), and then set to a value larger than three characters, the data is truncated to the defined size and the INSERT or UPDATE statement succeeds.

[*type_schema_name.*] *parameter_data_type*

Is the parameter data type, and optionally the schema to which it belongs. For Transact-SQL functions, all data types, including CLR user-defined types and user-defined table types, are allowed except the timestamp data type. For CLR functions, all data types, including CLR user-defined types, are allowed except text, ntext, image, user-defined table types and timestamp data types. The nonscalar types, cursor and table, cannot be specified as a parameter data type in either Transact-SQL or CLR functions.

If *type_schema_name* is not specified, the Database Engine looks for the *scalar_parameter_data_type* in the following order:

- The schema that contains the names of SQL Server system data types.
- The default schema of the current user in the current database.
- The dbo schema in the current database.

[=*default*]

Is a default value for the parameter. If a *default* value is defined, the function can be executed without specifying a value for that parameter.

Note
Default parameter values can be specified for CLR functions except for the varchar(max) and varbinary(max) data types.

When a parameter of the function has a default value, the keyword DEFAULT must be specified when the function is called in order to retrieve the default value. This behavior is different from using parameters with default values in stored procedures in which omitting the parameter also implies the default value. An exception to this behavior is when invoking a scalar function by using the EXECUTE statement. When using EXECUTE, the DEFAULT keyword is not required.

READONLY

Indicates that the parameter cannot be updated or modified within the definition of the function. If the parameter type is a user-defined table type, READONLY should be specified.

return_data_type

Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the timestamp data type. For CLR functions, all data types, including CLR user-defined types, are allowed except the text, ntext, image, and timestamp data types. The nonscalar types, cursor and table, cannot be specified as a return data type in either Transact-SQL or CLR functions.

function_body

Specifies that a series of Transact-SQL statements, which together do not produce a side effect such as modifying a table, define the value of the function. *function_body* is used only in scalar functions and multistatement table-valued functions.

In scalar functions, *function_body* is a series of Transact-SQL statements that together evaluate to a scalar value.

In multistatement table-valued functions, *function_body* is a series of Transact-SQL statements that populate a TABLE return variable.

scalar_expression

Specifies the scalar value that the scalar function returns.

TABLE

Specifies that the return value of the table-valued function is a table. Only constants and @*local_variables* can be passed to table-valued functions.

In inline table-valued functions, the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables.

In multistatement table-valued functions, @*return_variable* is a TABLE variable, used to store and accumulate the rows that should be returned as the value of the function. @*return_variable* can be specified only for Transact-SQL functions and not for CLR functions.

select_stmt

Is the single SELECT statement that defines the return value of an inline table-valued function.

ORDER (<order_clause>)

Specifies the order in which results are being returned from the table-valued function. For more information, see the section, "Guidance on Using Sort Order," later in this topic.

EXTERNAL NAME <method_specifier> *assembly_name.class_name.method_name*

Specifies the method of an assembly to bind with the function. *assembly_name* must match an existing assembly in SQL Server in the current database with visibility on. *class_name* must be a valid SQL Server identifier and must exist as a class in the assembly. If the class has a namespace-qualified name that uses a period (.) to separate namespace parts, the class name must be delimited by using brackets ([ ]) or quotation marks (" "). *method_name* must be a valid SQL Server identifier and must exist as a static method in the specified class.

Note

By default, SQL Server cannot execute CLR code. You can create, modify, and drop database objects that reference common language runtime modules; however, you cannot execute these references in SQL Server until you enable the clr enabled option. To enable this option, use sp_configure.

<table_type_definition> ({ <column_definition> <column_constraint> | <computed_column_definition> } [<table_constraint>] [,...*n*])

Defines the table data type for a Transact-SQL function. The table declaration includes column definitions and column or table constraints. The table is always put in the primary filegroup.

< clr_table_type_definition > ({ *column_namedata_type* } [,*...n*])

Defines the table data types for a CLR function. The table declaration includes only column names and data types. The table is always put in the primary filegroup.

<function_option>::= and <clr_function_option>::=

Specifies that the function will have one or more of the following options.

ENCRYPTION

Indicates that the Database Engine will convert the original text of the CREATE FUNCTION statement to an obfuscated format. The output of the obfuscation is not directly visible in any catalog views. Users that have no access to system tables or database files cannot retrieve the obfuscated text. However, the text will be available to privileged users that can either access system tables over the DAC port or directly access database files. Also, users that can attach a debugger to the server process can retrieve the original procedure from memory at runtime. For more information about accessing system metadata, see Metadata Visibility Configuration.

Using this option prevents the function from being published as part of SQL Server replication. This option cannot be specified for CLR functions.

SCHEMABINDING

Specifies that the function is bound to the database objects that it references. When SCHEMABINDING is specified, the base objects cannot be modified in a way that would affect the function definition. The function definition itself must first be modified or dropped to remove dependencies on the object that is to be modified.

The binding of the function to the objects it references is removed only when one of the following actions occurs:

- The function is dropped.
- The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.

A function can be schema bound only if the following conditions are true:

- The function is a Transact-SQL function.
- The user-defined functions and views referenced by the function are also schema-bound.
- The objects referenced by the function are referenced using a two-part name.
- The function and the objects it references belong to the same database.
- The user who executed the CREATE FUNCTION statement has REFERENCES permission on the database objects that the function references.

RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

Specifies the OnNULLCall attribute of a scalar-valued function. If not specified, CALLED ON NULL INPUT is implied by default. This means that the function body executes even if NULL is passed as an argument.

If RETURNS NULL ON NULL INPUT is specified in a CLR function, it indicates that SQL Server can return NULL when any of the arguments it receives is NULL, without actually invoking the body of the function. If the method of a CLR function specified in <method_specifier> already has a custom attribute that indicates RETURNS NULL ON NULL INPUT, but the CREATE FUNCTION statement indicates CALLED ON NULL INPUT, the CREATE FUNCTION statement takes precedence. The OnNULLCall attribute cannot be specified for CLR table-valued functions.

EXECUTE AS Clause

Specifies the security context under which the user-defined function is executed. Therefore, you can control which user account SQL Server uses to validate permissions on any database objects that are referenced by the function.

Note
EXECUTE AS cannot be specified for inline user-defined functions.

For more information, see EXECUTE AS Clause.

< column_definition >::=

Defines the table data type. The table declaration includes column definitions and constraints. For CLR functions, only *column_name* and *data_type* can be specified.

column_name

Is the name of a column in the table. Column names must comply with the rules for identifiers and must be unique in the table. *column_name* can consist of 1 through 128 characters.

data_type

Specifies the column data type. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except timestamp. For CLR functions, all data types, including CLR user-defined types, are allowed except text, ntext, image, char, varchar, varchar(max), and timestamp.The nonscalar type cursor cannot be specified as a column data type in either Transact-SQL or CLR functions.

DEFAULT *constant_expression*

Specifies the value provided for the column when a value is not explicitly supplied during an insert. *constant_expression* is a constant, NULL, or a system function value. DEFAULT definitions can be applied to any column except those that have the IDENTITY property. DEFAULT cannot be specified for CLR table-valued functions.

COLLATE *collation_name*

Specifies the collation for the column. If not specified, the column is assigned the default collation of the database. Collation name can be either a Windows collation name or a SQL collation name. For a list of and more information about collations, see Windows Collation Name and SQL Server Collation Name.

The COLLATE clause can be used to change the collations only of columns of the char, varchar, nchar, and nvarchar data types.

COLLATE cannot be specified for CLR table-valued functions.

ROWGUIDCOL

Indicates that the new column is a row globally 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.

The ROWGUIDCOL property does not enforce uniqueness of the values stored in the column. It also does not automatically generate values for new rows inserted into the table. To generate unique values for each column, use the NEWID function on INSERT statements. A default value can be specified; however, NEWID cannot be specified as the default.

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 typically used together with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to tinyint, smallint, int, bigint, 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).

IDENTITY cannot be specified for CLR table-valued functions.

seed

Is the integer value to be assigned to the first row in the table.

increment

Is the integer value to add to the *seed* value for successive rows in the table.

< column_constraint >::= and < table_constraint>::=

Defines the constraint for a specified column or table. For CLR functions, the only constraint type allowed is NULL. Named constraints are not allowed.

NULL | NOT NULL

Determines whether null values are allowed in the column. NULL is not strictly a constraint but can be specified just like NOT NULL. NOT NULL cannot be specified for CLR table-valued functions.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column through a unique index. In table-valued user-defined functions, the PRIMARY KEY constraint can be created on only one column per table. PRIMARY KEY cannot be specified for CLR table-valued functions.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints. UNIQUE cannot be specified for CLR table-valued functions.

CLUSTERED | NONCLUSTERED

Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints use CLUSTERED, and UNIQUE constraints use NONCLUSTERED.

CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY uses NONCLUSTERED.

CLUSTERED and NONCLUSTERED cannot be specified for CLR table-valued functions.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints cannot be specified for CLR table-valued functions.

logical_expression

Is a logical expression that returns TRUE or FALSE.

<computed_column_definition>::=

Specifies a computed column. For more information about computed columns, see CREATE TABLE.

column_name

Is the name of the computed column.

computed_column_expression

Is an expression that defines the value of a computed column.

<index_option>::=

Specifies the index options for the PRIMARY KEY or UNIQUE index. For more information about index options, see CREATE INDEX.

PAD_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

FILLFACTOR = *fillfactor*

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or change. *fillfactor* must be an integer value from 1 to 100. The default is 0.

IGNORE_DUP_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The default is OFF.

STATISTICS_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ALLOW_ROW_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ALLOW_PAGE_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

Best Practices

If a user-defined function is not created with the SCHEMABINDING clause, changes that are made to underlying objects can affect the definition of the function and produce unexpected results when it is invoked. We recommend that you implement one of the following methods to ensure that the function does not become outdated because of changes to its underlying objects:

- Specify the WITH SCHEMABINDING clause when you are creating the function. This ensures that the objects referenced in the function definition cannot be modified unless the function is also modified.
- Execute the sp_refreshsqlmodule stored procedure after modifying any object that is specified in the definition of the function.

Data Types

If parameters are specified in a CLR function, they should be SQL Server types as defined previously for *scalar_parameter_data_type*. For information about comparing SQL Server system data types to CLR integration data types or .NET Framework common language runtime data types, see Mapping CLR Parameter Data.

For SQL Server to reference the correct method when it is overloaded in a class, the method indicated in <method_specifier> must have the following characteristics:

- Receive the same number of parameters as specified in [,...*n*].

- Receive all the parameters by value, not by reference.
- Use parameter types that are compatible with those specified in the SQL Server function.

If the return data type of the CLR function specifies a table type (RETURNS TABLE), the return data type of the method in <method_specifier> should be of type IEnumerator or IEnumerable, and it is assumed that the interface is implemented by the creator of the function. Unlike Transact-SQL functions, CLR functions cannot include PRIMARY KEY, UNIQUE, or CHECK constraints in <table_type_definition>. The data types of columns specified in <table_type_definition> must match the types of the corresponding columns of the result set returned by the method in <method_specifier> at execution time. This type-checking is not performed at the time the function is created.

For more information about how to program CLR functions, see CLR User-Defined Functions.

General Remarks

Scalar-valued functions can be invoked where scalar expressions are used. This includes computed columns and CHECK constraint definitions. Scalar-valued functions can also be executed by using the EXECUTE statement. Scalar-valued functions must be invoked by using at least the two-part name of the function. For more information about multipart names, see Transact-SQL Syntax Conventions. Table-valued functions can be invoked where table expressions are allowed in the FROM clause of SELECT, INSERT, UPDATE, or DELETE statements. For more information, see Executing User-Defined Functions (Database Engine).

Interoperability

The following statements are valid in a function:

- Assignment statements.
- Control-of-Flow statements except TRY...CATCH statements.
- DECLARE statements defining local data variables and local cursors.
- SELECT statements that contain select lists with expressions that assign values to local variables.
- Cursor operations referencing local cursors that are declared, opened, closed, and deallocated in the function. Only FETCH statements that assign values to local variables using the INTO clause are allowed; FETCH statements that return data to the client are not allowed.
- INSERT, UPDATE, and DELETE statements modifying local table variables.
- EXECUTE statements calling extended stored procedures.
- For more information, see Creating User-Defined Functions (Database Engine).

Computed Column Interoperability

In SQL Server 2005 and later, functions have the following properties. The values of these properties determine whether functions can be used in computed columns that can be persisted or indexed.

Property	Description	Notes
IsDeterministic	Function is deterministic or nondeterministic.	Local data access is allowed in deterministic functions. For example, functions that always return the same result any time they are called by using a specific set of input values and with the

		same state of the database would be labeled deterministic.
IsPrecise	Function is precise or imprecise.	Imprecise functions contain operations such as floating point operations.
IsSystemVerified	The precision and determinism properties of the function can be verified by SQL Server.	
SystemDataAccess	Function accesses system data (system catalogs or virtual system tables) in the local instance of SQL Server.	
UserDataAccess	Function accesses user data in the local instance of SQL Server.	Includes user-defined tables and temp tables, but not table variables.

The precision and determinism properties of Transact-SQL functions are determined automatically by SQL Server. For more information, see User-Defined Function Design Guidelines. The data access and determinism properties of CLR functions can be specified by the user. For more information, see Overview of CLR Integration Custom Attributes.

To display the current values for these properties, use OBJECTPROPERTYEX.

A computed column that invokes a user-defined function can be used in an index when the user-defined function has the following property values:

- IsDeterministic = true
- IsSystemVerified = true (unless the computed column is persisted)
- UserDataAccess = false
- SystemDataAccess = false

For more information, see Creating Indexes on Computed Columns.

Calling Extended Stored Procedures from Functions

The extended stored procedure, when it is called from inside a function, cannot return result sets to the client. Any ODS APIs that return result sets to the client will return FAIL. The extended stored procedure could connect back to an instance of SQL Server; however, it should not try to join the same transaction as the function that invoked the extended stored procedure.

Similar to invocations from a batch or stored procedure, the extended stored procedure will be executed in the context of the Windows security account under which SQL Server is running. The owner of the stored procedure should consider this when giving EXECUTE permission on it to users.

Limitations and Restrictions

User-defined functions cannot be used to perform actions that modify the database state.

User-defined functions cannot contain an OUTPUT INTO clause that has a table as its target.

The following Service Broker statements cannot be included in the definition of a Transact-SQL user-defined function:

- BEGIN DIALOG CONVERSATION
- END CONVERSATION
- GET CONVERSATION GROUP
- MOVE CONVERSATION
- RECEIVE
- SEND

User-defined functions can be nested; that is, one user-defined function can call another. The nesting level is incremented when the called function starts execution, and decremented when the called function finishes execution. User-defined functions can be nested up to 32 levels. Exceeding the maximum levels of nesting causes the whole calling function chain to fail. Any reference to managed code from a Transact-SQL user-defined function counts as one level against the 32-level nesting limit. Methods invoked from within managed code do not count against this limit.

Using Sort Order in CLR Table-valued Functions

When using the ORDER clause in CLR table-valued functions, follow these guidelines:

- You must ensure that results are always ordered in the specified order. If the results are not in the specified order, SQL Server will generate an error message when the query is executed.
- If an ORDER clause is specified, the output of the table-valued function must be sorted according to the collation of the column (explicit or implicit). For example, if the column collation is Chinese (either specified in the DDL for the table-valued function or obtained from the database collation), the returned results must be sorted according to Chinese sorting rules.
- The ORDER clause, if specified, is always verified by SQL Server while returning results, whether or not it is used by the query processor to perform further optimizations. Only use the ORDER clause if you know it is useful to the query processor.
- The SQL Server query processor takes advantage of the ORDER clause automatically in following cases:
    - Insert queries where the ORDER clause is compatible with an index.
    - ORDER BY clauses that are compatible with the ORDER clause.
    - Aggregates, where GROUP BY is compatible with ORDER clause.
    - DISTINCT aggregates where the distinct columns are compatible with the ORDER clause.

The ORDER clause does not guarantee ordered results when a SELECT query is executed, unless ORDER BY is also specified in the query. See sys.function_order_columns for information on how to query for columns included in the sort-order for table-valued functions.

Metadata

The following table lists the system catalog views that you can use to return metadata about user-defined functions.

System View	Description
sys.sql_modules	Displays the definition of Transact-SQL user-defined functions. For example: `USE AdventureWorks2008R2;` `GO` `SELECT definition, type` `FROM sys.sql_modules AS m` `JOIN sys.objects AS o ON m.object_id = o.object_id` `AND type IN ('FN', 'IF', 'TF');` `GO` The definition of functions created by using the ENCRYPTION option cannot be viewed by using sys.sql_modules; however, other information about the encrypted functions is displayed.
sys.assembly_modules	Displays information about CLR user-defined functions.
sys.parameters	Displays information about the parameters defined in user-defined functions.
sys.sql_expression_dependencies	Displays the underlying objects referenced by a function.

Permissions

Requires CREATE FUNCTION permission in the database and ALTER permission on the schema in which the function is being created. If the function specifies a user-defined type, requires EXECUTE permission on the type.

Examples

A. Using a scalar-valued user-defined function that calculates the ISO week

The following example creates the user-defined function ISOweek. This function takes a date argument and calculates the ISO week number. For this function to calculate correctly, SET DATEFIRST 1 must be invoked before the function is called.

The example also shows using the EXECUTE AS clause to specify the security context in which a stored procedure can be executed. In the example, the option CALLER specifies that the procedure will be executed in the context of the user that calls it. The other options that you can specify are SELF, OWNER, and *user_name*.

Here is the function call. Notice that DATEFIRST is set to 1.

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID (N'dbo.ISOweek', N'FN') IS NOT NULL
    DROP FUNCTION dbo.ISOweek;
GO
CREATE FUNCTION dbo.ISOweek (@DATE datetime)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
     DECLARE @ISOweek int;
     SET @ISOweek= DATEPART(wk,@DATE)+1
          -DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104');
--Special cases: Jan 1-3 may belong to the previous year
     IF (@ISOweek=0)
          SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1
               AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS CHAR(2)))+1;
--Special case: Dec 29-31 may belong to the next year
     IF ((DATEPART(mm,@DATE)=12) AND
          ((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))
          SET @ISOweek=1;
     RETURN(@ISOweek);
END;
GO
SET DATEFIRST 1;
SELECT dbo.ISOweek(CONVERT(DATETIME,'12/26/2004',101)) AS 'ISO Week';
```

Here is the result set.

```
ISO Week
---------------
52
```

B. Creating an inline table-valued function

The following example returns an inline table-valued function. It returns three columns ProductID, Name and the aggregate of year-to-date totals by store as YTD Total for each product sold to the store.

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID (N'Sales.ufn_SalesByStore', N'IF') IS NOT NULL
    DROP FUNCTION Sales.ufn_SalesByStore;
GO
CREATE FUNCTION Sales.ufn_SalesByStore (@storeid int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'Total'
```

```
    FROM Production.Product AS P
    JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
    JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID = SD.SalesOrderID
    JOIN Sales.Customer AS C ON SH.CustomerID = C.CustomerID
    WHERE C.StoreID = @storeid
    GROUP BY P.ProductID, P.Name
);
GO
```

To invoke the function, run this query.

```
SELECT * FROM Sales.ufn_SalesByStore (602);
```

C. Creating a multi-statement table-valued function

The following example creates the table-valued function fn_FindReports(InEmpID). When supplied with a valid employee ID, the function returns a table that corresponds to all the employees that report to the employee either directly or indirectly. The function uses a recursive common table expression (CTE) to produce the hierarchical list of employees. For more information about recursive CTEs, see WITH common_table_expression.

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID (N'dbo.ufn_FindReports', N'TF') IS NOT NULL
    DROP FUNCTION dbo.ufn_FindReports;
GO
CREATE FUNCTION dbo.ufn_FindReports (@InEmpID INTEGER)
RETURNS @retFindReports TABLE
(
    EmployeeID int primary key NOT NULL,
    FirstName nvarchar(255) NOT NULL,
    LastName nvarchar(255) NOT NULL,
    JobTitle nvarchar(50) NOT NULL,
    RecursionLevel int NOT NULL
)
--Returns a result set that lists all the employees who report to the
--specific employee directly or indirectly.*/
AS
BEGIN
WITH EMP_cte(EmployeeID, OrganizationNode, FirstName, LastName, JobTitle, Rec
ursionLevel) -- CTE name and columns
    AS (
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName, p.LastNam
e, e.JobTitle, 0 -- Get the initial list of Employees for Manager n
        FROM HumanResources.Employee e
                    INNER JOIN Person.Person p
                    ON p.BusinessEntityID = e.BusinessEntityID
        WHERE e.BusinessEntityID = @InEmpID
```

```
        UNION ALL
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName, p.LastNam
e, e.JobTitle, RecursionLevel + 1 -- Join recursive member to anchor
        FROM HumanResources.Employee e
            INNER JOIN EMP_cte
            ON e.OrganizationNode.GetAncestor(1) = EMP_cte.OrganizationNode
                    INNER JOIN Person.Person p
                    ON p.BusinessEntityID = e.BusinessEntityID
        )
-- copy the required columns to the result of the function
   INSERT @retFindReports
   SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
   FROM EMP_cte
   RETURN
END;
GO
-- Example invocation
SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
FROM dbo.ufn_FindReports(1);

GO
```

D. Creating a CLR function

The following example assumes that the SQL Server Database Engine Samples are installed in the default location of the local computer and the StringManipulate.csproj sample application is compiled. For more information, see Considerations for Installing SQL Server Samples and Sample Databases.

The example creates CLR function len_s. Before the function is created, the assembly SurrogateStringFunction.dll is registered in the local database.

```
DECLARE @SamplesPath nvarchar(1024);
-- You may have to modify the value of this variable if you have
-- installed the sample in a location other than the default location.
SELECT @SamplesPath = REPLACE(physical_name, 'Microsoft SQL Server\MSSQL10_5.
MSSQLSERVER\MSSQL\DATA\master.mdf', 'Microsoft SQL Server\90\Samples\Engine\P
rogrammability\CLR\')
    FROM master.sys.database_files
    WHERE name = 'master';

CREATE ASSEMBLY [SurrogateStringFunction]
FROM @SamplesPath + 'StringManipulate\CS\StringManipulate\bin\debug\Surrogate
StringFunction.dll'
WITH PERMISSION_SET = EXTERNAL_ACCESS;
GO

CREATE FUNCTION [dbo].[len_s] (@str nvarchar(4000))
RETURNS bigint
```

```
AS EXTERNAL NAME [SurrogateStringFunction].[Microsoft.Samples.SqlServer.Surro
gateStringFunction].[LenS];
GO
```

For an example of how to create a CLR table-valued function, see CLR Table-Valued Functions.

Contents