

SQL Server 2012 Data Types



TechNet

This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. This document does not provide you with any legal rights to any intellectual property in any Microsoft product or product name. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes. © 2013 Microsoft. All rights reserved.

Terms of Use (<http://technet.microsoft.com/cc300389.aspx>) | Trademarks (<http://www.microsoft.com/library/toolbar/3.0/trademarks/en-us.mspx>)

Table Of Contents

Chapter 1

[Data Types \(Transact-SQL\)](#)
[int, bigint, smallint, and tinyint \(Transact-SQL\)](#)
[bit \(Transact-SQL\)](#)
[decimal and numeric \(Transact-SQL\)](#)
[money and smallmoney \(Transact-SQL\)](#)
[float and real \(Transact-SQL\)](#)
[date \(Transact-SQL\)](#)
[datetime2 \(Transact-SQL\)](#)
[datetime \(Transact-SQL\)](#)
[datetimeoffset \(Transact-SQL\)](#)
[smalldatetime \(Transact-SQL\)](#)
[time \(Transact-SQL\)](#)
[char and varchar \(Transact-SQL\)](#)
[ntext, text, and image \(Transact-SQL\)](#)
[nchar and nvarchar \(Transact-SQL\)](#)
[binary and varbinary \(Transact-SQL\)](#)
[cursor \(Transact-SQL\)](#)
[hierarchyid \(Transact-SQL\)](#)
[sql_variant \(Transact-SQL\)](#)
[table \(Transact-SQL\)](#)
[rowversion \(Transact-SQL\)](#)
[uniqueidentifier \(Transact-SQL\)](#)
[xml \(Transact-SQL\)](#)

Data Types (Transact-SQL)

SQL Server 2012

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 [CREATE TYPE \(Transact-SQL\)](#). 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.

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 \(Transact-SQL\)](#).
- 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 \(Transact-SQL\)](#).
- 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 \(Transact-SQL\)](#).

SQL Server provides data type synonyms for ISO compatibility. For more information, see [Data Type Synonyms \(Transact-SQL\)](#).

4 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	

See Also

- Reference
- [CREATE PROCEDURE \(Transact-SQL\)](#)
 - [CREATE TABLE \(Transact-SQL\)](#)
 - [DECLARE @local_variable \(Transact-SQL\)](#)
 - [EXECUTE \(Transact-SQL\)](#)
 - [Expressions \(Transact-SQL\)](#)
 - [Built-in Functions \(Transact-SQL\)](#)
 - [LIKE \(Transact-SQL\)](#)
 - [sp_droptype \(Transact-SQL\)](#)
 - [sp_help \(Transact-SQL\)](#)
 - [sp_rename \(Transact-SQL\)](#)

int, bigint, smallint, and tinyint (Transact-SQL)

SQL Server 2012

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 (9,223,372,036,854,775,807)	8 Bytes
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 autopermeterized or not.

Therefore, similar expressions in queries can sometimes produce different results. When a query is not autopermeterized, 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 autopermeterized, 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 autopermeterized 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 autopermeterized, because the results of the autopermeterized query will be truncated to fit into the **numeric (10, 0)** data type.

Converting Integer Data

When integers are implicitly converted to a character data type, if the integer is too large to fit into the character field, SQL Server enters ASCII character 42, the asterisk (*).

Integer constants greater than 2,147,483,647 are converted to the **decimal** data type, not the **bigint** data type. The following example shows that when the threshold value is exceeded, the data type of the result changes from an **int** to a **decimal**.

```
SELECT 2147483647 / 2 AS Resul t1, 2147483649 / 2 AS Resul t2 ;
```

Here is the result set.

Resul t1	Resul t2
1073741823	1073741824. 500000

See Also

Reference

[ALTER TABLE \(Transact-SQL\)](#)
[CAST and CONVERT \(Transact-SQL\)](#)
[CREATE TABLE \(Transact-SQL\)](#)
[DECLARE @local_variable \(Transact-SQL\)](#)
[SET @local_variable \(Transact-SQL\)](#)
[sys.types \(Transact-SQL\)](#)

bit (Transact-SQL)

SQL Server 2012

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.

Converting to bit promotes any nonzero value to 1.

See Also

Reference

[ALTER TABLE \(Transact-SQL\)](#)

[CAST and CONVERT \(Transact-SQL\)](#)

[CREATE TABLE \(Transact-SQL\)](#)

[Data Types \(Transact-SQL\)](#)

[DECLARE @local_variable \(Transact-SQL\)](#)

[SET @local_variable \(Transact-SQL\)](#)

[sys.types \(Transact-SQL\)](#)

Concepts

[Data Type Conversion \(Database Engine\)](#)

decimal and numeric (Transact-SQL)

SQL Server 2012

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 will 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 number of decimal digits that will be stored to the right of the decimal point. This number is subtracted from *p* to determine the maximum number of digits to the left 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, $0 \leq s \leq p$. Maximum storage sizes vary, based on the precision.

Precision	Storage bytes
1 - 9	5
10-19	9
20-28	13
29-38	17

▲ Converting decimal and numeric Data

For the **decimal** and **numeric** data types, SQL Server considers each specific combination of precision and scale as a different data type. For example, **decimal(5,5)** and **decimal(5,0)** are considered different data types.

In Transact-SQL statements, a constant with a decimal point is automatically converted into a **numeric** data value, using the minimum precision and scale necessary. For example, the constant 12.345 is converted into a **numeric** value with a precision of 5 and a scale of 3.

Converting from **decimal** or **numeric** to **float** or **real** can cause some loss of precision. Converting from **int**, **smallint**, **tinyint**, **float**, **real**, **money**, or **smallmoney** to either **decimal** or **numeric** can cause overflow.

By default, SQL Server uses rounding when converting a number to a **decimal** or **numeric** value with a lower precision and scale. However, if the SET ARITHABORT option is ON, SQL Server raises an error when overflow occurs. Loss of only precision and scale is not sufficient to raise an error.

When converting float or real values to decimal or numeric, the decimal value will never have more than 17 decimals. Any float value $< 5E-18$ will always convert as 0.

▲ See Also

Reference

[ALTER TABLE \(Transact-SQL\)](#)
[CAST and CONVERT \(Transact-SQL\)](#)
[CREATE TABLE \(Transact-SQL\)](#)
[DECLARE @local_variable \(Transact-SQL\)](#)
[SET @local_variable \(Transact-SQL\)](#)
[sys.types \(Transact-SQL\)](#)

money and smallmoney (Transact-SQL)

SQL Server 2012

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.

Use a period to separate partial monetary units, like cents, from whole monetary units. For example, 2.15 specifies 2 dollars and 15 cents.

These data types can use any one of the following currency symbols.

Symbol	Currency	Hexadecimal value
\$	Dollar sign	0024
¢	Cent sign	00A2
£	Pound sign	00A3
¤	Currency sign	00A4
¥	Yen sign	00A5
৳	Bengali Rupee mark	09F2
৳	Bengali Rupee sign	09F3
฿	Thai currency symbol Baht	0E3F
៛	Khmer currency symbol Riel	17DB
€	Euro-Currency sign	20A0
₡	Colon sign	20A1
₢	Cruzeiro sign	20A2
₣	French Franc sign	20A3
₤	Lira sign	20A4
₭	Mill sign	20A5
₮	Naira sign	20A6
₯	Peseta sign	20A7
₱	Rupee sign	20A8
₩	Won sign	20A9
₪	New Sheqel sign	20AA
₫	Dong sign	20AB
€	Euro sign	20AC
₭	Kip sign	20AD
₮	Tugrik sign	20AE
₯	Drachma sign	20AF
₰	German Penny sign	20B0
₱	Peso sign	20B1
₲	Rial sign	FDFC
₳	Small Dollar sign	FE69
₴	Fullwidth Dollar sign	FF04
¢	Fullwidth Cent sign	FFE0
£	Fullwidth Pound sign	FFE1
¥	Fullwidth Yen sign	FFE5
₩	Fullwidth Won sign	FFE6

Currency or monetary data does not need to be enclosed in single quotation marks ('). It is important to remember that while you can specify monetary values preceded by a currency symbol, SQL Server does not store any currency information associated with the symbol, it only stores the numeric value.

Converting money Data

When you convert to **money** from integer data types, units are assumed to be in monetary units. For example, the integer value of 4 is converted to the **money** equivalent of 4 monetary units.

The following example converts **smallmoney** and **money** values to **varchar** and **decimal** data types, respectively.

```
DECLARE @mymoney_sm smallmoney = 3148.29,
```

```
@mymoney    money = 3148.29;
SELECT  CAST(@mymoney_sm AS varchar) AS 'SM_MONEY varchar',
        CAST(@mymoney AS decimal)    AS 'MONEY DECIMAL';
```

Here is the result set.

SM_MONEY VARCHAR	MONEY DECIMAL
3148.29	3148
(1 row(s) affected)	

See Also

Reference


[ALTER TABLE \(Transact-SQL\)](#)
[CAST and CONVERT \(Transact-SQL\)](#)
[CREATE TABLE \(Transact-SQL\)](#)
[Data Types \(Transact-SQL\)](#)
[DECLARE @local_variable \(Transact-SQL\)](#)
[SET @local_variable \(Transact-SQL\)](#)
[sys.types \(Transact-SQL\)](#)

float and real (Transact-SQL)

SQL Server 2012

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 <i>n</i>
real	- 3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 to 3.40E + 38	4 Bytes

 [Transact-SQL Syntax Conventions](#)

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**.

<i>n</i> value	Precision	Storage size
1-24	7 digits	4 bytes
25-53	15 digits	8 bytes

Note	
SQL Server treats <i>n</i> as one of two possible values. If $1 \leq n \leq 24$, <i>n</i> is treated as 24 . If $25 \leq n \leq 53$, <i>n</i> 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)**.

Converting float and real Data

Values of **float** are truncated when they are converted to any integer type.

When you want to convert from **float** or **real** to character data, using the STR string function is usually more useful than CAST(). This is because STR enables more control over formatting. For more information, see [STR \(Transact-SQL\)](#) and [Built-in Functions \(Transact-SQL\)](#).

Conversion of **float** values that use scientific notation to **decimal** or **numeric** is restricted to values of precision 17 digits only. Any value with precision higher than 17 rounds to zero.

See Also

Reference

[ALTER TABLE \(Transact-SQL\)](#)

[CAST and CONVERT \(Transact-SQL\)](#)

[CREATE TABLE \(Transact-SQL\)](#)

[Data Types \(Transact-SQL\)](#)

[DECLARE @local_variable \(Transact-SQL\)](#)

[SET @local_variable \(Transact-SQL\)](#)

Concepts

[Data Type Conversion \(Database Engine\)](#)

date (Transact-SQL)

SQL Server 2012

Defines a date.

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 that follows.
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 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	[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 for interpreting two-digit years as four-digit years, use the Configure the two digit year cutoff Server Configuration 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		Description			
YYYY-MM-DD YYYYMMDD		Same as the SQL standard. This is the only format that is defined as an international standard.			
Unseparated		Description			
[yy]ymmdd 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 2012 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): <ul style="list-style-type: none"> • 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.

Backward Compatibility for Down-level Clients

Some down-level clients do not support the **time**, **date**, **datetime2** and **datetimeoffset** data types. The following table shows the type mapping between an up-level instance of SQL Server and down-level clients.

SQL Server 2012 data type	Default string literal format passed to down-level client	Down-level ODBC	Down-level OLEDB	Down-level JDBC	Down-level SQLCLIENT
time	hh:mm:ss[.nnnnnnn]	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
date	YYYY-MM-DD	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+ -]hh:mm	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString

Converting Date and Time Data

When you convert to date and time data types, SQL Server rejects all values it cannot recognize as dates or times. For information about using the CAST and CONVERT functions with date and time data, see [CAST and CONVERT \(Transact-SQL\)](#).

Converting date to Other Date and Time Types

The following table describes what occurs when a **date** data type is converted to other date and time data types.

--	--

Data type to convert to	Conversion details
time(n)	The conversion fails, and error message 206 is raised: "Operand type clash: date is incompatible with time".
datetime	<p>The date is copied and the time component is set to 00:00:00.000. The following code shows the results of converting a date value to a datetime value.</p> <pre>DECLARE @date date= '12-10-25'; DECLARE @datetime datetime= @date; SELECT @date AS '@date', @datetime AS '@datetime'; --Result --@date @datetime ----- --2025-12-10 2025-12-10 00:00:00.000 -- --(1 row(s) affected)</pre>
smalldatetime	<p>When the date value is in the range of a smalldatetime, the date component is copied and the time component is set to 00:00:00. When the date value is outside the range of a smalldatetime value, error message 242 is raised: "The conversion of a date data type to a smalldatetime data type results in an out-of-range value;and the smalldatetime value is set to NULL.</p> <p>The following code shows the results of converting a date value to a small datetime value.</p> <pre>DECLARE @date date= '1912-10-25'; DECLARE @smalldatetime smalldatetime = @date; SELECT @date AS '@date', @smalldatetime AS '@smalldatetime'; --Result --@date @smalldatetime ----- --1912-10-25 1912-10-25 00:00:00 -- --(1 row(s) affected)</pre>
datetimeoffset(n)	<p>The date is copied, and the time is set to 00:00:00.0000000 +00:00.</p> <p>The following code shows the results of converting a date value to a datetimeoffset(3) value.</p> <pre>DECLARE @date date = '1912-10-25'; DECLARE @datetimeoffset datetimeoffset(3) = @date; SELECT @date AS '@date', @datetimeoffset AS '@datetimeoffset'; --Result --@date @datetimeoffset ----- --1912-10-25 1912-10-25 00:00:00.000 +00:00 -- --(1 row(s) affected)</pre>
datetime2(n)	<p>The date component is copied, and the time component is set to 00:00:00.00 regardless of the value of (n).</p> <p>The following code shows the results of converting a date value to a datetime2(3) value.</p> <pre>DECLARE @date date = '1912-10-25'; DECLARE @datetime2 datetime2(3) = @date; SELECT @date AS '@date', @datetime2 AS '@datetime2(3)'; --Result --@date @datetime2(3) ----- --1912-10-25 1912-10-25 00:00:00.00 -- --(1 row(s) affected)</pre>

Converting String Literals to date

Conversions from string literals to date and time types are permitted if all parts of the strings are in valid formats. Otherwise, a runtime error is raised. Implicit conversions or explicit conversions that do not specify a style, from date and time types to string literals will be in the default format of the current session. The following table shows the rules for converting a string literal to the **date** data type.

--	--

Input string literal	date
ODBC DATE	ODBC string literals are mapped to the datetime data type. Any assignment operation from ODBC DATETIME literals into a date type will cause an implicit conversion between datetime and this type as defined by the conversion rules.
ODBC TIME	See previous ODBC DATE rule.
ODBC DATETIME	See previous ODBC DATE rule.
DATE only	Trivial
TIME only	Default values are supplied.
TIMEZONE only	Default values are supplied.
DATE + TIME	The DATE part of the input string is used.
DATE + TIMEZONE	Not allowed.
TIME + TIMEZONE	Default values are supplied.
DATE + TIME + TIMEZONE	The DATE part of local DATETIME will be used.

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 small datetime) AS
    'small datetime'
, 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

See Also

Reference
[CAST and CONVERT \(Transact-SQL\)](#)

datetime2 (Transact-SQL)

SQL Server 2012

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.

datetime2 Description

Property	Value
Syntax	datetime2 [(<i>fractional seconds precision</i>)]
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 that follows.
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 \(Transact-SQL\)](#) or [TYPEPROPERTY \(Transact-SQL\)](#). Precision and scale are variable for some date and time data types. To obtain the precision and scale for a column, see [COLUMNPROPERTY \(Transact-SQL\)](#), [COL_LENGTH \(Transact-SQL\)](#), or [sys.columns \(Transact-SQL\)](#).

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 \(Transact-SQL\)](#) and [time \(Transact-SQL\)](#).

ISO 8601	Descriptions
YYYY-MM-DDThh:mm:ss[.nnnnnnnn] YYYY-MM-DDThh:mm:ss[.nnnnnnnn]	This format is not affected by the SET LANGUAGE and SET 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: <ul style="list-style-type: none">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 2012, 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**.

Backward Compatibility for Down-level Clients

Some down-level clients do not support the **time**, **date**, **datetime2** and **datetimeoffset** data types. The following table shows the type mapping between an up-level instance of SQL Server and down-level clients.

SQL Server 2012 data type	Default string literal format passed to down-level client	Down-level ODBC	Down-level OLEDB	Down-level JDBC	Down-level SQLCLIENT
time	hh:mm:ss[.nnnnnnn]	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
date	YYYY-MM-DD	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+ -]hh:mm	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString

Converting Date and Time Data

When you convert to date and time data types, SQL Server rejects all values it cannot recognize as dates or times. For information about using the CAST and CONVERT functions with date and time data, see [CAST and CONVERT \(Transact-SQL\)](#)

Converting datetime2 Data Type to Other Date and Time Types

The following table describes what occurs when a **datetime2** data type is converted to other date and time data types.

Data type to convert to	Conversion details
date	<p>The year, month, and day are copied.</p> <p>The following code shows the results of converting a datetime2(4) value to a date value.</p> <pre>DECLARE @datetime2 datetime2(4) = '12-10-25 12:32:10.1234'; DECLARE @date date = @datetime2; SELECT @datetime2 AS '@datetime2', @date AS 'date'; --Result --@datetime2 date ----- --2025-12-10 12:32:10.1234 2025-12-10 -- --(1 row(s) affected)</pre>
time(n)	<p>The hour, minute, second, and fractional seconds are copied if the scale is the same and rounded up if the scale is different.</p> <p>The following code shows the results of converting a datetime2(4) value to a time(3) value.</p> <pre>DECLARE @datetime2 datetime2(4) = '12-10-25 12:32:10.1237'; DECLARE @time time(3) = @datetime2; SELECT @datetime2 AS '@datetime2', @time AS 'time(3)'; --Result --@datetime2 time(3) ----- --2025-12-10 12:32:10.1234 12:32:10.124 -- --(1 row(s) affected)</pre>
datetime	<p>The date and time values are copied. When the fractional precision of the datetime(n) value is greater than three digits, the value is truncated.</p> <p>The following code shows the results of converting a datetime2 value to a datetime value.</p> <pre>DECLARE @datetime2 datetime2 = '12-10-25 12:32:10.1234567'</pre>

	<pre>DECLARE @datetime datetime = @datetime2; SELECT @datetime2 AS 'datetime2', @datetime AS 'datetime'; --Result --datetime2 datetime ----- datetime2 datetime ----- 2025-12-10 12: 32: 10. 12 2025-12-10 12: 32: 10. 123 -- --(1 row(s) affected)</pre>
smalldatetime	<p>The date and hours are copied. The minutes are rounded up with respect to the seconds and the seconds are set to 0. The following code shows the results of converting a datetime2 value to a small datetime value.</p> <pre>DECLARE @datetime2 datetime2 = '12-10-25 12: 32: 30. 9234567'; DECLARE @smalldatetime smalldatetime = @datetime2; SELECT @datetime2 AS 'datetime2', @smalldatetime AS 'small datetime'; --Result datetime2 smalldatetime ----- 2025-12-10 12: 32: 30. 92 2025-12-10 12: 33: 00 (1 row(s) affected)</pre>
datetimeoffset(n)	<p>The datetime2(n) value is copied to the datetimeoffset(n) value. The time zone offset is set to +00:0. When the precision of the datetime2(n) value is greater than the precision of datetimeoffset(n) value, the value is rounded up to fit. The following code shows the results of converting a datetime2(5) value to a datetimeoffset(3) value.</p> <pre>D DECLARE @datetime2 datetime2(3) = '12-10-25 12: 32: 10. 12999'; DECLARE @datetimeoffset datetimeoffset(2) = @datetime2; SELECT @datetime2 AS 'datetime2', @datetimeoffset AS 'datetimeoffset(2)'; --Result --datetime2 datetimeoffset(2) ----- 2025-12-10 12: 32: 10. 13 2025-12-10 12: 32: 10. 13 +00: 00 -- --(1 row(s) affected)</pre>

Converting String Literals to datetime2

Conversions from string literals to date and time types are permitted if all parts of the strings are in valid formats. Otherwise, a runtime error is raised. Implicit conversions or explicit conversions that do not specify a style, from date and time types to string literals will be in the default format of the current session. The following table shows the rules for converting a string literal to the **datetime2** data type.

Input string literal	datetime2(n)
ODBC DATE	ODBC string literals are mapped to the datetime data type. Any assignment operation from ODBC DATETIME literals into datetime2 types will cause an implicit conversion between datetime and this type as defined by the conversion rules.
ODBC TIME	See previous ODBC DATE rule.
ODBC DATETIME	See previous ODBC DATE rule.
DATE only	The TIME part defaults to 00:00:00.
TIME only	The DATE part defaults to 1900-1-1.
TIMEZONE only	Default values are supplied.
DATE + TIME	Trivial
DATE + TIMEZONE	Not allowed.
TIME + TIMEZONE	The DATE part defaults to 1900-1-1. TIMEZONE input is ignored.
DATE + TIME + TIMEZONE	The local DATETIME will be used.

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

See Also

Reference

[CAST and CONVERT \(Transact-SQL\)](#)

datetime (Transact-SQL)

SQL Server 2012

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.

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: [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	<p>You can specify date data with a numeric month specified. For example, 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:</p> <ul style="list-style-type: none">number separator number separator <i>number</i> [<i>time</i>] [<i>time</i>] <p>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.</p> <p>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.</p>

(ydm) [19]96/[0]4/15 -- (ymd) Time formats: 14:30 14:30[:20:999] 14:30[:20.9] 4am 4 PM										
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]	<p>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.</p> <p>Here are some guidelines for using alphabetical date formats:</p> <ul style="list-style-type: none">• 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 Configure the two digit year cutoff Server Configuration Option 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. <p>The SET DATEFORMAT session setting is not applied when you specify the month in alphabetical form.</p>									
ISO 8601	Description									
YYYY-MM-DDThh:mm:ss[.mmm] YYYYMMDD[hh:mm:ss[.mmm]]	<p>Examples:</p> <ul style="list-style-type: none">• 2004-05-23T14:25:10• 2004-05-23T14:25:10.487 <p>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.</p> <p>The brackets indicate that the fraction of second component is optional. The time component is specified in the 24-hour format.</p> <p>The T indicates the start of the time part of the datetime value.</p> <p>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.</p>									
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' }	<p>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.</p> <p>ODBC timestamp escape sequences are of the format: { <i>literal_type</i> 'constant_value' }:</p> <ul style="list-style-type: none">• <i>literal_type</i> specifies the type of the escape sequence. Timestamps have three <i>literal_type</i> specifiers:<ul style="list-style-type: none">◦ d = date only◦ t = time only◦ ts = timestamp (time + date)• 'constant_value' is the value of the escape sequence. <i>constant_value</i> must follow these formats for each <i>literal_type</i>. <table><tr><th>literal_type</th><th>constant_value format</th></tr><tr><td>d</td><td>yyyy-mm-dd</td></tr><tr><td>t</td><td>hh:mm:ss[.fff]</td></tr><tr><td>ts</td><td>yyyy-mm-dd hh:mm:ss[.fff]</td></tr></table>		literal_type	constant_value format	d	yyyy-mm-dd	t	hh:mm:ss[.fff]	ts	yyyy-mm-dd hh:mm:ss[.fff]
literal_type	constant_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	1998-01-01 23:59:59.997

01/01/98 23:59:59.996 01/01/98 23:59:59.997 01/01/98 23:59:59.998	
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.

Converting Date and Time Data

When you convert to date and time data types, SQL Server rejects all values it cannot recognize as dates or times. For information about using the CAST and CONVERT functions with date and time data, see [CAST and CONVERT \(Transact-SQL\)](#).

Converting datetime to Other Date and Time Types

The following table describes what occurs when a **datetime** data type is converted to other date and time data types.

Data type to convert to	Conversion details
date	<p>The year month and day are copied. The time component is set to 00:00:00.000.</p> <p>The following code shows the results of converting a date value to a datetime value.</p> <div><pre>DECLARE @date date = '12-21-05'; DECLARE @datetime datetime = @date; SELECT @datetime AS '@datetime', @date AS '@date'; --Result --@datetime @date ----- --2005-12-21 00:00:00.000 2005-12-21</pre></div>
time(n)	<p>The time component is copied, and the date component is set to '1900-01-01'. When the fractional precision of the time(n) value greater than three digits, the value will be truncated to fit.</p> <p>The following example shows the results of converting a time(4) value to a datetime value.</p> <div><pre>DECLARE @time time(4) = '12:10:05.1237'; DECLARE @datetime datetime = @time; SELECT @datetime AS '@datetime', @time AS '@time'; --Result --@datetime @time ----- --1900-01-01 12:10:05.123 12:10:05.1237 -- --(1 row(s) affected)</pre></div>
smalldatetime	<p>The hours and minutes are copied. The seconds and fractional seconds are set to 0.</p> <p>The following code shows the results of converting a smalldatetime value to a datetime value.</p> <div><pre>DECLARE @smalldatetime smalldatetime = '12-01-01 12:32'; DECLARE @datetime datetime = @smalldatetime; SELECT @datetime AS '@datetime', @smalldatetime AS '@smalldatetime'; --Result --@datetime @smalldatetime ----- --2001-12-01 12:32:00.000 2001-12-01 12:32:00 -- --(1 row(s) affected)</pre></div>

datetimeoffset(n)	<p>The date and time components are copied. The time zone is truncated. When the fractional precision of the datetimeoffset(n) value is greater than three digits, the value will be truncated.</p> <p>The following example shows the results of converting a datetimeoffset(4) value to a datetime value.</p> <div><pre>DECLARE @datetimeoffset datetimeoffset(4) = '1968-10-23 12:45:37.1234 +10:0'; DECLARE @datetime datetime = @datetimeoffset; SELECT @datetime AS '@datetime', @datetimeoffset AS '@datetimeoffset'; --Result --@datetime @datetimeoffset ----- --1968-10-23 12:45:37.123 1968-10-23 12:45:37.1237 +01:0 -- --(1 row(s) affected)</pre></div>
datetime2(n)	<p>The date and time are copied. When the fractional precision of the datetime2(n) value is greater than three digits, the value will be truncated.</p> <p>The following example shows the results of converting a datetime2(4) value to a datetime value.</p> <div><pre>DECLARE @datetime2 datetime2(4) = '1968-10-23 12:45:37.1237'; DECLARE @datetime datetime = @datetime2; SELECT @datetime AS '@datetime', @datetime2 AS '@datetime2'; --Result --@datetime @datetime2 ----- --1968-10-23 12:45:37.123 1968-10-23 12:45:37.1237 -- --(1 row(s) affected)</pre></div>

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

See Also

Reference
[CAST and CONVERT \(Transact-SQL\)](#)

datetimeoffset (Transact-SQL)

SQL Server 2012

Defines a date that is combined with a time of a day that has time zone awareness and is based on a 24-hour clock.

datetimeoffset Description

Property	Value																																								
Syntax	datetimeoffset [(<i>fractional seconds precision</i>)]																																								
Usage	DECLARE @MyDatetimeoffset datetimeoffset(7) CREATE TABLE Table1 (Column1 datetimeoffset(7))																																								
Default string literal formats (used for down-level client)	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [(+ -)hh:mm] For more information, see the "Backward Compatibility for Down-level Clients" section that follows.																																								
Date range	0001-01-01 through 9999-12-31 January 1,1 A.D. through December 31, 9999 A.D.																																								
Time range	00:00:00 through 23:59:59.9999999																																								
Time zone offset range	-14:00 through +14:00																																								
Element ranges	YYYY is four digits, ranging from 0001 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 seven digits, ranging from 0 to 9999999, that represent the fractional seconds. hh is two digits that range from -14 to +14. mm is two digits that range from 00 to 59.																																								
Character length	26 positions minimum (YYYY-MM-DD hh:mm:ss {+ -}hh:mm) to 34 maximum (YYYY-MM-DD hh:mm:ss.nnnnnnn {+ -}hh:mm)																																								
Precision, scale	<table><tr><th>Specified scale</th><th>Result (precision, scale)</th><th>Column length (bytes)</th><th>Fractional seconds precision</th></tr><tr><td>datetimeoffset</td><td>(34,7)</td><td>10</td><td>7</td></tr><tr><td>datetimeoffset(0)</td><td>(26,0)</td><td>8</td><td>0-2</td></tr><tr><td>datetimeoffset(1)</td><td>(28,1)</td><td>8</td><td>0-2</td></tr><tr><td>datetimeoffset(2)</td><td>(29,2)</td><td>8</td><td>0-2</td></tr><tr><td>datetimeoffset(3)</td><td>(30,3)</td><td>9</td><td>3-4</td></tr><tr><td>datetimeoffset(4)</td><td>(31,4)</td><td>9</td><td>3-4</td></tr><tr><td>datetimeoffset(5)</td><td>(32,5)</td><td>10</td><td>5-7</td></tr><tr><td>datetimeoffset(6)</td><td>(33,6)</td><td>10</td><td>5-7</td></tr><tr><td>datetimeoffset(7)</td><td>(34,7)</td><td>10</td><td>5-7</td></tr></table>	Specified scale	Result (precision, scale)	Column length (bytes)	Fractional seconds precision	datetimeoffset	(34,7)	10	7	datetimeoffset(0)	(26,0)	8	0-2	datetimeoffset(1)	(28,1)	8	0-2	datetimeoffset(2)	(29,2)	8	0-2	datetimeoffset(3)	(30,3)	9	3-4	datetimeoffset(4)	(31,4)	9	3-4	datetimeoffset(5)	(32,5)	10	5-7	datetimeoffset(6)	(33,6)	10	5-7	datetimeoffset(7)	(34,7)	10	5-7
Specified scale	Result (precision, scale)	Column length (bytes)	Fractional seconds precision																																						
datetimeoffset	(34,7)	10	7																																						
datetimeoffset(0)	(26,0)	8	0-2																																						
datetimeoffset(1)	(28,1)	8	0-2																																						
datetimeoffset(2)	(29,2)	8	0-2																																						
datetimeoffset(3)	(30,3)	9	3-4																																						
datetimeoffset(4)	(31,4)	9	3-4																																						
datetimeoffset(5)	(32,5)	10	5-7																																						
datetimeoffset(6)	(33,6)	10	5-7																																						
datetimeoffset(7)	(34,7)	10	5-7																																						
Storage size	10 bytes, fixed is the default with the default of 100ns fractional second precision.																																								
Accuracy	100 nanoseconds																																								
Default value	1900-01-01 00:00:00 00:00																																								
Calendar	Gregorian																																								
User-defined fractional second precision	Yes																																								
Time zone offset aware and preservation	Yes																																								
Daylight saving aware	No																																								

Supported String Literal Formats for datetimeoffset

The following table lists the supported ISO 8601 string literal formats for **datetimeoffset**. For information about alphabetical, numeric, unseparated and time formats for the date and time parts of **datetimeoffset**, see [date \(Transact-SQL\)](#) and [time \(Transact-SQL\)](#).

ISO 8601	Description
YYYY-MM-DDThh:mm:ss[.nnnnnnn][+ -]hh:mm]	These two formats are not affected by the SET LANGUAGE and SET DATEFORMAT session locale settings. Spaces are not allowed between the datetimeoffset and the datetime parts.
YYYY-MM-DDThh:mm:ss[.nnnnnnn]Z (UTC)	This format by ISO definition indicates the datetime portion should be expressed in Coordinated Universal Time (UTC). For example, 1999-12-12 12:30:30.12345 -07:00 should be represented as 1999-12-12 19:30:30.12345Z.

Time Zone Offset

A time zone offset specifies the zone offset from UTC for a **time** or **datetime** value. The time zone offset can be represented as [+|-] hh:mm:

- hh is two digits that range from 00 to 14 and represent the number of hours in the time zone offset.
- mm is two digits, ranging from 00 to 59, that represent the number of additional minutes in the time zone offset.
- + (plus) or – (minus) is the mandatory sign for a time zone offset. This indicates whether the time zone offset is added or subtracted from the UTC time to obtain the local time. The valid range of time zone offset is from -14:00 to +14:00.

The time zone offset range follows the W3C XML standard for XSD schema definition and is slightly different from the SQL 2003 standard definition, 12:59 to +14:00.

The optional type parameter *fractional seconds precision* specifies the number of digits for the fractional part of the seconds. This value can be an integer with 0 to 7 (100 nanoseconds). The default *fractional seconds precision* is 100ns (seven digits for the fractional part of the seconds).

The data is stored in the database and processed, compared, sorted, and indexed in the server as in UTC. The time zone offset will be preserved in the database for retrieval.

The given time zone offset will be assumed to be daylight saving time (DST) aware and adjusted for any given **datetime** that is in the DST period.

For **datetimeoffset** type, both UTC and local (to the persistent or converted time zone offset) **datetime** value will be validated during insert, update, arithmetic, convert, or assign operations. The detection of any invalid UTC or local (to the persistent or converted time zone offset) **datetime** value will raise an invalid value error. For example, 9999-12-31 10:10:00 is valid in UTC, but overflow in local time to the time zone offset +13:50.

ANSI and ISO 8601 Compliance

The ANSI and ISO 8601 Compliance sections of the [date](#) and [time](#) topics apply to **datetimeoffset**.

Backward Compatibility for Down-level Clients

Some down-level clients do not support the **time**, **date**, **datetime2** and **datetimeoffset** data types. The following table shows the type mapping between an up-level instance of SQL Server and down-level clients.

SQL Server 2012 data type	Default string literal format passed to down-level client	Down-level ODBC	Down-level OLEDB	Down-level JDBC	Down-level SQLCLIENT
time	hh:mm:ss[.nnnnnnn]	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
date	YYYY-MM-DD	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+ -]hh:mm	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString

Converting Date and Time Data

When you convert to date and time data types, SQL Server rejects all values it cannot recognize as dates or times. For information about using the CAST and CONVERT functions with date and time data, see [CAST and CONVERT \(Transact-SQL\)](#)

Converting datetimeoffset Data Type to Other Date and Time Types

The following table describes what occurs when a **datetimeoffset** data type is converted to other date and time data types.

--	--

Data type to convert to	Conversion details
date	<p>The year, month, and day are copied.</p> <p>The following code shows the results of converting a datetimeoffset(4) value to a date value.</p> <pre> DECLARE @datetimeoffset datetimeoffset(4) = '12-10-25 12:32:10 +01:00'; DECLARE @date date = @datetimeoffset; SELECT @datetimeoffset AS '@datetimeoffset ', @date AS 'date'; --Result --@datetimeoffset date ----- --2025-12-10 12:32:10.0000 +01:0 2025-12-10 -- --(1 row(s) affected)</pre>
time(n)	<p>The hour, minute, second, and fractional seconds are copied. The time zone value is truncated. When the precision of the datetimeoffset(n) value is greater than the precision of the time(n) value, the value is rounded up.</p> <p>The following code shows the results of converting a datetimeoffset(4) value to a time(3) value.</p> <pre> DECLARE @datetimeoffset datetimeoffset(4) = '12-10-25 12:32:10.1237 +01:0'; DECLARE @time time(3) = @datetimeoffset; SELECT @datetimeoffset AS '@datetimeoffset ', @time AS 'time'; --Result --@datetimeoffset time ----- -- 2025-12-10 12:32:10.1237 +01:00 12:32:10.124 -- --(1 row(s) affected)</pre>
datetime	<p>The date and time values are copied, and the time zone is truncated. When the fractional precision of the datetimeoffset(n) value is greater than three digits, the value is truncated.</p> <p>The following code shows the results of converting a datetimeoffset(4) value to a datetime value.</p> <pre> DECLARE @datetimeoffset datetimeoffset(4) = '12-10-25 12:32:10.1237 +01:0'; DECLARE @datetime datetime = @datetimeoffset; SELECT @datetimeoffset AS '@datetimeoffset ', @datetime AS 'datetime'; --Result --@datetimeoffset datetime ----- --2025-12-10 12:32:10.1237 +01:0 2025-12-10 12:32:10.123 -- --(1 row(s) affected)</pre>
smalldatetime	<p>The date and hours are copied. The minutes are rounded up with respect to the seconds value and seconds are set to 0.</p> <p>The following code shows the results of converting a datetimeoffset(3) value to a smalldatetime value.</p> <pre> DECLARE @datetimeoffset datetimeoffset(3) = '1912-10-25 12:24:32 +10:0'; DECLARE @smalldatetime smalldatetime = @datetimeoffset; SELECT @datetimeoffset AS '@datetimeoffset', @smalldatetime AS '@smalldatetime'; --Result --@datetimeoffset @smalldatetime ----- --1912-10-25 12:24:32.000 +10:00 1912-10-25 12:25:00 -- --(1 row(s) affected)</pre>
datetime2(n)	<p>The date and time are copied to the datetime2 value, and the time zone is truncated. When the precision of the datetime2(n) value is greater than the precision of the datetimeoffset(n) value, the fractional seconds are truncated to fit.</p> <p>The follow code shows the results of converting a datetimeoffset(4) value to a datetime2(3) value.</p> <pre> DECLARE @datetimeoffset datetimeoffset(4) = '1912-10-25 12:24:32.1277 +10:0'; DECLARE @datetime2 datetime2(3) = @datetimeoffset;</pre>

```
SELECT @datetimeoffset AS '@datetimeoffset', @datetime2 AS '@datetime2';

--Result
@datetimeoffset          @datetime2
-----
1912-10-25 12:24:32.1277 +10:00    1912-10-25 12:24:32.12

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

Converting String Literals to datetimeoffset

Conversions from string literals to date and time types are permitted if all parts of the strings are in valid formats. Otherwise, a runtime error is raised. Implicit conversions or explicit conversions that do not specify a style, from date and time types to string literals will be in the default format of the current session. The following table shows the rules for converting a string literal to the **datetimeoffset** data type.

Input string literal	datetimeoffset(n)
ODBC DATE	ODBC string literals are mapped to the datetime data type. Any assignment operation from ODBC DATETIME literals into datetimeoffset types will cause an implicit conversion between datetime and this type as defined by the conversion rules.
ODBC TIME	See previous ODBC DATE rule.
ODBC DATETIME	See previous ODBC DATE rule.
DATE only	The TIME part defaults to 00:00:00. The TIMEZONE defaults to +00:00.
TIME only	The DATE part defaults to 1900-1-1. The TIMEZONE will default to +00:00.
TIMEZONE only	Default values are supplied
DATE + TIME	The TIMEZONE defaults to +00:00.
DATE + TIMEZONE	Not allowed
TIME + TIMEZONE	The DATE part defaults to 1900-1-1.
DATE + TIME + TIMEZONE	Trivial

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'
  ,CAST(' 2007-05-08 12:35:29.1234567+12:15' AS datetimeoffset(7)) AS
    'datetimeoffset ISO8601';
```

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

See Also

Reference

[CAST and CONVERT \(Transact-SQL\)](#)

smalldatetime (Transact-SQL)

SQL Server 2012

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.

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.

Converting Date and Time Data

When you convert to date and time data types, SQL Server rejects all values it cannot recognize as dates or times. For information about using the CAST and CONVERT functions with date and time data, see [CAST and CONVERT \(Transact-SQL\)](#).

Converting smalldatetime to Other Date and Time Types

The following table describes what occurs when a **smalldatetime** data type is converted to other date and time data types.

Data type to convert to	Conversion details

date	<p>The year, month, and day are copied.</p> <p>The following code shows the results of converting a small datetime value to a date value.</p> <pre> DECLARE @small datetime small datetime = '1955-12-13 12:43:10'; DECLARE @date date = @small datetime SELECT @small datetime AS 'small datetime', @date AS 'date'; --Result --small datetime date ----- --1955-12-13 12:43:00 1955-12-13 -- --(1 row(s) affected) </pre>
time(n)	<p>The hours, minutes, and seconds are copied. The fractional seconds are set to 0.</p> <p>The following code shows the results of converting a small datetime value to a time(4) value.</p> <pre> DECLARE @small datetime small datetime = '1955-12-13 12:43:10'; DECLARE @time time(4) = @small datetime; SELECT @small datetime AS 'small datetime', @time AS 'time'; --Result --small datetime time ----- --1955-12-13 12:43:00 12:43:00.0000 -- --(1 row(s) affected) </pre>
datetime	<p>The small datetime value is copied to the datetime value. The fractional seconds are set to 0.</p> <p>The following code shows the results of converting a small datetime value to a datetime value.</p> <pre> DECLARE @small datetime small datetime = '1955-12-13 12:43:10'; DECLARE @datetime datetime = @small datetime; SELECT @small datetime AS 'small datetime', @datetime AS 'datetime'; --Result --small datetime datetime ----- --1955-12-13 12:43:00 1955-12-13 12:43:00.000 -- --(1 row(s) affected) </pre>
datetimeoffset(n)	<p>The small datetime value is copied to the datetimeoffset(n) value. The fractional seconds are set to 0, and the time zone offset is set to +00:0.</p> <p>The following code shows the results of converting a small datetime value to a datetimeoffset(4) value.</p> <pre> DECLARE @small datetime small datetime = '1955-12-13 12:43:10'; DECLARE @datetimeoffset datetimeoffset(4) = @small datetime; SELECT @small datetime AS 'small datetime', @datetimeoffset AS 'datetimeoffset(4)'; --Result --small datetime datetimeoffset(4) ----- --1955-12-13 12:43:00 1955-12-13 12:43:00.0000 +00:0 -- --(1 row(s) affected) </pre>
datetime2(n)	<p>The small datetime value is copied to the datetime2(n) value. The fractional seconds are set to 0.</p> <p>The following code shows the results of converting a small datetime value to a datetime2(4) value.</p> <pre> DECLARE @small datetime small datetime = '1955-12-13 12:43:10'; DECLARE @datetime2 datetime2(4) = @small datetime; SELECT @small datetime AS 'small datetime', @datetime2 AS 'datetime2(4)'; --Result --small datetime datetime2(4) ----- --1955-12-13 12:43:00 1955-12-13 12:43:00.0000 </pre>

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

Examples

A. Casting string literals with seconds to smalldatetime

The following example compares the conversion of seconds in string literals to `small datetime`.

```
SELECT
    CAST(' 2007-05-08 12:35:29' AS small datetime)
, CAST(' 2007-05-08 12:35:30' AS small datetime)
, CAST(' 2007-05-08 12:59:59.998' AS small datetime);
```

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 small datetime) AS
    'small datetime'
, 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

See Also

Reference

[CAST and CONVERT \(Transact-SQL\)](#)

time (Transact-SQL)

SQL Server 2012

Defines a time of a day. The time is without time zone awareness and is based on a 24-hour clock.

time Description

Property	Value																																								
Syntax	time [(<i>fractional second precision</i>)]																																								
Usage	DECLARE @MyTime time (7) CREATE TABLE Table1 (Column1 time (7))																																								
<i>fractional seconds precision</i>	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).																																								
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 that follows..																																								
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)	<table><tr><th>Specified scale</th><th>Result (precision, scale)</th><th>Column length (bytes)</th><th>Fractional seconds precision</th></tr><tr><td>time</td><td>(16,7)</td><td>5</td><td>7</td></tr><tr><td>time(0)</td><td>(8,0)</td><td>3</td><td>0-2</td></tr><tr><td>time(1)</td><td>(10,1)</td><td>3</td><td>0-2</td></tr><tr><td>time(2)</td><td>(11,2)</td><td>3</td><td>0-2</td></tr><tr><td>time(3)</td><td>(12,3)</td><td>4</td><td>3-4</td></tr><tr><td>time(4)</td><td>(13,4)</td><td>4</td><td>3-4</td></tr><tr><td>time(5)</td><td>(14,5)</td><td>5</td><td>5-7</td></tr><tr><td>time(6)</td><td>(15,6)</td><td>5</td><td>5-7</td></tr><tr><td>time(7)</td><td>(16,7)</td><td>5</td><td>5-7</td></tr></table>	Specified scale	Result (precision, scale)	Column length (bytes)	Fractional seconds precision	time	(16,7)	5	7	time (0)	(8,0)	3	0-2	time (1)	(10,1)	3	0-2	time (2)	(11,2)	3	0-2	time (3)	(12,3)	4	3-4	time (4)	(13,4)	4	3-4	time (5)	(14,5)	5	5-7	time (6)	(15,6)	5	5-7	time (7)	(16,7)	5	5-7
Specified scale	Result (precision, scale)	Column length (bytes)	Fractional seconds precision																																						
time	(16,7)	5	7																																						
time (0)	(8,0)	3	0-2																																						
time (1)	(10,1)	3	0-2																																						
time (2)	(11,2)	3	0-2																																						
time (3)	(12,3)	4	3-4																																						
time (4)	(13,4)	4	3-4																																						
time (5)	(14,5)	5	5-7																																						
time (6)	(15,6)	5	5-7																																						
time (7)	(16,7)	5	5-7																																						
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]	<p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>An hour value of 24 is not valid. To represent midnight, use 12:00 AM or 00:00.</p> <p>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.</p>	
ISO 8601	Notes	
hh:mm:ss hh:mm[:ss][:fractional seconds]	<ul style="list-style-type: none">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 2012 as in SQL Server 2005.	

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.

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.

Backward Compatibility for Down-level Clients

Some down-level clients do not support the **time**, **date**, **datetime2** and **datetimeoffset** data types. The following table shows the type mapping between an up-level instance of SQL Server and down-level clients.

SQL Server 2012 data type	Default string literal format passed to down-level client	Down-level ODBC	Down-level OLEDB	Down-level JDBC	Down-level SQLCLIENT
time	hh:mm:ss[.nnnnnnn]	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
date	YYYY-MM-DD	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+ -]hh:mm	SQL_WVARCHAR or SQL_VARCHAR	DBTYPE_WSTRor DBTYPE_STR	Java.sql.String	String or SqString

Converting Date and Time Data

When you convert to date and time data types, SQL Server rejects all values it cannot recognize as dates or times. For information about using the CAST and CONVERT functions with date and time data, see [CAST and CONVERT \(Transact-SQL\)](#)

Converting time(n) Data Type to Other Date and Time Types

The following table describes what occurs when a **time** data type is converted to other date and time data types.

Data type to convert to	Conversion details
time(n)	<p>The hour, minute, and seconds are copied. When the destination precision is less than the source precision, the fractional seconds will be rounded up to fit the destination precision.</p> <p>The following example shows the results of converting a time(4) value to a time(3) value.</p> <div><pre>DECLARE @timeFrom time(4) = '12:34:54.1237'; DECLARE @timeTo time(3) = @timeFrom; SELECT @timeTo AS 'time(3)', @timeFrom AS 'time(4)';</pre></div>

	<pre>--Result --time(3) time(4) ----- --12: 34: 54. 124 12: 34: 54. 1237 -- --(1 row(s) affected)</pre>
date	The conversion fails, and error message 206 is raised: "Operand type clash: date is incompatible with time".
datetime	<p>The hour, minute, and second values are copied; and the date component is set to '1900-01-01'. When the fractional seconds precision of the time(n) value is greater than three digits, the datetime result will be truncated.</p> <p>The following code shows the results of converting a time(4) value to a datetime value.</p> <pre>DECLARE @time time(4) = '12: 15: 04. 1237'; DECLARE @datetime datetime= @time; SELECT @time AS '@time', @datetime AS '@datetime'; --Result --@time @datetime ----- --12: 15: 04. 1237 1900-01-01 12: 15: 04. 123 -- --(1 row(s) affected)</pre>
smalldatetime	<p>The date is set to '1900-01-01', and the hour and minute values are rounded up. The seconds and fractional seconds are set to 0.</p> <p>The following code shows the results of converting a time(4) value to a smalldatetime value.</p> <pre>-- Shows rounding up of the minute value. DECLARE @time time(4) = '12: 15: 59. 9999'; DECLARE @smalldatetime smalldatetime= @time; SELECT @time AS '@time', @smalldatetime AS '@smalldatetime'; --Result @time @smalldatetime ----- 12: 15: 59. 9999 1900-01-01 12: 16: 00-- --(1 row(s) affected) -- Shows rounding up of the hour value. DECLARE @time time(4) = '12: 59: 59. 9999'; DECLARE @smalldatetime smalldatetime= @time; SELECT @time AS '@time', @smalldatetime AS '@smalldatetime'; @time @smalldatetime ----- 12: 59: 59. 9999 1900-01-01 13: 00: 00 (1 row(s) affected)</pre>
datetimeoffset(n)	<p>The date is set to '1900-01-01', and the time is copied. The time zone offset is set to +00:00. When the fractional seconds precision of the time(n) value is greater than the precision of the datetimeoffset(n) value, the value is rounded up to fit.</p> <p>The following example shows the results of converting a time(4) value to a datetimeoffset(3) type.</p> <pre>DECLARE @time time(4) = '12: 15: 04. 1237'; DECLARE @datetimeoffset datetimeoffset(3) = @time; SELECT @time AS '@time', @datetimeoffset AS '@datetimeoffset'; --Result --@time @datetimeoffset ----- --12: 15: 04. 1237 1900-01-01 12: 15: 04. 124 +00: 00 -- --(1 row(s) affected)</pre>
datetime2(n)	<p>The date is set to '1900-01-01', the time component is copied, and the time zone offset is set to 00:00. When the fractional seconds precision of the datetime2(n) value is greater than the time(n) value, the value will be rounded up to fit.</p> <p>The following example shows the results of converting a time(4) value to a datetime2(2) value.</p> <pre>DECLARE @time time(4) = '12: 15: 04. 1237'; DECLARE @datetime2 datetime2(3) = @time; SELECT @datetime2 AS '@datetime2', @time AS '@time';</pre>

	--Result --@datetime2 @time ----- --1900-01-01 12: 15: 04. 124 12: 15: 04. 1237 -- --(1 row(s) affected)
--	---

Converting String Literals to time(n)

Conversions from string literals to date and time types are permitted if all parts of the strings are in valid formats. Otherwise, a runtime error is raised. Implicit conversions or explicit conversions that do not specify a style, from date and time types to string literals will be in the default format of the current session. The following table shows the rules for converting a string literal to the **time** data type.

Input string literal	Conversion Rule
ODBC DATE	ODBC string literals are mapped to the datetime data type. Any assignment operation from ODBC DATETIME literals into timetypes will cause an implicit conversion between datetime and this type as defined by the conversion rules.
ODBC TIME	See ODBC DATE rule above.
ODBC DATETIME	See ODBC DATE rule above.
DATE only	Default values are supplied.
TIME only	Trivial
TIMEZONE only	Default values are supplied.
DATE + TIME	The TIME part of the input string is used.
DATE + TIMEZONE	Not allowed.
TIME + TIMEZONE	The TIME part of the input string is used.
DATE + TIME + TIMEZONE	The TIME part of local DATETIME will be used.

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 +00:00	If the fractional seconds precision exceeds the value specified for the column, the string will be truncated without error.

See Also

Reference

[CAST and CONVERT \(Transact-SQL\)](#)

char and varchar (Transact-SQL)

SQL Server 2012

Are string data types of either fixed length or variable length.

char [(*n*)]

Fixed-length, non-Unicode string data. *n* defines the string length and 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 string data. *n* defines the string length and can be a value from 1 through 8,000. **max** indicates that the maximum storage size is 2³¹-1 bytes (2 GB). The storage size is the actual length of the data entered + 2 bytes. 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.

Converting Character Data

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. 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. See the Examples section that follows.

When a character expression is converted to a character expression of a different data type or size, such as from **char(5)** to **varchar(5)**, or **char(20)** to **char(15)**, the collation of the input value is assigned to the converted value. If a noncharacter expression is converted to a character data type, the default collation of the current database is assigned to the converted value. In either case, you can assign a specific collation by using the COLLATE clause.

Note

Code page translations are supported for **char** and **varchar** data types, but not for **text** data type. As with earlier versions of SQL Server, data loss during code page translations is not reported.

Character expressions that are being converted to an approximate **numeric** data type can include optional exponential notation (a lowercase e or uppercase E followed by an optional plus (+) or minus (-) sign and then a number).

Character expressions that are being converted to an exact **numeric** data type must consist of digits, a decimal point, and an optional plus (+) or minus (-). Leading blanks are ignored. Comma separators, such as the thousands separator in 123,456.00, are not allowed in the string.

Character expressions being converted to **money** or **smallmoney** data types can also include an optional decimal point and dollar sign (\$). Comma separators, as in \$123,456.00, are allowed.

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 = 'abc';
DECLARE @myNextVariable AS char = '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';
```

C. Converting Data for Display Purposes

The following example converts two columns to character types and applies a style that applies a specific format to the displayed data. A `money` type is converted to character data and style 1 is applied, which displays the values with commas every three digits to the left of the decimal point, and two digits to the right of the decimal point. A `datetime` type is converted to character data and style 3 is applied, which displays the data in the format dd/mm/yy. In the `WHERE` clause, a `money` type is cast to a character type to perform a string comparison operation.

```
USE AdventureWorks2012;
GO
SELECT BusinessEntityID,
       SalesYTD,
       CONVERT (varchar(12), SalesYTD, 1) AS MoneyDisplayStyle1,
       GETDATE() AS CurrentDate,
       CONVERT(varchar(12), GETDATE(), 3) AS DateDisplayStyle3
FROM Sales.SalesPerson
WHERE CAST(SalesYTD AS varchar(20)) LIKE '1%';
```

Here is the result set.

BusinessEntityID	SalesYTD	DisplayStyle1	CurrentDate	DisplayStyle3
278	1453719.4653	1,453,719.47	2011-05-07 14:29:01.193	07/05/11
280	1352577.1325	1,352,577.13	2011-05-07 14:29:01.193	07/05/11
283	1573012.9383	1,573,012.94	2011-05-07 14:29:01.193	07/05/11
284	1576562.1966	1,576,562.20	2011-05-07 14:29:01.193	07/05/11
285	172524.4512	172,524.45	2011-05-07 14:29:01.193	07/05/11
286	1421810.9242	1,421,810.92	2011-05-07 14:29:01.193	07/05/11
288	1827066.7118	1,827,066.71	2011-05-07 14:29:01.193	07/05/11

D. Converting Uniqueidentifier Data

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-E1C80E27ABAOwrong';
SELECT @ID, CONVERT(uniqueidentifier, @ID) AS TruncatedValue;
```

Here is the result set.

String	TruncatedValue
0E984725-C51C-4BF4-9960-E1C80E27ABAOwrong	0E984725-C51C-4BF4-9960-E1C80E27ABAO
(1 row(s) affected)	

See Also

- Reference
 - [nchar and nvarchar \(Transact-SQL\)](#)
 - [CAST and CONVERT \(Transact-SQL\)](#)
 - [COLLATE \(Transact-SQL\)](#)
 - [Data Types \(Transact-SQL\)](#)
- Concepts


[Data Type Conversion \(Database Engine\)](#)

[Other Resources](#)

[Estimate the Size of a Database](#)

ntext, text, and image (Transact-SQL)

SQL Server 2012

 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 string length of $2^{30} - 1$ (1,073,741,823) bytes. Storage size, in bytes, is two times the string length that is 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 string length of $2^{31} - 1$ (2,147,483,647). 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 (Transact-SQL)	READTEXT (Transact-SQL)
PATINDEX (Transact-SQL)	SET TEXTSIZE (Transact-SQL)
SUBSTRING (Transact-SQL)	UPDATETEXT (Transact-SQL)
TEXTPTR (Transact-SQL)	WRITETEXT (Transact-SQL)
TEXTVALID (Transact-SQL)	

See Also

- Reference
- [CAST and CONVERT \(Transact-SQL\)](#)
[Data Types \(Transact-SQL\)](#)
[LIKE \(Transact-SQL\)](#)
[SET @local_variable \(Transact-SQL\)](#)
- Concepts
- [Data Type Conversion \(Database Engine\)](#)
[Collation and Unicode Support](#)

nchar and nvarchar (Transact-SQL)

SQL Server 2012

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 string data. *n* defines the string length and must be a value from 1 through 4,000. The storage size is two times *n* bytes. When the collation code page uses double-byte characters, the storage size is still *n* bytes. Depending on the string, the storage size of *n* bytes can be less than the value specified for *n*. The ISO synonyms for **nchar** are **national char** and **national character**.

nvarchar [(*n* | **max**)]

Variable-length Unicode string data. *n* defines the string length and can be a value from 1 through 4,000. **max** indicates that the maximum storage size is $2^{31}-1$ bytes (2 GB). The storage size, in bytes, is two times the actual length of data entered + 2 bytes. 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.

Converting Character Data

For information about converting character data, see [char and varchar \(Transact-SQL\)](#).

See Also

Reference

[ALTER TABLE \(Transact-SQL\)](#)
[CAST and CONVERT \(Transact-SQL\)](#)
[COLLATE \(Transact-SQL\)](#)
[CREATE TABLE \(Transact-SQL\)](#)
[Data Types \(Transact-SQL\)](#)
[DECLARE @local_variable \(Transact-SQL\)](#)
[LIKE \(Transact-SQL\)](#)
[SET ANSI_PADDING \(Transact-SQL\)](#)
[SET @local_variable \(Transact-SQL\)](#)

Concepts

[Collation and Unicode Support](#)

binary and varbinary (Transact-SQL)

SQL Server 2012

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.

Converting binary and varbinary Data

When data is converted from a string data type (**char**, **varchar**, **nchar**, **nvarchar**, **binary**, **varbinary**, **text**, **ntext**, or **image**) to a **binary** or **varbinary** data type of unequal length, SQL Server pads or truncates the data on the right. When other data types are converted to **binary** or **varbinary**, the data is padded or truncated on the left. Padding is achieved by using hexadecimal zeros.

Converting data to the **binary** and **varbinary** data types is useful if **binary** data is the easiest way to move data around. Converting any value of any type to a binary value of large enough size and then back to the type, will always result in the same value if both conversions are taking place on the same version of SQL Server. The binary representation of a value might change from version to version of SQL Server.

You can convert **int**, **smallint**, and **tinyint** to **binary** or **varbinary**, but if you convert the **binary** value back to an integer value, this value will be different from the original integer value if truncation has occurred. For example, the following SELECT statement shows that the integer value **123456** is usually stored as a binary **0x0001e240**:

```
SELECT CAST( 123456 AS BINARY(4) );
```

However, the following **SELECT** statement shows that if the **binary** target is too small to hold the entire value, the leading digits are silently truncated so that the same number is stored as **0xe240**:

```
SELECT CAST( 123456 AS BINARY(2) );
```

The following batch shows that this silent truncation can affect arithmetic operations without raising an error:

```
DECLARE @BinaryVar1 AS BINARY(2);

SET @BinaryVar1 = 123456;
SET @BinaryVar1 = @BinaryVar1 + 1;

SELECT CAST( @BinaryVar1 AS INT);
GO
```

The final result is **57921**, not **123457**.



Note

Conversions between any data type and the **binary** data types are not guaranteed to be the same between versions of SQL Server.

See Also

Reference

[CAST and CONVERT \(Transact-SQL\)](#)

[Data Types \(Transact-SQL\)](#)

Concepts

[Data Type Conversion \(Database Engine\)](#)

cursor (Transact-SQL)

SQL Server 2012

A data type for variables or stored procedure OUTPUT parameters that contain a reference to a cursor. Any variables created with the **cursor** data 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 **cursor** data 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.

See Also

Reference

[CAST and CONVERT \(Transact-SQL\)](#)

[CURSOR_STATUS \(Transact-SQL\)](#)

[Data Types \(Transact-SQL\)](#)

[DECLARE CURSOR \(Transact-SQL\)](#)

[DECLARE @local_variable \(Transact-SQL\)](#)

[SET @local_variable \(Transact-SQL\)](#)

Concepts

[Data Type Conversion \(Database Engine\)](#)

hierarchyid (Transact-SQL)

SQL Server 2012

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 $6 \cdot \log_A n$ 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 [Hierarchical Data \(SQL Server\)](#).

- 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 2012, 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 **hierarchyid** columns work with one directional replication depends on the version of SQL Server the Subscriber is running.

- A SQL Server 2012 Publisher can replicate **hierarchyid** columns to a SQL Server 2012 Subscriber without any special considerations.
- A SQL Server 2012 Publisher must convert **hierarchyid** columns to replicate them to a Subscriber that is running SQL Server Compact or an earlier version of SQL Server. SQL Server Compact 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.

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 2012.
- 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.

■ See Also

Concepts

[Hierarchical Data \(SQL Server\)](#)

[hierarchyid Data Type Method Reference](#)

sql_variant (Transact-SQL)

SQL Server 2012

A data type that stores values of various SQL Server-supported data types.

 Transact-SQL Syntax Conventions

Syntax

sql _vari ant

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	rowversion (timestamp)
sql_variant	geography
hierarchyid	geometry
User-defined types	datetimeoffset

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.

Converting sql_variant Data

When handling the **sql_variant** data type, SQL Server supports implicit conversions of objects with other data types to the **sql_variant** type. However, SQL Server does not support implicit conversions from **sql_variant** data to an object with another data type.

See Also

Reference

[CAST and CONVERT \(Transact-SQL\)](#)

[SQL_VARIANT_PROPERTY \(Transact-SQL\)](#)

table (Transact-SQL)

SQL Server 2012

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. Functions and variables can be declared to be of type **table**. **table** variables can be used in functions, stored procedures, and batches. To declare variables of type **table**, use [DECLARE @local_variable](#).

 [Transact-SQL Syntax Conventions](#)

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 \(Transact-SQL\)](#), [CREATE FUNCTION \(Transact-SQL\)](#), and [DECLARE @local_variable \(Transact-SQL\)](#).

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.

General Remarks

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);
```

table variables provide the following benefits for small-scale queries that have query plans that do not change and when recompilation concerns are dominant:

- 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.

- **table** variables used in stored procedures cause fewer recompilations of the stored procedures than when temporary tables are used when there are no cost-based choices that affect performance.
- 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.

Limitations and Restrictions

Table variables does not have distribution statistics, they will not trigger recompiles. Therefore, in many cases, the optimizer will build a query plan on the assumption that the table variable has no rows. For this reason, you should be cautious about using a table variable if you expect a larger number of rows (greater than 100). Temporary tables may be a better solution in this case. Alternatively, for queries that join the table variable with other tables, use the RECOMPILE hint, which will cause the optimizer to use the correct cardinality for the table variable.

table variables are not supported in the SQL Server optimizer's cost-based reasoning model. Therefore, they should not be used when cost-based choices are required to achieve an efficient query plan. Temporary tables are preferred when cost-based choices are required. This typically includes queries with joins, parallelism decisions, and index selection choices.

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 \(Transact-SQL\)](#). Queries that read **table** variables without modifying them can still be parallelized.

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 \(Transact-SQL\)](#).

CHECK constraints, DEFAULT values and computed columns in the **table** type declaration cannot call user-defined functions.

Assignment operation between **table** variables is not supported.

Because **table** variables have limited scope and are not part of the persistent database, they are not affected by transaction rollbacks.

Table variables cannot be altered after creation.

Examples

A. Declaring a variable of type table

The following example creates a **table** variable that stores the values specified in the OUTPUT clause of the UPDATE statement. Two **SELECT** statements follow that return the values in **@MyTableVar** and the results of the update operation in the **Employee** table. Note that the results in the **INSERTED.ModifiedDate** column differ from the values in the **ModifiedDate** column in the **Employee** table. This is because the **AFTER UPDATE** trigger, which updates the value of **ModifiedDate** to the current date, is defined on the **Employee** table. However, the columns returned from **OUTPUT** reflect the data before triggers are fired. For more information, see [OUTPUT Clause \(Transact-SQL\)](#).

Transact-SQL

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

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.

Transact-SQL

```
USE AdventureWorks2012;
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.

Transact-SQL

```
SELECT * FROM Sales.ufn_SalesByStore (602);
```

See Also

Reference

[COLLATE \(Transact-SQL\)](#)
[CREATE FUNCTION \(Transact-SQL\)](#)
[CREATE TABLE \(Transact-SQL\)](#)
[DECLARE @local_variable \(Transact-SQL\)](#)
[Query Hints \(Transact-SQL\)](#)

Concepts

[User-Defined Functions](#)
[Use Table-Valued Parameters \(Database Engine\)](#)

rowversion (Transact-SQL)

SQL Server 2012

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 \(Transact-SQL\)](#).

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 -- Severi ty.
        ,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 -- Severi ty.
        ,1 -- State
        ,1) -- myKey that was changed
END;

```

See Also

Reference

[ALTER TABLE \(Transact-SQL\)](#)
[CAST and CONVERT \(Transact-SQL\)](#)
[CREATE TABLE \(Transact-SQL\)](#)
[Data Types \(Transact-SQL\)](#)
[DECLARE @local_variable \(Transact-SQL\)](#)
[DELETE \(Transact-SQL\)](#)
[INSERT \(Transact-SQL\)](#)
[MIN_ACTIVE_ROWVERSION \(Transact-SQL\)](#)
[SET @local_variable \(Transact-SQL\)](#)
[UPDATE \(Transact-SQL\)](#)

uniqueidentifier (Transact-SQL)

SQL Server 2012

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)	

See Also

Reference

[ALTER TABLE \(Transact-SQL\)](#)
[CAST and CONVERT \(Transact-SQL\)](#)
[CREATE TABLE \(Transact-SQL\)](#)
[Data Types \(Transact-SQL\)](#)
[DECLARE @local_variable \(Transact-SQL\)](#)
[NEWID \(Transact-SQL\)](#)
[SET @local_variable \(Transact-SQL\)](#)

Concepts

[Updatable Subscriptions for Transactional Replication](#)

xml (Transact-SQL)

SQL Server 2012

Is the data type that stores XML data. You can store **xml** instances in a column, or a variable of **xml** type.

 [Transact-SQL Syntax Conventions](#)

Syntax

```
xml ( [ CONTENT | DOCUMENT ] xml_schema_col l e c t i o n )
```

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 [Compare Typed XML to Untyped XML](#).

Remarks

The stored representation of **xml** data type instances cannot exceed 2 gigabytes (GB) in size.

The CONTENT and DOCUMENT facets apply only to typed XML. For more information see [Compare Typed XML to Untyped XML](#).

Examples

```
USE AdventureWorks;
GO
DECLARE @y xml (Sales.IndividualSurveySchemaCollection)
SET @y = (SELECT TOP 1 Demographics FROM Sales.Individual);
SELECT @y;
GO
```

See Also

Reference

[Data Types \(Transact-SQL\)](#)

Concepts

[Data Type Conversion \(Database Engine\)](#)

Other Resources

[xml Data Type Methods](#)

[XQuery Language Reference \(SQL Server\)](#)

