**Data types**

Objects that contain data have an associated data type that defines the kind of data,

Exact Numeric Data Types

The exact numeric data types are the most common [SQL Server data types](http://databases.about.com/od/sqlserver/a/mssql_datatypes.htm) used to store numeric information.

Data types in the exact numeric category include:

**Integer data type**

The integer data types are the only ones that can be used with the IDENTITY property, which is an automatically incrementing number. The IDENTITY property is typically used to automatically generate unique identification numbers or primary keys.

**Int**

Variables store 4-byte whole numbers ranging from -2,147,483,648 to 2,147,483,647. Integers are whole numbers and do not contain decimals or fractions. Integer objects and expressions can be used with any mathematical operations. Any fractions generated by these operations are truncated, not rounded.

**Bigint:**

Has a length of 8 bytes and stores numbers from –2^63 (-9,223,372,036,854,775,808) through 2^63-1 (9,223,372,036,854,775,807).

**smallint**  
Has a length of 2 bytes, and stores numbers from -32,768 through 32,767.

**tinyint** variables store 1-byte whole numbers ranging from 0 to 255.

**decimal** :

The **decimal** data type can store a maximum of 38 digits, all of which can be to the right of the decimal point. The **decimal** data type stores an exact representation of the number; there is no approximation of the stored value.

The two attributes that define **decimal** columns, variables, and parameters are:

* *p*   
  Specifies the precision, or the number of digits the object can hold.
* *s*   
  Specifies the scale or number of digits that can be placed to the right of the decimal point.   
  *p* and *s* must observe the rule: 0 <= *s* <= *p* <= 38.

**decimal** and **numeric** variables are functionally equivalent and store numbers of fixed precision and scale. Precision indicates the maximum number of digits that may be stored (including those before and after the decimal point. Scale indicates the number that may be stored to the right of the decimal point.

**Monetary Data**

**money** variables store 8-byte currency values ranging from -922,337,203,685,477.5808 to 922,337,203,685,477.5807. They may reflect any currency type. Microsoft SQL Server 2005 stores monetary data, or currency values, using two data types: **money** and **smallmoney**. Currency or monetary data does not need to be enclosed in single quotation marks ( ' ).

**smallmoney** : variables store 4-byte currency values ranging from -214,748.3648 to 214,748.3647. They may reflect any currency type. **money** and **smallmoney** are limited to four decimal points.

**The approximate numeric data types:**

**float** variables store 4-byte or 8-byte floating point numbers. They are specified as float(p), where p is 24 for a 4-byte number and 53 for an 8-byte number.

**real** variables store 4-byte floating point numbers and are functionally equivalent to float(24) variables.

The **float** and **real** data types are known as approximate data types. Approximate numeric data types do not store the exact values specified for many numbers; they store an extremely close approximation of the value.

Avoid using **float** or **real** columns in WHERE clause search conditions, especially the = and <> operators. It is best to limit **float** and **real** columns to > or < comparisons.

**Date and Time Data**

These new data types allow designers to easily work with time zones, dates without times (and vice versa) and dates in ancient history and far into the future.

**datetime** variables store 8-byte time and date values ranging from January 1, 1753 to December 31, 9999 with an accuracy of 3.33 milliseconds. Enter new dates and times or change existing dates and times.

SQL Server recognizes date and time data that is enclosed in single quotation marks (') in the following formats:

* Alphabetic date such as 'April 15, 1998'
* Numeric date formats, such as '4/15/1998'
* Unseparated string formats. For example, '19981207' would be December 7, 1998.

**smalldatetime** variables store 4-byte time and date values ranging from January 1, 1900 to June 6, 2079 with an accuracy of 1 minute.

**timestamp** variables are automatically populated by SQL Server with the time that a row is created or modified. The timestamp value is based upon an internal clock and does not correspond to real time. Each table may have only one timestamp variable.

**Character String Data Types**

Character string [data types](http://databases.about.com/od/sqlserver/a/mssql_datatypes.htm) are used to store text values in Microsoft SQL Server databases.

**char(n)** variables store fixed-length character strings conisisting of exactly n characters (and, therefore, n bytes). They are limited to 8,000 characters in size.

**nchar(n)** variables store fixed-length Unicode character strings conisisting of exactly n characters (and, therefore, 2\*n bytes). They are limited to 4,000 characters in size.

**varchar(n)** variables store non-fixed length character strings consisting of approximately n characters. They consume l+2 bytes of space, where l is the actual length of the string. They are limited to 8,000 characters in size.

**nvarchar(n)** variables store non-fixed length Unicode character strings consisting of approximately n characters. They consume 2\*l+2 bytes of space, where l is the actual length of the string. They are limited to 4,000 characters in size.

**varchar(max)** variables store non-fixed length character strings consisting of up to 1,073,741,824 characters. They consume l+2 bytes of space, where l is the actual length of the string.

**nvarchar(max)** variables store non-fixed length Unicode character strings consisting of up to 536,870,912 characters. They consume l\*2+2 bytes of space, where l is the actual length of the string.

**text** and **ntext** variables store up to 2GB of text data (ANSI and Unicode, respectively), but can not be used in many text operations. Therefore, they are usually only used to support legacy applications and have been replaced by the varchar(max) and nvarchar(max) data types.

**Binary Data Types**

**binary** data can store a maximum of 8,000 bytes. **varbinary**, using the max specifier, can store a maximum of 2^31 bytes. The **binary** and **varbinary** data types store strings of bits

Data types in the binary category include:

**bit** variables store a single bit with a value of 0, 1 or [NULL](http://databases.about.com/cs/sql/a/aa042803a.htm).

**binary(n)** variables store n bytes of fixed-size binary data. They may store a maximum of 8,000 bytes.

**varbinary(n)** variables store variable-length binary data of approximately n bytes. They may store a maximum of 8,000 bytes.

**varbinary(max)** variables store variable-length binary data of approximately n bytes. They may store a maximum of 2 gigabytes.

**image** variables store up to 2 gigabytes of data and are commonly used to store any type of data file (not just images).

If the binary data types don’t suit your needs, read more about [other SQL Server data types](http://databases.about.com/od/sqlserver/a/mssql_datatypes.htm).

**Other data type**

**cursor** variables store references to cursors used for database operations. A table may not contain a cursor variable.

**sql\_variant** variables store up to 8,000 bytes of data from any SQL Server data type other than varchar(max), nvarchar(max), text, image, sql\_variant, varbinary(max), xml, ntext and timestamp.

**table** variables store temporary tables used during database operations. SQL Server database tables may not contain variables of type table.

**xml** variables store XML formatted data. They may store a maximum of 2 gigabytes.

**uniqueidentifier** variables store 16-bit globally unique identifiers. They may be instantiated with a new GUID using the NEWID function.

The **uniqueidentifier** data type stores 16-byte binary values that operate as globally unique identifiers (GUIDs). A GUID is a unique binary number; no other computer in the world will generate a duplicate of that GUID value.

A GUID value for a **uniqueidentifier** column is usually obtained by one of the following ways:

* In a Transact-SQL statement, batch, or script by calling the NEWID function.
* In application code by calling an application API function or method that returns a GUID.

The Transact-SQL NEWID function and the application API functions and methods generate new **uniqueidentifier** values from the identification number of their network card plus a unique number from the CPU clock. Each network card has a unique identification number. The **uniqueidentifier** value that is returned by NEWID is generated by using the network card on the server. The **uniqueidentifier** value returned by application API functions and methods is generated by using the network card on the client.

A **uniqueidentifier** value is not typically defined as a constant. You can specify a **uniqueidentifier** constant in the following ways:

* Character string format: '6F9619FF-8B86-D011-B42D-00C04FC964FF'
* Binary format: 0xff19966f868b11d0b42d00c04fc964ff

The **uniqueidentifier** data type does not automatically generate new IDs for inserted rows in the way the IDENTITY property does. For example, to obtain new **uniqueidentifier** values, a table must have a DEFAULT clause specifying the NEWID or NEWSEQUENTIALID function, or INSERT statements must use the NEWID function.

**uniqueidentifier** columns may contain multiple occurrences of an individual **uniqueidentifier** value, unless the UNIQUE or PRIMARY KEY constraints are also specified for the column. A foreign key column that references a **uniqueidentifier** primary key in another table will have multiple occurrences of individual **uniqueidentifier** values when multiple rows reference the same primary key in the source table.

The **uniqueidentifier** data type has the following disadvantages:

* The values are long and obscure. This makes them difficult for users to type correctly, and more difficult for users to remember.
* The values are random and cannot accept any patterns that may make them more meaningful to users.
* There is no way to determine the sequence in which **uniqueidentifier** values were generated. They are not suited for existing applications that depend on incrementing key values serially.
* At 16 bytes, the **uniqueidentifier** data type is relatively larger than other data types, such as 4-byte integers. This means indexes that are built using **uniqueidentifier** keys might be relatively slower than indexes using an **int** key.