D

Data Type Reference

Choosing proper data types to match the domain chosen during logical modeling is a very important task. One data type might be more efficient than another of a similar type. For example, storing integer data can be done in an integer data type, a numeric data type, or even a floating point data type, but they are certainly not alike in implementation or performance.

In this appendix, I'll introduce you to all of the intrinsic data types that are provided by Microsoft and discuss the situations where they are best used.

Precise Numeric Data

There are many base data types in which you can store numerical data. Here you'll learn to deal with two different types of numeric data: **precise** and **approximate**. The differences are very important and must be well understood by any architect who is building a system that stores readings, measurements, or other numeric data.

The precise numeric values include the `bit`, `int`, `bigint`, `smallint`, `tinyint`, `decimal`, and `money` data types (`money` and `smallmoney`). Precise values have no error in the way they are stored, from integer to floating point values, because they have a fixed number of digits before and after the decimal point (or radix). However, when you must store decimal values in precise data types, you'll pay a performance and storage cost in the way they are stored and dealt with.

bit

The `bit` has values of 0, 1, or `NULL`.

It's generally used as a Boolean. It isn't the ideal Boolean, but because SQL Server doesn't have a distinct Boolean data type, it's the best that you currently have. `bit` columns can't be indexed–having only two values (technically three with `NULL`) makes for a poor index–so they shouldn't be used in heavily searched combinations of fields. You should avoid having a `bit` field set to `NULL` if possible; including `NULL`s in a Boolean comparison increases the complexity of queries and will cause problems.

The `bit` column requires 1 byte of storage per eight instances in a table. Hence, having eight `bit` columns will cause your table to be no larger than if your table had only a single `bit` column.

int

The `int` indicates whole numbers from –2,147,483,648 to 2,147,483,647 (that is, -2^{31} to $2^{31} - 1$).

The integer data type is used to store signed (+ or –) whole number data. The integer data type is frequently employed as a primary key for tables, as it's very small (it requires 4 bytes of storage) and very efficient to store and retrieve.

The only real downfall of the `int` data type is that it doesn't include an unsigned version, which could store nonnegative values from 0 to 4,294,967,296 (or 2^{32}). As most primary key values start out at 1, this would give you over two billion extra values for a primary key value. This may seem unnecessary, but systems that have billions of rows are becoming more and more common.

Another application where the typing of the `int` field plays an important part is the storage of IP addresses as integers. An IP address is simply a 32-bit integer broken down into four octets. For example, if you had an IP address of 234.23.45.123, you would take $(234 * 2^3) + (23 * 2^2) + (45 * 2^1) + (123 * 2^0)$. This value will fit nicely into an unsigned 32-bit integer, but not into a signed one. There is, however, a 64-bit integer in SQL Server that covers the current IP address standard nicely, but requires twice as much storage.

bigint

The `bigint` specifies whole numbers from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (that is, -2^{63} to $2^{63} - 1$).

One of the main reasons to use the 64-bit data type is as a primary key for tables where you'll have more than two billion rows, or if the situation directly dictates it like the IP address situation I previously discussed.

smallint

The `smallint` indicates whole numbers from –32,768 through to 32,767 (or -2^{15} through $2^{15} - 1$).

If you can be guaranteed of needing only values in this range, the `smallint` can be a useful type. It requires 2 bytes of storage.

Using the `smallint` can be a tradeoff. On the one hand, it does save 2 bytes over the `int` data type, but frequently the `smallint` isn't large enough to handle all needs. If the data can fit into the constraints of this domain, and your table will be very large, it may be worth using the `smallint` to save 2 bytes per row. For small to moderate databases, the possibility of running out of space and having to change all of the code to use a different data type can be too much to overcome. You'll generally use the `int` data type in your databases.

It's also generally a bad idea to use a `smallint` for a primary key (except in extreme situations, such as when you need to migrate a foreign key to a table with trillions of rows or where a byte or two will make a performance difference). Uniformity makes your database code more consistent. This may seem like a small point, but in most average systems it's much easier to code when you automatically know what the data type is.

One use of a `smallint` that crops up from time to time is as a Boolean. I guess this is because in Visual Basic, 0 equals `False` and –1 equals `True` (technically, VB will treat any nonzero value as `True`). Storing data in this manner is not only a tremendous waste of space (2 bytes versus potentially 1/8th of a byte), but also confusing to all other SQL Server programmers. ODBC and OLE DB drivers do this translation for you, but even if they didn't, it's worth the time to write a method or a function in VB to translate `True` to a value of 1.

tinyint

The `tinyint` specifies nonnegative whole numbers from 0 through 255.

The same comments for the `smallint` can be made for the `tinyint`. The `tinyint` types are very small, using a single byte for storage, but you'll rarely use them because there are very few situations where you can guarantee that a value will never exceed 255 or employ negative numbers.

decimal (or numeric)

These data types cover all numeric data between $-10^{38} - 1$ through $10^{38} - 1$.

The `decimal` data type is a precise data type because it's stored in a manner that's like character data (as if the data only had 12 characters, 0 to 9 and the minus and decimal point symbols). The way it's stored prevents the kind of imprecision you'll see with the `float` and `real` data types a bit later. It does, however, incur an additional cost in getting and manipulating values.

To specify a `decimal` number, you need to define the **precision** and the **scale**:

- ❑ Precision is the total number of significant digits in the number. For example, the number 10 would need a precision of 2, and 43.00000004 would need a precision of 10. The precision may be as small as 1 or as large as 38.
- ❑ Scale is the possible number of significant digits to the right of the decimal point. Reusing the previous example, 10 would have a scale of 0, and 43.00000004 would need 8.

Numeric data types are bound by this precision and scale to define how large the data is. For example, take the following declaration of a numeric variable:

```
DECLARE @testvar decimal(3,1)
```

This will allow you to enter any numeric values greater than –99.94 and less than 99.94. Entering 99.949999 works but entering 99.95 doesn't, because it will be rounded up to 100.0, which can't be displayed by `decimal(3,1)`. The following, for example:

```
SELECT @testvar = -10.155555555
SELECT @testvar
```

returns –10.2.

When a computer programming language does this for you, it's considered an implicit conversion. This is both a blessing and a curse. The point here is that you must be really careful when butting up to the edge of the data type allowable values. Note that there is a setting—`SET NUMERIC_ROUNDABORT ON`—that will cause an error to be generated when a loss of precision would occur from an implicit data conversion. This can be quite dangerous to use and may throw off applications using SQL Server if set to `ON`. However, if you need to prevent implicit round-off due to system constraints, it's a very valuable tool.

Storage for the numeric types depends on how much precision is required:

Precision	Bytes Required
1–9	5
10–19	9
20–28	13
29–38	17

As far as usage is concerned, the `decimal` data type should generally be used sparingly. There is nothing wrong with the type at all, but it does take that little bit more processing than integers or real data, and hence there is a performance hit. You should use it when you have specific values that you wish to store where you can't accept any loss of precision. Again, the topic of loss of precision will be dealt with in more detail in the "Approximate Numeric Data" section.

Money Values

The `money` data types are basically integers with a decimal point inserted in the number. Because it always has the same precision and scale, the processor can deal with it exactly like a typical integer. The decimal can be shifted once the mathematical functions have been processed. There are two flavors of `money` variables:

- `money`: Values from –922,337,203,685,477.5808 through 922,337,203,685,477.5807, with an accuracy of one ten-thousandth of a monetary unit. As would be expected, this is obviously a 64-bit integer and it requires 8 bytes of storage.
- `smallmoney`: Values from –214,748.3648 through 214,748.3647, with an accuracy of one ten-thousandth of a monetary unit. It requires 4 bytes of storage.

The `money` and `smallmoney` data types are excellent storage for almost any requirement for storing monetary values, unless you have requirements to store your monetary values to greater than four decimal places, which isn't usually the case in most situations.

Approximate Numeric Data

Approximate numeric values contain a decimal point and are stored in a format that is fast to manipulate, but they are only accurate to the 15th decimal place. Approximate numeric values have some very important advantages, as you'll see later in the appendix.

Approximate is such a negative term, but it's technically the proper term. It refers to the `real` and `float` data types, which are IEEE 75454 standard single- and double-precision floating point values. The reason they are called floating point values is based on the way that they are stored. Basically, the number is stored as a 32-bit or 64-bit value, with four parts:

- **Sign**: Determines if this is a positive or negative value
- **Exponent**: The exponent in base 2 of the mantissa
- **Mantissa**: Stores the actual number that is multiplied by the exponent
- **Bias**: Determines whether or not the exponent is positive or negative

A complete description of how these data types are actually formed is beyond the scope of this book, but may be obtained from the IEEE body at www.ieee.org for a nominal charge.

There are two different flavors of floating point data types available in SQL Server:

- `float [ (N) ]`: Values in the range from –1.79E + 308 through to 1.79E + 308. The `float` data type allows you to specify a certain number of bits to use in the mantissa, from 1 to 53. You specify this number of bits with the value in `N`.
- `real`: Values in the range from –3.40E + 38 through to 3.40E + 38. `real` is a synonym for a `float(24)` data type.

The storage and precision for the float data types are as follows:

N (Number of Mantissa Bits for Float)	Precision	Storage Size
1–24	7	4 bytes
25–53	15	8 bytes

Hence, you are able to represent most values from –1.79E + 308 to 1.79E + 308 with a precision of 15 significant digits. This isn't as many as the numeric data types can deal with, but is plenty for almost any scientific application.

Note that in the previous paragraph I said "represent most values." This is an issue that you must understand. There are some values that simply can't be stored in floating point format. A classic example of this is the value 1/10. There is no way to store this value exactly. When you execute the following snippet of code:

```
SELECT CAST(1 AS float(53))/CAST( 10 AS float(53)) AS funny_result
```

you expect that the result will be exactly 0.1. However, executing the statement returns

```
funny_result
---------------------------------
0.10000000000000001
```

Even if you change the insert to

```
SELECT CAST(.1 AS float) AS funny_result
```

you get the same bad result. The difference is interesting, but insignificant for all but the most needy calculations. The float and real data types are intrinsic data types that can be dealt with using the floating point unit. In all processors today, this unit is part of the main CPU, so this is slightly more costly than integer math that can be done directly in the CPU registers, but tremendously better than the noninteger exact data types. The flaw is based on the way floating point data is stored. Any further discussion of floating point storage flaws is beyond the scope of this book.

The approximate numeric data types are very important for storing measurements where calculations are required. The round-off that occurs is insignificant because there are still very few devices that are capable of taking measurements with 15 significant digits and, as you'll likely be doing large quantities of mathematics on the measurements, the performance will be significantly better over using the decimal type.

When coding with floating point values, it's generally best to avoid trying to compare these values using the = operator for comparisons. As values are approximate, it isn't always possible to type in the exact value as it will be stored. If you need to be able to find exact values, then it's best to use the numeric data types.

Date and Time Data

If you recall, this isn't the first time I've discussed the `datetime` and `smalldatetime` data types. The first time you encountered them was when I was discussing violations of the First Normal Form. In this section, I'll present the facts about the `datetime` data types, then I'll show you some of the alternatives.

smalldatetime

This represents date and time data from January 1, 1900, through to June 6, 2079.

The `smalldatetime` data type is accurate to one minute. It requires 4 bytes of storage. `smalldatetime`s are the best choice when you need to store the date and possibly time of some event where accuracy of a minute isn't a problem. If more precision is required, then the `datetime` data type is probably a better choice.

datetime

This represents date and time data from January 1, 1753, to December 31, 9999.

Storage for the `smalldatetime` and `datetime` data types is interesting. The `datetime` is stored as two 4-byte integers. The first integer stores the number of days before *or* after January 1, 1990. The second stores the number of milliseconds after midnight of the day. `smalldatetime` values are really close to this, but use 2-byte integers instead of 4-byte integers. The first is an unsigned 2-byte integer and the second is used to store the number of minutes past midnight.

The `datetime` data type is accurate to 0.03 seconds. Using 8 bytes, it does however require a sizable chunk of memory. Use `datetime` when you either need the extended range, or when you need the precision. There are few cases where you need this kind of precision. The only application that comes to mind is a timestamp column (not to be confused with the `timestamp` data type, which I'll discuss in a later section), used to denote *exactly* when an operation takes place. This isn't uncommon if you need to get timing information such as the time taken between two activities in seconds, or if you want to use the `datetime` value in a concurrency control mechanism. I'll delve deeper into the topic of concurrency control later in this appendix.

Using User-Defined Data Types to Manipulate Dates and Times

A word about storing date and time values. Employing the `datetime` data types when you only need the date part, or just the time, is a pain. Take the following code sample. You want to find every record where the date is equal to July 12, 2001. If you code this in the very obvious manner:

```
SELECT *
FROM employee
WHERE birthDate = 'July 12,1967'
```

you'll get a match for every employee where birthDate exactly matches "July 12, 1967 00:00:00.000". However, if the date is stored as "July 12, 1967 10:05:23.300"–as it might be by design (that is, she was born at that exact time), or by error (that is, a date control may send the current time by default if you forget to clear it)–you can get stuck with having to write queries like the following to answer the question of who was born on July 12.

To do this, you would need to rewrite your query like so:

```
SELECT *
FROM employee
WHERE birthDate >= 'July 12,1967 0:00:00.000'
AND birthDate < 'July 13,1967 0:00:00.000'
```

Note that you don't use BETWEEN *for this operation. If you do, the* WHERE *clause would have to state*

```
WHERE birthDate BETWEEN 'July 12, 1967 0:00:00.00'
AND 'July 12, 1967 23:59:59.997'
```

first to exclude any July 13 dates, then to avoid any round-off with the datetime *data type. As it's accurate to 0.03 of a second, when it evaluates "July 12, 1967 23:59:59.997", it rounds it off to "July 13, 1967 0:00:00.00".*

This is not only troublesome and cumbersome, but also can be inefficient. In the case where you don't need the date or the time parts of the datetime data types, it may best suit you to create your own. For example, let's take a very typical need for storing the time when something occurs, without the date. There are two possibilities for dealing with this situation. Both are much better than simply using the datetime data type, but each has its own set of inherent problems.

- Use multiple columns: one each for hour, minute, second, and whatever fractions of a second you may need. The problem here is that this isn't easy to query. You need at least three columns to query for a given time, and you'll need to index all of the columns involved in the query to get good performance on large tables. Storage will be better, as you can fit all of the different parts into tinyint columns. If you want to look at what has happened during the tenth hour of the day over a week, then you could use this method.
- Use a single column that holds the number of seconds (or parts of seconds between midnight and the current time). This, however, suffers from an inability to do a reasonable query for all values that occur during, say, the tenth hour, and, if you specified the number of seconds, this would be every value between 36000 and 39600. However, this is ideal if you are using these dates in some sort of internal procedure that human users wouldn't need to interact with.

Using these columns, you could create user-defined functions to convert your data types to readable values.

As another example, you could create your own date type that is simply implemented as an integer. You could add a user-defined type named `intDateType` (more coverage will be allocated to this subject later in the appendix) as follows:

```
EXEC sp_addtype @typename = intDateType,
   @phystype = integer,
   @NULLtype = 'NULL',
   @owner = 'dbo'
GO
```

and then a user-defined function to convert your data values in the variable into a real `datetime` variable, to use in date calculations or simply to show to a client.

```
CREATE FUNCTION intDateType$convertToDatetime
(
   @dateTime   intDateType
)
RETURNS datetime
AS
BEGIN
   RETURN ( dateadd(day,@datetime,'jan 1, 2001'))
END
GO
```

From this basic example, it should be pretty obvious that the user-defined function plays an extremely important part in coding your SQL Server applications.

To test your new function, you could start with looking at the second day of the year, by converting date one to a `datetime` variable as follows:

```
SELECT dbo.intDateType$convertToDatetime(1) as convertedValue
```

which returns

```
convertedValue
----------------------------------
2001-01-02 00:00:00.000
```

You would have to build several other data entry functions to take in a `datetime` variable and store it as a new data type. Though I've deliberately not gone into much detail with user-defined functions, I've suggested a range of possibilities to solve a date or time problem.

The `datetime` data type presents programmatic problems that you must deal with. Just be careful out there.

Binary Data

Binary data allows you to store a string of bits, with a range of 1 byte (8 bits) to around 2 gigabytes of bits (which is $2^{31} - 1$ or 2,147,483,647 bytes).

A concept that I'll begin to show you with binary data types is the use of **variable-length** data. Basically this means that the size of the data store in each instance of the column or variable of the data type can change, based on how much data is stored. In all previous data types the storage was fixed, because what was stored was a fixed-length representation of the value that was interpreted using some formula (for example, an integer is 32 bits, or a date is the number of time units since some point in time). Any data type that allows variable-length values is generally stacking values together, like letters in a word. Variable-length data generally comprises two pieces–a value explaining how long the data is, and the actual data itself. There is some overhead with storing variable-length data, but it's usually better than storing data in fixed-length columns with considerable wasted space. Smaller data means more records per page, which reduces I/O, and I/O operations are far more expensive than locating fields in a record.

Binary data is stored in three different flavors:

- `binary` is fixed-length data with a maximum length of 8,000 bytes. Only use the `binary` type if you are certain how large your data will be. Storage will be equivalent to the number of bytes that you declare the variable for.
- `varbinary` is for variable-length binary data with a maximum length of 8,000 bytes.
- `image` is for really large variable-length image data with a maximum length of $2^{31} - 1$ (2,147,483,647) bytes.

One of the restrictions of binary data types is that they don't support bitwise operators, which would allow you to do some very powerful bitmask storage by being able to compare two binary columns to see not only if they differ, but how they differ. The whole idea of the `binary` data types is that they store strings of bits. The bitwise operators can operate on integers, which are physically stored as bits. The reason for this inconsistency is actually fairly clear from the point of view of the internal query processor. The bitwise operations are actually operations that are handled in the processor, whereas the binary data types are SQL Server specific.

An additional concept I must discuss is how SQL Server stores huge data values such as image data commonly referred to as BLOBs or Binary Large Objects. When a data type needs to store data that is potentially greater in size than a SQL Server page, special handling is required. Instead of storing the actual data in the page with the record, only a pointer to another physical memory location is stored. The pointer is a 16-byte binary (the same data type as I've discussed) that points to a different page in the database. The page may have stored the value of one or more BLOBs, and a BLOB may span many pages.

Uses of `binary` fields, especially the `image` field, is fairly limited. Basically, they can be used to store any binary values that aren't dealt with by SQL Server. You can store text, JPEG and GIF images, even Word documents and Excel spreadsheet data. So why are the uses limited? The simple answer is performance. It's generally a bad idea to use the `image` data type to store very large files because it's slow to retrieve them from the database. When you need to store image data in the database, you'll generally just store the name of a file on shared access. The accessing program will simply use the filename to go to external storage to fetch the file. File systems are built for nothing other than storing and serving up files, so you use them for their strengths.

Character Strings

Most data that is stored in SQL Server uses character data types. In fact, usually far too much data is stored in character data types. Frequently, character fields are used to hold noncharacter data, such as numbers and dates. Although this may not be technically wrong, it isn't ideal. For starters, to store a number with eight digits in a character string requires at least 8 bytes, but as an integer it requires 4 bytes. Searching on integers is far easier because 1 always precedes 2, whereas 11 precedes 2 in character strings. Additionally, integers are stored in a format that can be manipulated using intrinsic processor functions as opposed to having SQL Server-specific functions to deal with the data.

char

The `char` data type is used for fixed-length character data. You must choose the size of the data that you wish to store. Every value will be stored with exactly the same number of characters, up to a maximum of 8060 bytes. Storage is exactly the number of bytes as per the column definition, regardless of actual data stored; any remaining space to the right of the last character of the data is padded with spaces.

> *Note that versions of SQL Server earlier than 7.0 implement* `char` *columns differently. The setting* `ANSI_PADDING` *determines exactly how this is actually handled. If this setting is on, the table is as I've described; if not, data will be stored as I'll discuss in the "varchar" section that follows. It's usually best to have the ANSI settings set to the most standard (or even the most restrictive) settings, as these settings will help you to avoid making use of SQL Server enhancements in your coding that are likely to not exist from version to version.*

The maximum limit for a `char` is 8060 bytes, but it's very unlikely that you would ever get within a mile of this limit. For character data where large numbers of characters are anticipated, you would use one of the other character types. The `char` data type should only be employed in cases where you are guaranteed to have *exactly* the same number of characters in every row and each row isn't `NULL`. As such, you'll employ this type very seldom, because most data doesn't fit the equal length stipulation.

There is one example where chars would be used that you'll come across fairly frequently: identification numbers. These can contain mixed alphabetic and numeric characters, such as vehicle identification numbers (VINs). Note that this is actually a composite attribute, as you can determine many things about the automobile from its VIN.

Another example where a char field is usually found is Social Security numbers (SSNs).

varchar

For the varchar data type, you choose the maximum length of the data you wish to store, up to 8000 bytes. The varchar data type is far more useful than the char, as the data doesn't have to be of the same length and SQL Server doesn't pad out excess memory with spaces. There is some additional overhead in storing variable-length data.

Use the varchar data type when your data is usually small, say, several hundred bytes or less. The good thing about varchar columns is that, no matter how long you make the maximum, the data stored is the text in the column plus the few extra bytes that specify how long the data is.

You'll generally want to choose a maximum limit for your data type that is a reasonable value, large enough to handle most situations but not too large as to be impractical to deal with in your applications and reports. For example, take people's first names. These obviously require the varchar type, but how long should you allow the data to be? First names tend to be a maximum of 15 characters long, though you might want to specify 20 or 30 characters for the unlikely exception.

varchar data is the most prevalent storage type for non-key values that you'll use.

text

The text data type is used to store larger amounts of character data. It can store a maximum of 2,147,483,647 ($2^{31} - 1$) characters, that is, 2GB. Storage for text is very much like that for the image data type. However, an additional feature allows you to dispense with the pointer and store text data directly on the page. This is allowed by SQL Server if the following conditions are met:

- The column value is smaller than 7000 bytes.
- The data in the entire record will fit on one data page.
- The Text In Row table option is set using the sp_tableoption procedure, like so:

```
        EXECUTE sp_tableoption 'TableName', 'text in row', '1000'
```

In this case, any text columns in the table `TableName` will have their text stored with the row instead of split out into different pages, until the column reaches 1000 or the entire row is too big to fit on a page. If the row grows too large to fit on a page, one or more of the text columns will be removed from the row and put onto separate pages. This setting allows you to use `text` values in a reasonable manner, so that when the value stored is small, performance is basically the same as for a `varchar` column.

Hence, you can now use the `text` data type to hold character data where the length can vary from 10 characters to 50K of text. This is an especially important improvement for notes-type columns, where users want to have space for unlimited text notes, but frequently only put 10 or 20 characters of data in each row.

Another issue with `text` fields is that they can't be indexed using conventional indexing methods. As discussed in the previous appendix, if you need to do heavy searching on text columns, you can employ the full-text searching capabilities of SQL Server 2000. It should also be noted that, even if you could index text columns, it might not be that good an idea. Because text columns or even large `varchar` columns can be so large, making them fit in the 900-byte limit for an index is unfeasible anyway.

Unicode Character Strings

So far, the character data types I've been discussing have been for storing typical ASCII data. In SQL Server 7.0 (and NT 4.0), Microsoft implemented a new standard character format called Unicode. This specifies a 16-bit character format that can store characters beyond just the Latin character set. In ASCII–a 7-bit character system (with the 8 bits for Latin extensions)–you were limited to 256 distinct characters, which was fine for most English speaking people, but was insufficient for other languages. Asian languages have a character for each different syllable and are nonalphabetic; Middle Eastern languages use several different symbols for the same letter according to its position in the word. So a standard was created for a 16-bit character system, allowing you to have 65,536 distinct characters.

For these data types, you have the `nchar`, `nvarchar`, and `ntext` data types. They are exactly the same as the similarly named types (without the `n`) that I've already described, except for one thing: Unicode uses double the number of bytes to store the information so it takes twice the space, thus cutting by half the number of characters that can be stored.

One quick tip. If you want to specify a Unicode value in a string, you append an N to the front of the string, like so:

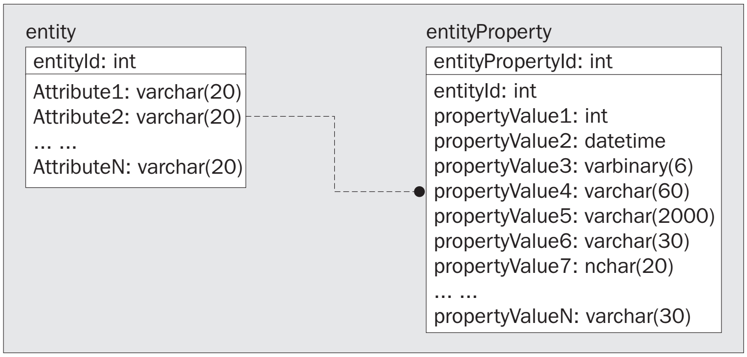
```
SELECT N'Unicode Value'
```

Variant Data

A variant data type allows you to store almost any data type that I've discussed. This allows you to create a column or variable where you don't know exactly what kind of data will be stored. This is a new feature of SQL Server 2000. The name of the new data type is `sql_variant` and it allows you to store values of various SQL Server-supported data types–except `text`, `ntext`, `timestamp`, and `sql_variant`. (It may seem strange that you can't store a variant in a variant, but all this is saying is that the `sql_variant` data type doesn't actually exist as such–SQL Server chooses the best type of storage to store the value you give to it.)

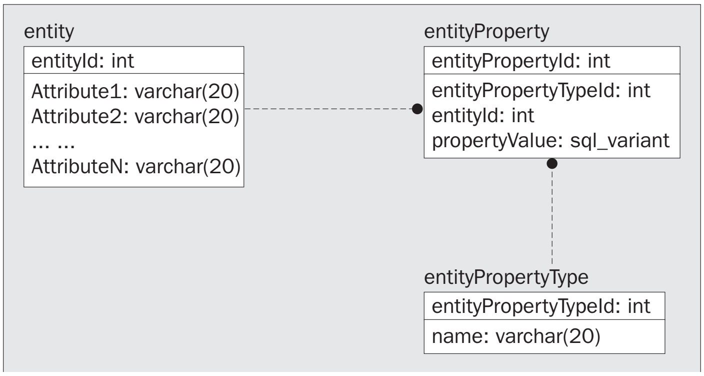
The Advantages of Variant

The `sql_variant` data type will allow you to create user-definable "property bag" type tables that will help you avoid having very long tables with many columns that may or may not be filled in. Take the `entityProperty` table in the following example:



In this example, you have N columns with N data types that are all nullable to let the user store any one or more of the values. This type of table is generally known as a **sparse table**. The problem with this example is that, if the user comes up with a new property, you'll be forced to modify the table, the UI, and any programs that refer to the columns in the table.

On the other hand, if you implement the table in the following manner:



then each of the properties that you implemented in the previous example, by adding a column to the `entityProperty` table, would now be added as an instance in the `entityPropertyType` table, and the value would be stored in the `propertyValue` column. Whatever type of data needed for the property could be stored as a `sql_variant`.

The `entityPropertyType` table could be extended to include many other properties without the user having to carry out major changes to the database. And if you implement your reporting solution in such a way that your new reports know about any changes, you won't have to recode for any new properties.

The Disadvantages of Variant

It isn't easy to manipulate the data once it has been stored in a `sql_variant` column. I'll leave it to the reader to fully read the information in SQL Server Books Online that deals with variant data. The issues to consider are as follows:

- Assigning data from a `sql_variant` column to a stronger typed data type–you have to be very careful as the rules for casting a variable from one data type to another are difficult, and may cause errors if the data can't be cast. For example, a `varchar(10)` value `'Not a Date'` can't be cast to a `datetime` data type. Such problems really become an issue when you start to retrieve the variant data out of the `sql_variant` data type and try to manipulate it.
- `NULL sql_variant` values are considered to have no data type–hence, you'll have to deal with `sql_variant` nulls differently from other data types.
- Comparisons of variants to other data types could cause programmatic errors, because of the data type of the instance of the `sql_variant` value–the compiler will know if you try to run a statement that compares two incompatible data types, such as `@intVar = @varcharVar`. However, if the two variables in question were defined as `sql_variants`, and the data types don't match, then the values won't match due to the data type incompatibilities.

There is a function provided that discovers the data type of a value stored in a variant column:

```
DECLARE @varcharVariant sql_variant
SET @varcharVariant = '1234567890'
SELECT @varcharVariant AS varcharVariant,
   SQL_VARIANT_PROPERTY(@varcharVariant,'BaseType') as baseType,
   SQL_VARIANT_PROPERTY(@varcharVariant,'MaxLength') as maxLength
```

This returns

```
VarcharVariant    baseType      maxLength
-------------------------------------------------------------
1234567890        varchar       10
```

Other Data Types

The following data types are somewhat less frequently employed but are still very useful.

timestamp (or rowversion)

The `timestamp` data type is a database-wide unique number. When you have a `timestamp` column in a table, the value of the `timestamp` column changes for each modification to each row. The value in the `timestamp` column is guaranteed to be unique across all tables in the data type. It also has a pretty strange name, as it doesn't have any time implications–it's merely a unique value to tell you that your row has changed.

The `timestamp` column of a table (you may only have one) is usually used as the data for an optimistic locking mechanism. I'll discuss this further later in this appendix when I talk about physical-only columns.

The `timestamp` data type is a mixed blessing. It's stored as an 8-byte `varbinary` value. Binary values aren't always easy to deal with and depend on which mechanism you are using to access your data.

When using ODBC, or OLE DB and ADO, they can be tricky at first. This is due to the fact that they are binary values, and to retrieve them you have to use BLOB chunking mechanisms such as the following:

```
SET nocount on
CREATE TABLE testTimestamp
(
   value   varchar(20) NOT NULL,
   auto_rv   timestamp NOT NULL
)

INSERT INTO testTimestamp (value) values ('Insert')

SELECT value, auto_rv FROM testTimestamp
UPDATE testTimestamp
SET value = 'First Update'

SELECT value, auto_rv from testTimestamp
UPDATE testTimestamp
SET value = 'Last Update'

SELECT value, auto_rv FROM testTimestamp
```

which returns

```
value                    auto_rv
-------------------------------------------------------------------
Insert                   0x0000000000000089
```

```
Value                    auto_rv
------------------------------------------------------------------------
First Update             0x000000000000008A

Value                    auto_rv
------------------------------------------------------------------------
Last Update              0x000000000000008B
```

You didn't touch the `auto_rv` variable and yet it incremented itself twice. However, you can't bank on the order of the `timestamp` variable being sequential, as updates of other tables will change this. It's also in your best interest not to assume that the number is an incrementing value in your code. How timestamps are implemented is a detail that may change in the future (from SQL Server 2000 Books Online, "The definition of timestamp in a future release of SQL Server will be modified to align with the SQL-99 definition of timestamp."). If a better method of building database-wide unique value comes along that is even a hair faster, they will likely use it.

You may create variables of the `timestamp` type for holding timestamp values, and you may retrieve the last used timestamp via the `@@dbts` global variable.

The topic of optimistic locks is covered in greater detail in Chapter 13.

uniqueidentifier

A globally unique identifier (GUID) is fast becoming a mainstay of Microsoft computing. The name says it all–they are globally unique. According to the way that GUIDs are formed, there is a tremendously remote chance that there will ever be any duplication in their values. They are generated by a formula that includes the usage of the network card identification number (if one exists), the current date and time, a unique number from the CPU clock, and some other "magic numbers."

In SQL Server 2000, the `uniqueidentifier` has a very important purpose. When you need to have a unique key that is guaranteed to be unique across databases and servers, for example:

```
DECLARE @guidVar uniqueidentifier
SET @guidVar = newid()

SELECT @guidVar as guidVar
```

returns

```
guidVar
--------------------------------------------------------------------
6C7119D5-D48F-475C-8B60-50D0C41B6EBF
```

They are actually stored as 16-byte binary values. Note that it isn't exactly a *straight* 16-byte binary value. You may not put just any binary value into a uniqueidentifier column, as the value must meet the criteria for their generation, which aren't well documented for obvious reasons.

If you need to create a uniqueidentifier column that is auto-generating, there is a property you can set in the CREATE TABLE statement (or ALTER TABLE for that matter). It's the rowguidcol property, and it's used like so:

```
CREATE TABLE guidPrimaryKey
(
   guidPrimaryKeyId uniqueidentifier NOT NULL
   rowguidcol DEFAULT newId(),
   value varchar(10)
)
```

I've introduced a couple of new things here—rowguidcol and default values. Suffice it to say that, if you don't provide a value for a column in an insert operation, the default operation will provide it. In this case, you use the newId() function to get a new uniqueidentifier. So when you execute the following insert statement:

```
INSERT INTO guidPrimaryKey(value)
VALUES ('Test')
```

and then run the following command to view the data entered:

```
SELECT *
FROM guidPrimaryKey
```

this returns

```
guidPrimaryKeyId                                    value
-----------------------------------------------------------------------------
8A57C8CD-7407-47C5-AC2F-E6A884C7B646                Test
```

The rowguidcol property of a column built with the uniqueidentifier notifies the system that this is just like an identity column value for the table—a unique pointer to a row in a table. Note that neither the identity nor the rowguidcol properties guarantee uniqueness. To provide such a guarantee, you have to implement your tables using unique constraints.

It would seem that the uniqueidentifier would be a better way of implementing primary keys, because when they are created, they are unique across all databases, servers, and platforms. However, there are two main reasons why you won't use uniqueidentifier columns to implement all of your primary keys:

- **Storage requirements:** As they are 16 bytes in size, they are considerably more bloated than a typical integer field.
- **Typeability:** As there are 36 characters in the textual version of the GUID, it's very hard to actually type the value of the GUID into a query, and it isn't easy to enter.

cursor

A cursor is a mechanism that allows row-wise operations instead of the normal set-wise way.
The `cursor` data type is used to hold a reference to a SQL Server T-SQL cursor. You may not use a `cursor` data type as a column in a table. The only use for this data type is in T-SQL code to hold a reference to a cursor.

table

The `table` data type has a few things in common with the `cursor` data type, but holds a reference to a result set. The name of the data type is really a pretty bad choice, as it will make functional programmers think that they can store a pointer to a table. It's actually used to store a result set as a temporary table. In fact, the `table` is exactly like a temporary table in implementation. The following is an example of the syntax needed to employ the `table` variable type:

```
DECLARE @tableVar TABLE
(
   id int IDENTITY,
   value varchar(100)
)
INSERT INTO @tableVar (value)
VALUES ('This is a cool test')

SELECT id, value
FROM @tableVar
```

which returns

```
Id          value
----------------------------------------
1           This is a cool test
```

As with the `cursor` data type, you may not use the `table` data type as a column in a table, and it may only be used in T-SQL code to hold a result set. Its primary purpose is for returning a table from a user-defined function as in the following example:

```
CREATE FUNCTION table$testFunction
(
   @returnValue varchar(100)

)
RETURNS @tableVar table
(
     value varchar(100)
)
AS
BEGIN
   INSERT INTO @tableVar (value)
   VALUES (@returnValue)

   RETURN
END
```

Once created, you can invoke it using the following syntax:

```
SELECT *
FROM dbo.table$testFunction('testValue')
```

which returns the following:

```
value
--------------
testValue
```

Summary

Choosing the proper data type for a column is a very important task. In this appendix, I've shown you each of the data types that are built in and part of SQL Server for every database. Each data type has its place, depending on what data is needed to be stored. The following is a list of the data types you've looked at:

- **Precise numeric data: Stores data with no possible loss of precision**
  - `bit`: Stores either 1, 0, or NULL, used for Boolean type columns
  - `tinyint`: Nonnegative values between 0 and 255
  - `smallint`: Integers between –32,768 and 32,767
  - `int`: Integers between 2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31} - 1$)
  - `bigint`: Integers between 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (that is, -2^{63} to $2^{63} - 1$)
  - `decimal`: All numbers between $-10^{38} - 1$ through $10^{38} - 1$
  - `money`: Values from –922,337,203,685,477.5808 through 922,337,203,685,477.5807
  - `smallmoney`: Values from –214,748.3648 through 214,748.3647
- **Approximate numeric data: Stores approximations of numbers and gives a very large range of values**
  - `float (N)`: Values in the range from –1.79E + 308 through to 1.79E + 308
  - `real`: Values in the range from –3.40E + 38 through to 3.40E + 38. Real is a synonym for a `float(24)` data type.

- **Date and time: Stores date values, including time of day**
    - `smalldatetime`: Dates from January 1, 1900, through to June 6, 2079
    - `datetime`: Dates from January 1, 1753, to December 31, 9999
- **Binary data: Strings of bits, for example files or images**
    - `binary`: Fixed-length binary data up to 8000 bytes long
    - `varbinary`: Variable-length binary data up to 8000 bytes long
    - `image`: Really large binary data, max length of $2^{31} - 1$ (2,147,483,647) bytes or 2GB
- **Character (or string) data**
    - `char`: Fixed-length character data up to 8000 characters long
    - `varchar`: Variable-length character data up to 8000 characters long
    - `text`: Large text values, max length of $2^{31} - 1$ (2,147,483,647) bytes or 2GB
    - `nchar`, `nvarchar`, `ntext`: Unicode equivalents of `char`, `varchar`, and `text`
- **Other data types**
    - `variant`: Stores any data type
    - `timestamp` (or rowversion): Used for optimistic locking
    - `uniqueidentifier`: Stores a globally unique identifier (GUID) value