# Beginning SQL Server 2005 Express Database Applications

with Visual Basic Express and Visual Web Developer Express From Novice to Professional

■ ■ ■

Rick Dobson

Apress®

**Beginning SQL Server 2005 Express Database Applications**
**with Visual Basic Express and Visual Web Developer Express From Novice to Professional**

**Copyright © 2006 by Rick Dobson**

ISBN (pbk): 1-59059-523-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Tony Davis and Matthew Moodie
Technical Reviewer: Cristian Lefter
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editors: Damon Larson and Freelance Editorial Services
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositors: Dina Quan and Diana Van Winkle, Van Winkle Design Group
Proofreader: April Eddy
Indexer: Valerie Perry
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 4

# Data Types, Tables, and Constraints

The main purpose of most relational database managers, such as SQL Server Express, is to store data in a way that can help to model organizations or systems. Relational databases store data in tables. The tables frequently represent entities that are being modeled, such as the students and classes at a college. By expressing relationships between distinct tables, you facilitate the retrieval and maintenance of information for the tables in a database.

This chapter drills down on data types and table design techniques. You need to learn about SQL Server Express (SSE) data types, because picking the wrong data types can waste storage space and cause your database solutions to run slowly. Picking the right data types can give you faster, easier, and more flexible options for retrieving data from your database. This chapter begins by acquainting you with the dozens of data types that SSE makes available so that you can make informed decisions about which data types to use.

The chapter covers data design techniques, introducing you to the basics of creating tables, including the use of data types in table definitions. You also learn how to specify constraints in table definitions that help to manage the integrity and usefulness of the data in a database. Different types of constraints are optimized to enhance data integrity and the accessibility of data within tables. Several especially important constraint types that you'll learn about in this chapter include the following:

- A `PRIMARY KEY` constraint designates one or more columns that are unique across all rows in a table. This type of constraint can avoid duplicate rows within a single table and help to define relationships between tables.
- A `CHECK` constraint lets you specify a Boolean expression that must be satisfied for a row to enter a table. For example, you can specify a `CHECK` constraint to avoid a situation where a student inadvertently enters her birth date for a year in the future.
- `FOREIGN KEY` constraints in coordination with either `PRIMARY KEY` or `UNIQUE` constraints facilitate managing data integrity and accessibility between pairs of tables. `FOREIGN KEY` constraints are optimized for designating one-to-many and many-to-many relationships between tables.

## Learning About Data Types

SQL Server Express supports dozens of native data types. The data type that you use to store a column's values says a lot about the data in that column. Some data types are tailored for storing characters, whereas others are best for holding large numbers or small numbers. Picking the right data type for a column can often save storage space and generate performance improvements. This section organizes data types into groups, and subgroups in some cases, to help you understand which data types to use in a particular situation. The three main categories are

- Numbers and dates
- Character and binary byte streams
- Miscellaneous

---

**Note** As mentioned in previous chapters, SSE is merely an edition of SQL Server 2005. It shares a common core database engine with commercial editions of SQL Server 2005. This commonality between SSE and other SQL Server 2005 editions is particularly evident for data types. The SSE data types are exactly the same as those in other editions of SQL Server 2005. For this reason, you can refer interchangeably to SQL Server data types and SSE data types.

---

## Numbers and Dates

Data types for numbers have four common characteristics: precision, scale, length, and a range of legitimate values.

- *Precision* refers to how accurately a data type can represent a number value. You can think of a data type's precision as the maximum number of digits available to specify a value.
- *Scale* refers to the number of digits to the right of a decimal point.
- *Length* designates the number of bytes used to store a value.
- The *range of legitimate values* for a data type provides a very concrete way of appreciating the role of a data type.

The precision, scale, length, and range of legitimate values relate to one another. A data type's length in bytes sets an upper limit on the precision of values that a data type can represent. The more bytes a data type has, the more precise the values that it can represent. The scale for a data type can never be greater than the precision. That is, you cannot have more digits to the right of the decimal point than the total number of digits in a value. The precision, scale, and length set constraints on the range of values that a data type can store.

---

**Tip** You should generally select a data type for a column that has the smallest possible length and a legitimate range of values for all the values that a column can hold.

---

Number data types can represent values either precisely or approximately. Several data types for storing numbers support the storage of integer quantities, such as 1, 2, or 100. Other values, such as 1 divided by 3, cannot be represented precisely. To accommodate values that cannot be represented precisely or whose length would be prohibitive to represent precisely, SQL Server offers approximate data types.

In many cases, data types for precise numbers represent integer values with a scale of zero (no digits to the right of the decimal point). Some data types for numbers represent values precisely even while they permit the inclusion of digits to the right of the decimal point. Three data types like this are `money`, `smallmoney`, and `decimal` (or `numeric`).

---

**Note** The `decimal` and `numeric` data types are synonyms for each other.

---

SQL Server offers two data types for representing date and time values, `datetime` and `smalldatetime`. These data types store their values as numbers with digits to the left and right of a decimal point. Digits to the left of the decimal point represent whole days. Digits to the right of the decimal point denote fractions of a day. The `datetime` and `smalldatetime` data types have different lengths, which enable these two data types to represent a different range of date and time values to different levels of accuracy.

`datetime` and `smalldatetime` have characteristics that closely resemble the characteristics for numbers. For example, number data types and data types for dates and times have fixed lengths. In addition, both sets of data types have a legitimate range of values (from a minimum number or beginning date and time to a maximum number or ending date and time). The accuracy of date and time data types (1 minute for `smalldatetime` vs. 3.33 milliseconds for `datetime`) corresponds roughly to scale.

## Bit Data Type

The `bit` data type is for values that can be either 0 or 1. This makes the `bit` data type particularly appropriate for representing column values that can be either true (1) or false (0). Another common use for the `bit` data type is to represent a check box on a questionnaire that can either be checked (1) or not checked (0).

`bit` values can be unknown (or null) besides 0 or 1. A null value represents information that is not known. For a table describing businesses, you can have a column indicating whether a business is incorporated in the state of Delaware. At the time that you enter a row for a business, the characteristic can be yes, no, or unknown (because you do not have information about the state of incorporation for a business). Unknown is different from either yes for incorporated within Delaware or no for not incorporated within Delaware.

---

■ **Note** Null data values, usually represented by the NULL keyword, can apply to nearly all SQL Server data types. Whenever a column value is not known or missing, the column's value is null. A null value is different from zero or an empty string. Before the introduction of null values, developers used to represent missing values with some arbitrary value and did not need to follow any standard from one database to another. Using null values to represent missing data ensures your database follows widely used standards. Null values have distinctive features. For example, any value concatenated with a NULL returns another NULL. A null value does not equal (=) another null value.

---

A `bit` data type can correspond to a bit within a byte of computer storage. The SQL Server database engine automatically groups `bit` data values to fill the bits within a byte. If a table has between 1 and 8 `bit` values per table row, SQL Server stores the values in a byte. Similarly, from 9 to 16 `bit` values are stored within 2 bytes.

## Integer Data Types

The four integer data types are `tinyint`, `smallint`, `int`, and `bigint`. The precision of these data types ranges from 3 through 19 digits. The scale for all integer data types is 0; no digits appear to the right of the decimal point. The integer data types have lengths of 1, 2, 4, and 8 bytes, respectively. The `tinyint` data type is distinct from the other three integer data types because it represents just positive values and zero, but the other three data types represent positive and negative values, including zero.

- `tinyint`
  - range of values: 0 through 255
  - precision: 3
  - scale: 0
  - length: 1
- `smallint`
  - range of values: –32, 768 (–2^15) through 32, 767 (2^15 – 1)
  - precision: 5
  - scale: 0
  - length: 2
- `int`
  - range of values: –2,147,483,648 (–2^31) through 2,147,483,647 (2^31 – 1)
  - precision: 10
  - scale: 0
  - length: 4
- `bigint`
  - range of values: –9,223,372,036,854,775,808 (–2^63) through 9,223,372,036,854,775,807 (2^63 – 1)
  - precision: 19
  - scale: 0
  - length: 8

## Currency Data Types

The `money` and `smallmoney` data types explicitly target the representation of currency values. Both data types have four digits to the right of the decimal point, but all values are precisely represented. These four places allow for the precise representation of a ten-thousandth of a currency unit. Aside from the explicit targeting of currency applications, the `smallmoney` and `money` data types are very similar to the `int` and `bigint` data types. The specifications for the `smallmoney` and `money` data types are

- `smallmoney`
  - range of values: –214,748.3648 through 214,748.3647
  - precision: 10
  - scale: 0
  - length: 4
- `money`
  - range of values: –922,337,203,685,477.5808 through 922,337,203,685,477.5807
  - precision: 19
  - scale: 0
  - length: 8

## Date and Time Data Types

The `datetime` and `smalldatetime` data types represent date and time values similarly to the way that `money` and `smallmoney` represent currency values. The similarity follows from the fact that the `money` data type represents a wider range of currency values than the `smallmoney` data type, and the `datetime` data type can denote a wider range of dates than the `smalldatetime` data type. In addition, the `datetime` and `money` data types are both 8 bytes long, whereas the `smalldatetime` and `smallmoney` data types are both 4 bytes long.

The two data types for date and time values diverge in one major way from the two data types for currency. This is because the `smalldatetime` data type does not have the same level of accuracy for time that the `datetime` data type does. Recall that both `money` and `smallmoney` represent currency values to four places after the decimal point. The `smalldatetime` data type distinguishes time to the level of 1-second increments, but the `datetime` data type can represent time to the level of 3.33-millisecond increments. For most business applications, the finer resolution of the `datetime` data type relative to the `smalldatetime` data type is not a driving factor for choosing the `datetime` data type in favor of the `smalldatetime` data type. Remember, you should always choose the smallest possible data type for a column in a table.

The specifications for the two date and time data types have a different format than other data types for representing number values. Although SQL Server does internally represent dates and times as number values with a precision and scale, it is not relevant to think about time units in the metrics of precision and scale. Instead, you should use the date range and the time resolution specifications in the following list:

- `smalldatetime`
    - date range: January 1, 1900, through June 6, 2079
    - time resolution: 1 minute
    - length: 4
- `datetime`
    - date range: January 1, 1753, through December 31, 9999
    - time resolution: 3.33 milliseconds
    - length: 8

## Decimal and Numeric Data Types

The `decimal` data type and `numeric` data type both describe the same kind of data. A table designer can specify both the precision (`p`) and scale (`s`) of table columns with either a `numeric` or `decimal` data type at the time that they designate the column for the table. The value of `p` must be less than or equal to 38, and the value of `s` must be greater than or equal to 0, as well as less than or equal to `p`. Although `decimal` and `numeric` data type values permit digits to the right of the decimal, they precisely represent values. For this reason, you can use `decimal` and `numeric` data types for currency applications in addition to the `money` and `smallmoney` data types.

---

■**Note** As indicated previously, the `numeric` data type is a synonym for the `decimal` data type. The `numeric` data type, despite the *Random House Unabridged Dictionary* definition for the term *numeric*, does not mean "of or pertaining to numbers." The term *numeric* when used with respect to SQL Server data types has a much more limited meaning. For these reasons, you should be careful how you use the word *numeric* when describing numbers in SQL Server applications.

---

Unlike other data types for numbers, the specifications for the `decimal` and `numeric` data types can vary depending on the values that a table designer explicitly assigns to `p` and `s`.

- `decimal` or `numeric`
  - range of values: –10^38 + 1 through 10^38 – 1
  - precision: minimum of 0 and maximum of 38
  - scale: minimum of 0 and maximum of 38
  - length: varies (for `p` of 1–9, the length is 5; for `p` of 10–19, the length is 9; for `p` of 20–28, the length is 13; for `p` of 29–38, the length is 17)

An example may help to clarify how to use of the `decimal` data type as it approaches its limiting upper value. The following script declares the `@dec1` local variable as a `decimal` data type with a precision of 38 and a scale of 0. Next, a `POWER` function nested within a `CAST` function creates a value of 1,000,000,000 in `@dec1`. Then, a `SELECT` statement with a product as its list value returns a `decimal` value of 1 followed by 37 zeroes for a total of 38 digits. The second item in the `SELECT` list with the `FLOOR` and `LOG10` functions actually computes the number of digits in the product.

```
SET NOCOUNT ON
DECLARE @dec1 decimal(38,0)
SET @dec1 = CAST(POWER(10,9) as decimal(38,0))
SELECT @dec1 * @dec1 * @dec1 * @dec1 * 10 AS 'Large Decimal value',
    FLOOR(LOG10(@dec1 * @dec1 * @dec1 * @dec1 * 10)
    + 1) AS 'Number of digits'
```

The following listing shows the result generated by the preceding script. Notice the large `decimal` value has 38 digits. Attempting to generate a `decimal` data type value with 39 digits (e.g., by replacing the 10 in the product with 100) will generate an arithmetic overflow error because the result of the product expression is too large to represent as a `decimal` data type. The script generating this listing is available as Ch04Decimal.sql.

```
Large Decimal value                     Number of digits
--------------------------------------- ----------------------
10000000000000000000000000000000000000  38
```

## Approximate Data Types

`real` and `float` data types can represent values approximately instead of precisely, as with integer data types, such as `int` and `bigint`. `real` and `float` data types have a null scale setting because all values have a floating decimal point. The `real` and `float` data types follow an IEEE (Institute of Electrical and Electronic Engineers) specification for approximate data types. Because approximate values are subject to rounding error, they are not suitable for financial operations or other cases where precise results are required.

■ **Tip** Approximate data types are best suited for scenarios when you are working with engineering or scientific data values. Avoid using `real` and `float` data types whenever you need precise results, such as in currency calculations.

There are three advantages for approximate data types relative to precise data types.

- Some number values without a precise representation, such as 1 divided by 3, can only be represented approximately (you can't represent them precisely).
- In exchange for not representing values precisely, approximate data types can represent a wider range of values than corresponding integer data types.
- Approximate data types save storage space, compared to precise data types, because they have a smaller length.

Here are the specifications for the `real` and `float` data types:

- `real`
    - range of values: –3.40E + 38 through 3.40E + 38
    - precision: from 1 to 7
    - scale: Null
    - length: 4
- `float`
    - range of values: –1.79E + 308 through 1.79E + 308
    - precision: from 1 to 15
    - scale: Null
    - length: 8

The following script helps to contrast the `float` data type from the `decimal` data type. Notice that the script follows the same general design as the preceding script for computing a large `decimal` value, but this script uses a `float` data type instead of a `decimal` data type. As a consequence, you can compute larger values, such as those with 39 digits. By computing with a multiple of 100 instead of a multiple of 10, the following script generates a value that is an order of magnitude larger than in the preceding sample for the `decimal` data type.

```
SET NOCOUNT ON
DECLARE @float1 float
SET @float1 = CAST(POWER(10,9) as float)
SELECT @float1 * @float1 * @float1 * @float1 * 100 AS 'Large Float value',
    FLOOR(LOG10(@float1 * @float1 * @float1 * @float1 * 100)
    + 1) AS 'Number of digits'
```

The following listing confirms the operation of the preceding script. The computed value (`@float1 * @float1 * @float1 * @float1 * 100`) is a number with 1 followed by 38 zeroes, for a total of 39 digits. The preceding script with the `float` script will return an error if you run it with a `decimal` data type instead of a `float` data type. The script generating the following listing is available as Ch04Float.sql.

```
Large Float value      Number of digits
---------------------- ----------------------
1E+38                  39
```

The `float` data type represents values precisely whenever its internal architecture can avoid representing a value approximately. For example, 7 divided by 10 with `float` values results in precisely .7. Similarly, the `float` data type can precisely represent .0001. However, the difference between 7 divided by 10 and .0001 is a number that a `float` data type can only approximately represent.

The following script contrasts float and decimal data types to indicate how a decimal data type can return a precise value for a computation for which a float data type returns an approximate value. Four local variables (@dec1–@dec4) are declared, with a decimal data type having four places to the right of the decimal and a total of 15 digits. This matches the number of digits for float data type accuracy. Four local variables (@float1–@float4) are also declared with a float data type. The script computes the difference between the quotient of 7 divided by 10 less .0001. The computation demonstrates the contrasting results from float and decimal data types. The script generating the following listing is available as Ch04FloatDecimal.sql.

```
SET NOCOUNT ON
DECLARE @float1 float, @float2 float, @float3 float, @float4 float
DECLARE @dec1 decimal(15,4), @dec2 decimal(15,4),
        @dec3 decimal(15,4), @dec4 decimal(15,4)
DECLARE @pointseven decimal(15,4)
SET @float1 = 7
SET @float2 = 10
SET @float3 = .0001
SET @float4 = @float1/@float2 - @float3
SET @dec1 = 7
SET @dec2 = 10
SET @dec3 = .0001
SET @dec4 = @dec1/@dec2 - @dec3
SET @pointseven = .7
SELECT @float4 '@float4', @float3 '@float3',
       @pointseven - @float4 'float diff'
SELECT @dec4 '@dec4', @dec3 '@dec3',
       @pointseven - @dec4 'dec diff'
```

The following listing shows the difference between the float and the decimal results. The preceding script compares each difference with a local variable named @pointseven that is equal to .7. This local variable has a decimal data type. The decimal difference is precisely .0001. The float difference is approximately .0001, that is, 9.9999999999989E–05.

```
@float4                @float3                float diff
---------------------- ---------------------- ----------------------
0.6999                 0.0001                 9.9999999999989E-05

@dec4                  @dec3                  dec diff
---------------------- ---------------------- ----------------------
0.6999                 0.0001                 0.0001
```

## Character and Binary Byte Streams

A sequence of 1 or more bytes is a byte stream. SQL Server Express can decode a byte stream to characters, such as letters, numbers, and symbols, or it can just save and retrieve a byte stream without decoding the byte(s). Two common ways for decoding byte streams are with a character code set based on a code page or with a Unicode character set. Decoded byte streams from a database frequently contain names, such as student names or class titles. Binary byte strings are saved and retrieved as raw byte values. A client application, such as a graphics viewer, can decode a binary byte stream into an image.

---

■**Note** A code page is a mapping that translates byte values to characters. There are many code pages associated with different languages and locales. Just as a single code page maps byte values to characters for a single language and locale, a Unicode character set maps byte values to characters for all modern languages. There is more than one Unicode character set. SQL Server Express uses the UCS-2 Unicode standard.

---

Character codes typically match characters to single-byte representations. When a database application is meant for use with one language in a single locale, such as United States English, then a character code set is an especially useful way of translating bytes to characters. This recommendation uses a single code page that translates bytes to characters.

As the number of regional locales that an application has to serve grows beyond a few, it can become exceedingly difficult to find a single code page for translating byte streams properly across all locales. In these circumstances, switching to Unicode characters, which use 2 bytes per character, simplifies the process of translating byte streams to characters in a consistent way across locales. Recall that Unicode character sets are applicable to all modern languages used for business throughout the world. Unicode characters are also especially appropriate for selected Asian languages with more characters than a single byte can represent, such as Simplified Chinese, Traditional Chinese, Japanese, and Korean.

Each of the three types of byte stream types (character, Unicode character, and binary) has three corresponding data types. The first basic data type is for fixed-length values, such as Social Security numbers or credit card numbers. The second basic data type is for variable-length strings. This kind of data type is particularly appropriate for columns storing names, such as first name or last name, in which the values within a column can vary substantially in length across a table's rows. The third basic data type is for byte streams that exceed 8,000 bytes. The column values designated by these long strings are sometimes called large objects.

---

■**Note** It is common in database applications to use a character data type to represent an identification "number," such as a credit card number or a postal code.

---

Because large objects can slow the retrieval of data from a table, you may wish to isolate large objects in a separate table that points back to another table with nonlarge objects. This limits the impact of retrieving large objects to occasions when you explicitly need to access them. By storing large objects in a database, you can secure access to them via database security features. Another option is to store large objects as files and then include path and file names in your database. This practice relieves a database of storing and retrieving large objects. In this second scenario, you can manage access to paths containing files with large objects via Windows access control lists.

## Character Data Types

The three character data types are `char`, `varchar`, and `text`. These data types correspond, respectively, to the fixed-length, variable-length, and large object data types. They use a single byte per character to represent non-Unicode characters. Byte values are decoded to characters with the default code page for an SSE instance, which is automatically set to a computer's Windows regional setting unless you override this setting during installation. You can designate a character constant value by including the value in single quotes—`'this is a character constant'`.

Use the char data type for table columns that hold values that are all of almost the same length. A syntax such as char(n) represents the maximum number of characters in a column with the char data type, where the n parameter can assume values of 1 through 8,000. The char data type is very efficient for processing character data because SSE can process a fixed number of characters per column value without the need to determine when each character string ends.

The varchar data type is appropriate for columns that will hold character strings that differ substantially across rows. Use a syntax of varchar(n) to denote a varchar data type, where the n parameter represents the maximum number of characters. As with the char data type, n can assume values from 1 through 8,000. The actual length in bytes of a varchar column value is n + 2. When the column values are nearly all of the same length, a char data type will process faster and require less storage space than a varchar data type.

■ **Note** When you have a need for a varchar data type that can accommodate more than 8,000 characters, use max instead of n. The varchar(max) syntax is particularly appropriate in situations where you might consider the text data type.

As with the other character data types, the text data type stores non-Unicode data. The maximum number of characters in a text data type is 2GB. The varchar(max) specification represents large objects—just like the text data type. Microsoft will ultimately phase out the text data type in favor of varchar(max), as it also adds features for processing large objects that have similar syntax to character strings. For example, the UPDATE statement in SQL Server 2005, including the SSE edition, has been augmented to allow you to modify portions of column values declared with a varchar(max) data type.

## Unicode Character Data Types

The fixed-length, variable-length, and large object data types for Unicode characters are nchar, nvarchar, and ntext. When you designate a Unicode character constant, specify a preceding N, such as N'a Unicode character constant'. Recall that Unicode characters require 2 bytes per character as opposed to 1 byte per character for non-Unicode characters. Because the maximum number of bytes for Unicode and non-Unicode strings is the same, Unicode character strings can hold only half as many characters as non-Unicode character strings.

Designate a fixed-length Unicode data type with a syntax of nchar(n). The value of n can range from 1 through 4,000. The upper limit of 4,000 is the maximum number of characters that you can hold in a fixed-length Unicode string. A 4,000-character fixed-length Unicode data type value and an 8,000-character fixed-length non-Unicode data type are both 8,000 bytes long.

Specify variable-length Unicode strings with the nvarchar(n) syntax. Again, n can range from 1 through 4,000. The nvarchar data type also takes a max argument to specify large objects.

■ **Note** SQL Server Express uses the sysname data type to reference database object names. The sysname data type is a built-in user-defined data type based on nvarchar(128).

The ntext data type is for large objects with Unicode characters. As with the text data type, the ntext data type will eventually be dropped in favor of the nvarchar(max) data type for large objects. If you require a Unicode string of more than 4,000 characters, use an nvarchar(max) or ntext data type. As with the text data type, large objects made up of Unicode characters can have a maximum byte length of 2GB. However, because each Unicode character takes twice as many bytes as non-Unicode characters, you can hold up to 1GB characters in either an `ntext` or `nvarchar(max)` data type versus 2GB characters in either a `text` or `varchar(max)` data type.

### Binary Strings

The fixed-length, variable-length, and large object binary string data types are `binary`, `varbinary`, and `image`.

- Use `binary(n)` to designate the fixed-length binary string data type for a column of binary strings. The `n` parameter can assume values from 1 through 8,000.
- Use `varbinary(n)` to specify a variable length binary string data type. Values for `n` can range from 1 through 8,000.
- Use `image` or `varbinary(max)` to specify a data type for a large object binary string with `image` or `varbinary(max)`. Because of the future obsolescence of the `image` data type (along with the `text` and `ntext` data types), use `varbinary(max)` for all new database applications.

## Miscellaneous

The remaining SQL Server data types serve a variety of specialized purposes. Unless your database application has special needs, you may find no use for these data types.

### timestamp

The `timestamp` data type is a good example of the specialized nature of the remaining data types. You can use this data type for timestamping table rows for an insert or identifying the last update. Instead of assigning a `datetime` value to a row, the `timestamp` data type assigns a sequential `binary(8)` number that is unique throughout a database. `rowversion` is another name for the `timestamp` data type. Eventually, Microsoft will make the `timestamp` data type obsolete in favor of the `rowversion` data type name because `rowversion` is a more accurate reflection of the data type's role. In addition, the `rowversion` name is more consistent with industry standards for SQL data types. The "Assigning the Current User, Date, and Time" section demonstrates the use of the `timestamp` data type and discusses how to convert from a `timestamp` data type to a `rowversion` data type.

### uniqueidentifier

The `uniqueidentifier` data type holds globally unique identifier (GUID) values, which have a 16-byte length. It is common to express `uniqueidentifier` values with a 32-character hexadecimal representation (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx); two hexadecimal characters represent 1 byte. GUID values are unique in space and time. As a result, `uniqueidentifier` values are sometimes used instead of the `IDENTITY` property for a column to denote the primary key for tables in a replicated database because the GUIDs will be unique across all replicated copies no matter how geographically dispersed the replicas are.

You can assign either the `NEWID` function or the `NEWSEQUENTIAL` function to the `DEFAULT` setting for a table column to automatically generate new `uniqueidentifier` values for successively inserted rows. See the "Managing Data Integrity with Basic Constraints and Column Properties" section for commentary on and samples using the `DEFAULT` setting for a table. The `NEWID` function generates `uniqueidentifier` values in a random order. The `NEWSEQUENTIALID` function generates `uniqueidentifier` values in a sequential order. You can perform equality and inequality comparisons between two different `uniqueidentifier` values.

**Tip** You can assign the ROWGUIDCOL property to a column declaration with a uniqueidentifier data type to specify a column of uniqueidentifier values. By adding the ROWGUIDCOL property to a column declaration, you gain the ability to return uniqueidentifier values in a result set by including the ROWGUIDCOL keyword in the list for a SELECT statement.

The uniqueidentifier data type is an advanced data type that the typical reader of this book should not use. It is very resource intensive because of its length. The introduction of the NEWSEQUENTIALID function with SQL Server 2005 makes uniqueidentifiers more comparable than in previous SQL Server versions to IDENTITY values as primary keys and index values generally. Nevertheless, there is still a substantial performance penalty associated with uniqueidentifier values relative to data types used with an IDENTITY property. You also do not have the same richness of processing options available for a column with a ROWGUIDCOL property that you do for a column with an IDENTITY property.

### cursor

The cursor data type represents a result set, such as one returned from a stored procedure, that you can scroll through. Set techniques that you implement with SELECT statements are generally a much more efficient way to recover values from a data source than for scrolling through result set rows with a cursor. My own preference is to reserve the use of cursors for Visual Basic 2005 code that manipulates the values in a result set derived with an SSE SELECT statement.

### sql_variant

The sql_variant data type can store values declared with data types, except for text, ntext, image, timestamp, sql_variant, and the max data types for varchar, nvarchar, and varbinary. The sql_variant data type is similar to the variant type in Visual Basic. An application can use it to store values when you are not sure what types of values a user will need to enter. For example, a sales person may benefit from a column into which can be entered miscellaneous data, such as a client's birth date, a spouse's name, and the price paid for a product in the last order. The availability of the new xml data type removes the need for the sql_variant data type to serve this purpose while providing a widely adopted standard format for exchanging data. For this reason, you should consider using an xml data type whenever a sql_variant data type may be appropriate.

### xml

The xml data type enables you to assign an XML document or XML fragment as a value for a column in a table's row, a variable, or a parameter. An XML document consists of a single root element with one or more nested other elements. An XML fragment consists of one or more elements without an outer root element. You can assign values to an XML document or XML fragment with element values and attribute values. SQL Server Express can implicitly convert a character constant to XML and assign the converted value to a column, variable, or parameter declared with an xml data type.

xml data type values can be either typed or untyped. A typed xml data type value has one or more schemas. An untyped xml data type value does not have a schema. SQL Server Express automatically checks xml values to make sure they are well formed. A well-formed document follows XML document syntax. If an xml value is typed, then SSE checks the value assignments and structure of a document to make sure that they are valid relative to the schemas for the XML document.

**Note** As indicated next, XML can quickly become an advanced topic that has very little to do with traditional database development topics. The terms *schema, typed xml, well-formed,* and *valid* are special terms in the XML nomenclature. A well-formed XML document is one that follows all general XML syntax rules. A schema is an XML document that represents the structure of one or more other XML documents. A typed `xml` value is one that references a schema. A valid `xml` value is one that follows the rules of a specific schema.

XML can quickly become an advanced topic that requires knowledge of many topics that are tangential to database design and analysis, such as XML document syntax and XML schema design. To keep the book's focus on core database development topics, I do not deal with `xml` data types beyond this section.

The following script illustrates three attempts to define an `xml` data type value with a character constant. These examples demonstrate how you can assign ad hoc values to a variable with an `xml` data type.

- The first attempt, which succeeds, starts with a declaration for the `@xdata` local variable. Notice the variable has an `xml` data type. Then, the script assigns a character constant for an XML document to the local variable and displays the data. The character constant includes a root element and two `firstname` elements within the root.
- In the second attempt, the local variable is reused with a character constant representing an XML fragment. Notice there is no root tag, but there are two properly specified `firstname` elements.
- The third attempt does not succeed in assigning a character constant to a local variable declared with an `xml` data type. The problem is that the initial `firstname` element misses a closing tag.

```
DECLARE @xdata xml
SET @xdata = '<root><firstname>Rick</firstname>' +
    '<firstname>Virginia</firstname></root>'
SELECT @xdata AS 'xml data'

SET @xdata = '<firstname>Rick</firstname>' +
    '<firstname>Virginia</firstname>'
SELECT @xdata AS 'xml data'

SET @xdata = '<root><firstname>Rick' +
    '<firstname>Virginia</firstname></root>'
SELECT @xdata AS 'xml data'
```

The following listing shows the outcome of running the preceding script. You can see that the first two attempts succeed. The value of the `xml` data type varies depending on the syntax used in the character constant for the first two attempts. The SSE XML parser correctly detects a syntax error in the third assignment of a character constant to the local variable with an `xml` data type. The script generating this listing is available as Ch04Xml.sql.

```
xml data
---------------------------------------------------------------------
<root><firstname>Rick</firstname><firstname>Virginia</firstname></root>

(1 row(s) affected)

xml data
---------------------------------------------------------------------
```

```
<firstname>Rick</firstname><firstname>Virginia</firstname>

(1 row(s) affected)


Msg 9436, Level 16, State 1, Line 10
XML parsing: line 1, character 59, end tag does not match start tag
```

## table

The `table` data type can represent a result set from a `SELECT` statement. You can return a `table` value from a table-valued, user-defined function. The returned `table` value can be referenced by the `FROM` clause of a `SELECT` statement in a script or a stored procedure. `table` values can be assigned to local variables. The `table` data type also offers an alternative to using temporary tables in stored procedures. Using `table` values in a stored procedure requires fewer stored procedure recompilations than when a stored procedure uses temporary tables. Most likely, the most common way you will use the `table` data type will be with user-defined functions that return a table. The "Creating and Using IF User-defined Functions" section in Chapter 7 includes samples illustrating the use of the `table` data type.

## alias

The `alias` data type corresponds to what was called a user-defined type in earlier versions of SQL Server. However, the introduction of user-defined types based on classes for the embedded CLR in SSE caused the introduction of the `alias` term for referring to traditional user-defined data types. The `alias` data type is suitable for use in defining table columns as well as for variables and parameters. You can create an `alias` type based on any of the other built-in SQL Server data types and further constrain the `alias` type by specifying any characteristics of the base data type as well as the nullability of the new `alias` data type. Recall that the `sysname` data type is really an `alias` of `nvarchar` with a value of `n` equal to 128. The `sysname` data type does not permit null values.

■**Note** The new user-defined data type based on the embedded CLR is outside the scope of this book because of its advanced programming concepts and its indirect relevance to traditional database development topics.

You can create a new `alias` type with either the

- `sp_addtype` system-stored procedure
- `CREATE TYPE` statement introduced with SQL Server 2005

The `CREATE TYPE` statement facilitates the creation of `alias` data types as well as the new embedded CLR-based user-defined type. The `sp_addtype` will become obsolete in a future version of SQL Server. Therefore, use the `CREATE TYPE` whenever possible. `alias` data types apply to only the database in which you create them unless you add an `alias` data type to the model database. Then, the `alias` data type is available for all new databases created after the addition of the `alias` data type to the model database.

## Creating Tables and Using Data Types

The review of data types indicates the role that data types can play in adding structure to a table. In a very real sense, specifying a data type for a table's column helps to define the table. A table column defined with a `tinyint` data type cannot hold any negative values nor any positive values greater than 255. Similarly, a table column defined with an `int` data type cannot store names based on the letters of the alphabet nor even numbers with digits after a decimal point.

Before you can add a column to a table, you must first create a table. You can add a new table to a database with the `CREATE TABLE` statement. Within the `CREATE TABLE` statement, there are many specifications that you can add, such as

- Data types associated with columns
- Column property assignments
- Constraints associated with individual columns
- Constraints associated with more than one column
- Constraints that tie one table to another

---

■**Note** SQL Server Express requires you to have permission to invoke a capability, such as run a `CREATE TABLE` statement, before you can invoke the capability. Anyone who can connect to an SSE instance as a sysadmin member, for example, a computer's administrator, can invoke the `CREATE TABLE` statement. In addition, this security feature (availability to sysadmin members) is true for nearly all other SSE capabilities. SQL Server Express also offers much more granular control over the permission to invoke its capabilities. Permissions are part of SSE security, which is the topic for Chapter 8.

---

This section briefly introduces the `CREATE TABLE` statement to demonstrate the role of a table as a container for columns with data types. As mentioned, data type specifications for columns add structure to a table. In addition, tables can constrain how data types work within a table. This section highlights the interplay between several data types when they are in a table.

### Creating a Table with Columns

In its most basic form, the `CREATE TABLE` statement can act as a container for a set of column specifications delimited by commas. Follow the `CREATE TABLE` keywords with a name for the table. A table's name must be unique throughout a database. In addition, table names must follow the rules for SQL Server identifiers (see SQL Server Identifier Rules in Chapter 3). You can optionally qualify a table name with a database name for the database containing the table and a schema denoting ownership for the table. Chapter 8 covers schemas along with other SQL Server security topics.

Place the column specifications for a table inside parentheses after the table's name. At a minimum, each column must have a name and a data type (with the exception of timestamp columns, which don't have to have a name). As with table names, column names must follow the rules for SQL Server identifiers. SQL Server Express automatically assigns the name `timestamp` to a column with a `timestamp` data type, but you can override this default name with a custom one. This section presents a series of samples demonstrating how to use the `CREATE TABLE` statement to achieve different results.

## Specifying Fixed-Width Data Types for Table Columns

You can essentially add an infinite number of tables to a database (the actual upper limit is 2GB). Within any table, you can designate up to 1KB columns. This number of columns is more than sufficient for the overwhelming majority of tables that you are likely to build. However, the maximum width for rows with exclusively fixed-width data types, such as `tinyint`, `int`, `real`, and `char`, is 8,060 bytes. Actually, because SSE reserves 7 bytes for overhead, the upper limit for data within a row is 8,053 bytes. This limit is based on the way SSE organizes data on storage devices; it ignores the upper limit on row width when a row contains one or more columns with a variable-length data type, such as `varchar`, `text`, or `varchar(max)`.

■**Note** The width of a column with a `char` data type can change before you run a `CREATE TABLE` statement. Once you run a `CREATE TABLE` statement including a column with a `char` data type, all column values in any row of the table will have the same width specified in the `CREATE TABLE` statement.

The following script shows the syntax for creating a table named `T`. The table's name appears immediately after the `CREATE TABLE` keywords. The table has three columns named `c1`, `c2`, and `c3`. The data types for the columns are `int` for `c1` and `char` for both `c2` and `c3`. After the first and second column specifications, a comma delimits the column setting for the designation of a new column. A trailing comma is not necessary for the third column because there are no subsequent column specifications.

```
CREATE TABLE T (
    c1 int,
    c2 char(49),
    c3 char(8000)
)
```

The `T` table illustrates how easy it is to reach the upper limit for a row's width. The `char` data type consumes 1 byte per character. The column widths for `c2` and `c3` total to 8,049 bytes. An `int` data type has a width of 4 bytes, which when added to 8,049 equals the upper limit (8,053 plus 7 bytes for overhead) for the width of a row exclusively composed of fixed-width data types.

As with many data definition statements, it is good practice to make sure that the name for a new object is unique. If the name for a new object is not unique, its data definition statement, such as `CREATE TABLE`, will fail. When you are initially designing a table, it is often convenient to remove any previously existing table with the name of the new table. This is because you will probably be iteratively refining the table's design and you will probably have sample data that's relatively easy to insert into the table (or even no data). Another strategy is to rename the old table so that you can preserve, and later recover, its data.

The following sample code merely drops any existing table with the name of the new table. You need to run the code before the preceding `CREATE TABLE` statement. For the sake of clarity, the code listing tests for a table's existence before repeating the `CREATE TABLE` statement that implements the table's design. The `EXISTS` keyword takes a `SELECT` argument that searches for a table named `T`. If the table exists, the code invokes a `DROP TABLE` statement for the prior version of the table before creating a new table named `T`. The following script is available from Ch04IntChar.sql.

**■Note** The sample files from this point on in the chapter reference use, via a `USE` statement, the `ProSSEAppsCh04` database. You can create this database with a statement as simple as `CREATE DATABASE ProSSEAppsCh04`. Alternatively, you can add and drop tables from any existing database that you prefer to use instead of the `ProSSEAppsCh04` database. Either choice will allow the code samples listed in the book to work as described. However, if you want to run the downloaded code sample files, then you must create `ProSSEAppsCh04` or modify the `USE` statement in the sample files so that they point at another database which you prefer to use instead of `ProSSEAppsCh04`.

```
IF EXISTS(SELECT name FROM sys.tables
    WHERE name = 'T')
    DROP TABLE T

CREATE TABLE T (
    c1 int,
    c2 char(49),
    c3 char(8000)
)
```

What if your application requirements change, and you need another character in the `c2` column without decreasing the width of the `c3` column? Well, you could increase the number of characters for `c2` from 49 to 50, but this would cause the `CREATE TABLE` statement to fail because the row size exceeds the maximum allowable width. By changing the data type for `c1` from `int` to `smallint`, you can reduce the row size enough to accommodate the extra character for `c2` plus one more character.

## Specifying Variable-Width Data Types for Table Columns

As mentioned in the "Character Data Types" section, SSE processes `char` data type values faster than `varchar` data type values, but the `varchar` data type is more flexible. One area of flexibility pertains to maximum column widths. When a row contains at least one column with a variable-width data type, such as `varchar`, then SSE does not enforce the 8,060-byte limit. If you are working with tables that can have substantial variability in the number of characters across rows, the `varchar` data type is definitely preferable to the `char` data type in spite of the speed advantage for the `char` data type.

The following script, which is available as Ch04IntVarchar.sql, shows a `CREATE TABLE` statement with the same general design as the preceding one. Two key differences are the swapping of the `varchar` data type for the `char` data type for columns `c2` and `c3`. In addition, the row width is 8,064 (8,000 + 60 + 4) plus 7 more bytes for overhead. Although this row width exceeds the 8,060-byte limit for rows with fixed-length data types, using `varchar` data types for `c2` and `c3` allows the row to exceed the limit without generating an error that blocks the creation of the table.

```
IF EXISTS(SELECT name FROM sys.tables
    WHERE name = 'T')
    DROP TABLE T

CREATE TABLE T (
    c1 int,
    c3 varchar(60),
    c2 varchar(8000)
)
```

### Specifying Table Columns with Unicode Data Types

Data types specifying Unicode characters follow the same rules about row widths as single-byte characters. Of course, Unicode characters require two bytes of storage for each character. Therefore, you can hold just about half as many characters per row when you are using exclusively fixed-width data types. The next script in the Ch04IntNchar.sql file implements the initial design for table `T` with `nchar` instead of `char` data types for columns `c2` and `c3`. Notice how the width for `c3` is set to 4,000 characters. This number of Unicode characters requires 8,000 bytes of storage. This leaves just 53 bytes for the remaining two columns. The `c2` column specification of `nchar(24)` consumes an additional 48 bytes of the remaining 53 bytes. This leaves 5 remaining bytes, which is more than sufficient for the `int` specification for `c1`.

```
IF EXISTS(SELECT name FROM sys.tables
    WHERE name = 'T')
    DROP TABLE T

CREATE TABLE T (
    c1 int,
    c2 nchar(24),
    c3 nchar(4000)
)
```

## Adding Data to Tables

After you design a table, it is natural to want to add data to it. This section presents a couple of approaches for inserting rows into a table. The basics of data manipulation, including inserting rows, are too important to cover from just one perspective. Therefore, data manipulation is revisited in Chapters 5 and 6.

- A couple of row insertion samples in this section highlight the `varchar` data type, which is a popular one for storing character data.
- This section also demonstrates a basic technique for recovering the data from an old version of a table even as you redesign the actual structure of the table. This capability to recover data from an old table version is especially important if you have important data in the table that you cannot just drop from a database.

### Inserting Very Wide Character Data into a Varchar Column

Both the `char` and `varchar` data types let you enter values into columns that are up to 8,000 characters in width. However, what happens if you need more than 8,000 characters in a single column? This requirement can exist for some comment fields in a database application as well as for storing messages or documents in a database. This section illustrates a basic technique for populating a column with a `varchar` data type. You also learn a solution based on the `varchar` data type that allows the insertion of more than 8,000 characters as a column value for a row in a table.

The following script, which is the top part of the listing in Ch04InsertToVarchar.sql, shows a new specification for the `T` table with just two columns, `c1` and `c2`. The `varchar` setting for `c2` designates a width of 8,000 characters. This is the maximum discrete number of characters that you can specify without referencing `max`. Recall that designating `max` as a width for a `varchar` data type allows up to 2GB of characters in a single column.

The sample script contains two `GO` statements—one after conditionally dropping `T` and another after creating `T`. The `GO` statement can force the execution of a batch of T-SQL statements. The use of a `GO` statement is especially useful when some downstream statements require the completed execution of some upstream statements.

**■Note** `GO` is not a T-SQL statement itself. Instead, `GO` is an instruction to a query tool, such as `sqlcmd` or the graphical query tool for SSE based on SQL Server Management Studio, which you use to submit T-SQL statements for processing by SSE.

```
IF EXISTS(SELECT name FROM sys.tables
    WHERE name = 'T')
    DROP TABLE T
GO

CREATE TABLE T (
    c1 int,
    c2 varchar(8000)
)
GO
```

The bottom portion of the script in Ch04InsertToVarchar.sql attempts to insert two rows into the table (see the following code). The code declares a local variable, `@v1`, with a `varchar(max)` specification to hold a value for insertion into the `c2` column in `T`. The script uses the `@v1` local variable to attempt the insertion of two rows into `T`. When using this format for an `INSERT` statement, the correct syntax is to list values for all the columns in a table, which is `T` in this case. You list the column values for a row after the `VALUES` keyword in the order that they appear in the `CREATE TABLE` statement.

- The first `@v1` value consists of 7,999 instances of A, generated with the `REPLICATE` function followed by a single instance of B. A concatenation operator (+) appends B to the end of the 7,999 instances of A. The attempt to insert a row with the first value of `@v1` succeeds.
- The second `@v1` value consists of the prior `@v1` instance value with another B concatenated to the end for a total of 8,001 characters. The attempt to insert a row with the second value of `@v1` fails because the `@v1` local variable is too wide to fit in the `c2` column of table `T`.

```
DECLARE @v1 varchar(max)

SET @v1 = REPLICATE('A',7999) + 'B'
INSERT T VALUES (1, @v1)
SELECT RIGHT(c2,2) 'Right 2 of c2' FROM T

SET @v1 = @v1 + 'B'
INSERT T VALUES (2, @v1)
SELECT RIGHT(c2,2) 'Right 2 of c2' FROM T
```

The listing from the `SELECT` statements after the two attempted inserts appears next. Notice the right two characters of the first row in `T` are A followed by B. The error message for the second attempt to insert a row explains that a column value must be truncated to fit in the table. This is because a value with 8,001 characters, the second `@v1` instance, does not fit into a column specified for a maximum of 8,000 characters. Therefore, the last `SELECT` statement in the preceding script invokes the `RIGHT` function for the last valid c2 column value.

```
(1 row(s) affected)

Right 2 of c2
-------------
AB
(1 row(s) affected)
```

```
Msg 8152, Level 16, State 10, Line 9
String or binary data would be truncated.
The statement has been terminated.
Right 2 of c2
-------------
AB

(1 row(s) affected)
```

To accommodate the second row of input, the c2 column in table T needs a new data type specification. In particular, change varchar(8000) to varchar(max). This simple redesign of the table permits its c2 column to accept up to 2GB of characters. The following script from the top portion of Ch04InsertToVarcharMax.sql shows a CREATE TABLE statement with the modification for c2.

```
IF EXISTS(SELECT name FROM sys.tables
    WHERE name = 'T')
    DROP TABLE T
GO

CREATE TABLE T (
    c1 int,
    c2 varchar(max)
)
GO
```

The bottom portion of Ch04InsertToVarcharMax.sql attempts the same two inserts described for the earlier version of table T. However, in this case, both inserts succeed. The following listing shows the results. Notice, in particular, the two row values returned by the second SELECT statement. The two rightmost characters are AB for the first row and BB for the second row. The outcome for the second row reflects the successive concatenation on two occasions of B to a string starting out with 7,999 instances of A.

```
(1 row(s) affected)

Right 2 of c2
---------------------
AB

(1 row(s) affected)


(1 row(s) affected)


Right 2 of c2
---------------------
AB
BB

(2 row(s) affected)
```

## Recovering Values from an Old Table for a New Table

All the prior table-creation samples dropped a previously existing version of a table with the same name as the new one to be created. This step ensures that a CREATE TABLE statement will not fail because of a previously existing table with the same name. However, dropping the table throws the data away along with the old design. What if you want the old data in a newly designed table? The sample in this section illustrates one approach to saving and reusing the data from an old version of a table in a newly designed table.

This sample creates yet another version of table T. The sample assumes that the old version of table T has some data that you want to reuse even if you have to redesign the table. In this case, the specification for c1 changes so that its data type has to be revised from int to bigint. This kind of change is common as the number of rows in a table grows beyond an original expectation. The sample in this section runs immediately after the script in Ch04InsertToVarcharMax.sql, which populates table T with a known set of values.

The following script from the top of Ch04CreateRecover.sql starts by checking if a table named T exists already. If there is a prior version of the table, the procedure uses the sp_rename system-stored procedure to rename the prior version to T_old. The sp_rename system-stored procedure cannot rename an object to an object that already exists in the database. Therefore, the script attempts to drop the T_old table with a DROP TABLE statement before invoking sp_rename. If the T_old table is not already there, the DROP TABLE statement generates an error message, but the rest of the script runs successfully. The CREATE TABLE statement illustrates the syntax for assigning the bigint data type to the c1 column.

```
IF EXISTS(SELECT name FROM sys.tables
    WHERE name = 'T')
    BEGIN
        PRINT 'T already.'
        DROP TABLE T_old
        EXEC sp_rename 'T', 'T_old'
    END
ELSE PRINT 'No T already.'

CREATE TABLE T (
    c1 bigint,
    c2 nvarchar(max)
)
```

The most significant advantage of the preceding script is that it saves the data from the prior version of table T in a new version of the T_old table. After the preceding CREATE TABLE statement creates a new version of table T, the table has no data. An INSERT statement for table T that selects rows from the T_old table can populate the new version of the T table with the data from the prior version of the T table. The following script from Ch04CreateRecover.sql illustrates how simple the syntax can be to accomplish this task. The script also includes a SELECT statement to confirm the copying of the row values.

```
INSERT T
SELECT * FROM T_old

SELECT c1, RIGHT(C2,2) 'Right 2 of c2' FROM T
```

The following listing confirms the operation of the code from Ch04CreateRecover.sql. As you can see, the script detected a prior version of T, and the sp_rename system-stored procedure issued a precaution about the impact of changing an object name that may have other objects dependent on it. Because we are creating a replacement version of table T, and our modification is very minor, the precaution does not apply in this case. The listing ends by showing the two rightmost characters from the first and second rows of the data copied to the new version of table T.

```
T already.
Caution: Changing any part of an object name could break
scripts and stored procedures.

(2 row(s) affected)
```

```
c1                   Right 2 of c2
-------------------- --------------
1                    AB
2                    BB

(2 row(s) affected)
```

**Note** Another approach to modifying a table is to invoke the `ALTER TABLE` statement. Although the `ALTER TABLE` statement does facilitate many table modifications, it is not as flexible as the `CREATE TABLE` statement. In addition, you will always need the `CREATE TABLE` statement whenever you need to make a new table from scratch. When you are just starting out with T-SQL, there are many statements as well as syntactical and semantic issues to learn. My recommendation is that beginners should master the essential, general-purpose statements, such as `CREATE TABLE`, before investing their time in more specialized statements, such as `ALTER TABLE`. However, as your skills and experience grow, I strongly recommend that you experiment with the `ALTER TABLE` statement.

## Managing Data Integrity with Basic Constraints and Column Properties

Just as you can manage the data in a table by setting the data type for a column within a `CREATE TABLE` statement, you can also set column properties and basic constraints that help you control the data entering a table. This section introduces you to a popular subset of the constraints and column properties that you can apply to individual columns to manage the integrity of the data in a table.

- You can add a constraint to a column that determines whether SSE will reject new rows with a missing value for the column (by default, SSE accepts rows with missing data for any column).
- You can use the `DEFAULT` column property to designate a value to enter when no input is specified for a column.
- Designating a table's primary key constrains a column to reject null values, and a primary key designates columns that uniquely identify the rows in a table.
- The `IDENTITY` property can interact with the `PRIMARY KEY` phrase to simplify the setting of primary key values.

### Inserting Data for a Subset of Table Columns

When you enter a new row into a table, you may not always have values available for every column in the row. Columns in a new row with no values at input time typically have null values; this is another way of saying the values for those columns are unknown or missing. Null values behave differently than other column values. The sample code for this section illustrates how you can generate null values and demonstrates some special handling procedures for null column values.

The following excerpt from the Ch04InsertWithMissingValues.sql file creates a table with five columns. The first four columns specify in order `int`, `bit`, `varchar`, and `dec` data types. The last column is a computed column whose value depends on the sum of the first and second columns with `int` and `bit` data types.

```
CREATE TABLE T (
    int1 int,
    bit1 bit,
    varchar1 varchar(3),
    dec1 dec(5,2),
    cmp1 AS (int1 + bit1)
)
```

You can think of a computed column as a virtual column that is not physically a part of the table. Unless you explicitly persist a computed column (with the PERSISTED keyword), SSE stores just the expression for the computed column values and automatically computes values for the column whenever they are needed. However, a persisted computed column is stored as actual column values that are updated when any of the inputs to an expression are revised.

A computed column value will be null if one or more of its inputs are null. In addition, arithmetic overflows or underflows can generate null values for a computed column even when none of its inputs are null. An overflow is an outcome that is larger than the biggest number that a data type can represent. Similarly, an underflow is smaller, or more negative, than a data type can denote.

---

■ **Tip** You can ensure that a computed column never results in a null value by using the ISNULL function, which returns a constant instead of a null value. The general syntax for such an expression is ISNULL(computedcolumnexpression, constant).

---

The following excerpt from Ch04InsertWithMissingValues.sql inserts four rows into the T table resulting from the immediately preceding CREATE TABLE statement. Although there are five columns in the table, each INSERT statement specifies values for just two columns. Furthermore, the designated columns change from row to row. By the way, you never insert values for a computed column, such as the cmp1 column in table T, because its value is computed from other values. Notice that the INSERT statement, when used with the following format, has column names in parentheses immediately after the INSERT keyword. Values in a second set of parentheses after the VALUES keyword designate values for the new row in the order of the column names in the first set of parentheses.

```
INSERT T (int1, bit1) VALUES (1, 0)
INSERT T (int1, varchar1) VALUES (2, 'abc')
INSERT T (int1, dec1) VALUES (3, 5.25)
INSERT T (bit1, dec1) VALUES (1, 9.75)
```

So, the first INSERT statement inserts 1 into int1 and 0 into bit1, because of the order of the column names in the first set of parentheses and the order of values in the second set of parentheses.

The next excerpt from Ch04InsertWithMissingValues.sql shows some ways of extracting and processing table values some of which may be null.

- The first SELECT statement merely lists all column values for each row in the table. This will clearly show which column values have known values versus null values.
- The next SELECT statement counts all the rows in the T table.
- The third SELECT statement invokes the COUNT function for the values in the int1 column. Because aggregate functions, such as COUNT, do not process null values, the count reflects just rows with known values.
- Although the third SELECT statement *implicitly* filters null values with the COUNT function, the fourth SELECT statement *explicitly* filters null values as it counts just the rows with known values for the bit1 column. The IS NOT NULL phrase in the WHERE clause references exclusively known values.

- The fifth `SELECT` statement counts just the rows with null values for the `bit1` column. The `IS NULL` phrase in the `WHERE` clause designates just unknown or missing values.
- The last `SELECT` statement invokes the `AVG` function to compute the average of values in the `dec1` column. Because `AVG` is an aggregate function, it automatically ignores null vales. However, the `AVG` function can cast its output with different precision and scale settings than its input. The list in the `SELECT` statement applies the `CAST` function to ensure a return value with the same precision and scale settings for the `dec1` column.

```
--All columns for all rows
SELECT * FROM T
GO

--Count all rows
SELECT COUNT(*) 'Rows in T'
FROM T

--Count int1 values (implicitly non-null)
SELECT COUNT(int1) 'int1 values in T'
FROM T

--Count non-null bit1 values
SELECT COUNT(*) 'Count of non-null bit1'
FROM T
WHERE bit1 IS NOT NULL

--Count null bit1 values
SELECT COUNT(*) 'Count of null bit1'
FROM T
WHERE bit1 IS NULL

--Average of dec1 values
SELECT CAST(AVG(dec1) AS dec(5,2)) 'Avg of dec1'
FROM T
WHERE dec1 IS NOT NULL
```

Running Ch04InsertWithMissingValues.sql generates a result listing like the following one.

- The first result set clearly identifies the null column values with the `NULL` keyword for each row within the `T` table.
- The second result set returns the count of all the rows in `T`.
- The third result set shows how many of the rows in `T` have a non-null `int1` value.
- The fourth and fifth result sets display, respectively, the number of non-null and null column values in the `bit1` column of `T`.
- The sixth result set presents the average across the non-null values in the `dec1` column.

```
int1        bit1  varchar1 dec1                                      cmp1
----------- ----- -------- --------------------------------------- -----------
1           0     NULL     NULL                                      1
2           NULL  abc      NULL                                      NULL
3           NULL  NULL     5.25                                      NULL
NULL        1     NULL     9.75                                      NULL

Rows in T
-----------
```

```
4

int1 values in T
----------------
3
Warning: Null value is eliminated by an aggregate or other SET operation.

Count of non-null bit1
----------------------
2

Count of null bit1
------------------
2

Avg of dec1
-----------
7.50
```

## Not Allowing Null Values in a Column

As the preceding sample demonstrates, null values are permitted by default for columns with `int`, `bit`, `varchar`, and `dec` data types. The same default behavior applies to most other data types that do not have special settings to force the population of a column value. The `timestamp` data type is one exception that requires no special settings to avoid the possibility of a null value.

Some database applications require a known value for a column. For example, you may require users to say whether they smoke or not on a questionnaire for health insurance. Failing to answer the question can make it impossible to compute a life insurance rate estimate. In this kind of situation, you need a way to constrain the values within a column so that null values are not permitted.

The `NOT NULL` phrase on the line declaring a column in a `CREATE TABLE` statement causes a column's default behavior with respect to null values to change. When a column declaration includes `NOT NULL`, SSE rejects attempts to insert rows that have unknown values for the column. Inserts succeed when they specify a known value in the legitimate range for a column's data type.

The following excerpt from the Ch04NoBit1Nulls.sql file shows the syntax for specifying the `NOT NULL` constraint for a column with a `bit` data type. The same syntax applies to other data types. If you look back at the preceding sample, you'll recognize the `CREATE TABLE` statement as the same, except for the `NOT NULL` phrase on the `bit1` column specification. Because of the `NOT NULL` constraint in the `bit1` column specification, only `INSERT` statements with a `bit1` column value of 0 or 1 succeed.

```
CREATE TABLE T (
    int1 int,
    bit1 bit NOT NULL,
    varchar1 varchar(3),
    dec1 dec(5,2),
    cmp1 AS (int1 + bit1)
)
```

The T-SQL code in Ch04NoBit1Nulls.sql is nearly identical to the code in Ch04InsertWithMissingValues.sql from the preceding sample. For example, the `INSERT` statements are exactly the same in both files. The two differences are the inclusion of the `NOT NULL` constraint for the `bit1` column and the exclusion of all `SELECT` statements except for the first one in the Ch04NoBit1Nulls.sql. The first `SELECT` statement lists the column values for all rows in the table.

The following excerpt from the listing for running the script inCh04NoBit1Nulls.sql shows the rows that succeeded in entering table T. Notice that there are just two rows, although there are four INSERT statements (see the INSERT statements from the preceding sample). Each row of output includes a known value (0 or 1) for a bit data type in the bit1 column. The two omitted rows correspond to INSERT statements that did not specify a value for the bit1 column. This result confirms the impact of the NOT NULL constraint for the bit1 column in the preceding CREATE TABLE listing.

```
int1        bit1  varchar1 dec1 cmp1
----------- ----- -------- ---- ----
1           0     NULL     NULL 1
NULL        1     NULL     9.75 NULL
```

## Designating Default Column Values

You can use the DEFAULT keyword in a column declaration to designate a default value for the column. If an INSERT statement or an UPDATE statement fails to specify a value for the column, then SSE assigns the DEFAULT value, instead of a null value, to the column in a new row.

Using a DEFAULT column property assignment makes sense when a null value has no meaning for a column, such as a column to designate whether a contract is signed and available for review. You can represent this type of column with a bit data type that is either 0 for not signed or not available, but 1 for signed and available. A null value makes no sense for this type of column (unless you assume it is possible to not know if a contract is signed and available for review).

A DEFAULT constraint can specify

- A constant, such as a constant value equal to a number or a sequence of characters
- A function's return value, such as the GETDATE() function to return the current date and time
- A SQL-92 niladic function, which can supply either the current user's name or the current date and time
- NULL, which is not necessary because data column values are null by default

You specify a column's default value by following the DEFAULT keyword in the column's declaration with a constant, built-in function, a niladic function, or another keyword (NULL). The SQL-92 niladic CURRENT_TIMESTAMP function and the built-in GETDATE() function both return the current date and time. Four different SQL-92 niladic functions return the user account name for the current user: CURRENT_USER, SESSION_USER, SYSTEM_USER, and USER.

■**Note** You can also specify default values independently of column declarations and bind the default values to columns in one or more tables. This capability will be removed from future versions of SQL Server, including its SSE edition. Therefore, you should use the DEFAULT constraint whenever possible.

### Assign Zeroes Instead of Null Values

The Ch04NullToZero.sql file illustrates how to define a default value for a bit column that represents missing values as zeroes rather than the default of null values. The CREATE TABLE statement from the file shows the syntax for the DEFAULT keyword. There is no need for any parentheses after the DEFAULT keyword to designate the constant value (0).

```
CREATE TABLE T (
    int1 int,
    bit1 bit NOT NULL DEFAULT 0,
    varchar1 varchar(3),
    dec1 dec(5,2),
    cmp1 AS (int1 + bit1)
)
```

The script in NullToZero.sql includes the same four `INSERT` statements and the first `SELECT` statement from Ch04InsertWithMissingValues.sql. When you run these statements after the preceding `CREATE TABLE` statement, the following listing appears. All four rows appear in the output from the `SELECT` statement. The `DEFAULT 0` constraint for the `bit1` column in the `CREATE TABLE` statement assigns a value of 0 to rows with a missing value—namely, the second and third rows. The `NOT NULL` constraint is not strictly necessary because the `DEFAULT` constraint overrides it, but the inclusion of the `NOT NULL` phrase does not cause a compilation error.

■ **Note** If you have prior experience with earlier versions of SQL Server, you may be familiar with severity levels for errors. Error levels are reported after one or more batches of T-SQL statements run; the error level numbers tell you that something went wrong and how severe it is after the T-SQL code runs. The SSE query tool based on SQL Server Management Studio introduces the notion of a compilation error for T-SQL code. A compilation error is detected by the SSE query tool before batch statements run. The query tool marks lines with syntax or semantic errors. As noted, the `NOT NULL` and `DEFAULT 0` column properties for constraining `bit1` column values do not conflict with one another. However, replacing `NOT NULL` with `NULL` in the Ch04InsertWithMissingValues.sql file generates a compilation error because a column cannot have a default value of zero and allow null values. If you change `NOT NULL` to `NULL`, the SSE query tool marks the changed line.

```
int1        bit1  varchar1 dec1 cmp1
----------- ----- -------- ---- -----------
1           0     NULL     NULL 1
2           0     abc      NULL 2
3           0     NULL     5.25 3
NULL        1     NULL     9.75 NULL

Command(s) completed successfully.
```

## Assigning the Current User, Date, and Time

It is often desirable to track the user, date, and time associated with changes to the rows in a database. The following demonstration from WhoWhenDefaults.sql tracks who made the last change to a row and when the user either inserted or modified the row. The `DEFAULT` property used along with the `USER` and `CURRENT_TIMESTAMP` SQL-92 niladic functions can help you store this kind of information in a table. The demonstration also contrasts the `datetime` return value from the `CURRENT_TIMESTAMP` function with a `timestamp` data type value. Recall that the `timestamp` data type adds a `binary(8)` number to a row rather than `datetime` data type.

The following excerpt from WhoWhenDefaults.sql creates a table with five columns.

- The first two columns have `int` and `bit` data types along with constraints, borrowed from the preceding sample, for the `bit1` column.
- The declaration for the third column shows how to assign a name (`rvr1`) to a timestamp column. With the current declaration syntax, you can replace `timestamp` with `rowversion`. The designation of a name for a timestamp column is optional. If you do not explicitly specify a name, SSE uses the default name of `timestamp` for the column. Unlike the `timestamp` data type, the `rowversion` data type requires a column name in its declaration.
- The fourth column's declaration assigns the return value from the `USER` function to the `usr1` column. The `USER` function returns the name of the current user, which is a user account. SQL Server Express represents user account names with a `sysname` data type that you can store in a column with an `nvarchar(128)` data type.
- The last column's declaration stores the date and time for a change in a `datetime` format. The `CURRENT_TIMESTAMP` function returns this value.

```
CREATE TABLE T (
    int1 int,
    bit1 bit NOT NULL DEFAULT 0,
    rvr1 timestamp,
    usr1 nvarchar(128) DEFAULT USER,
    createtime datetime DEFAULT CURRENT_TIMESTAMP
)
```

The following excerpt from WhoWhenDefaults.sql inserts three rows into the `T` table at 1-second intervals. The two `WAITFOR` statements generate 1-second delays between the `INSERT` statements. Without the delays, it is possible for SSE to insert more than one row within a single `datetime` instance. You will not normally program delays when inserting rows, but using them in this context helps to clarify the values returned by the `CURRENT_TIMESTAMP` function.

```
INSERT T (int1) VALUES (1)
WAITFOR DELAY '00:00:01'
INSERT T (int1, bit1) VALUES (2, 0)
WAITFOR DELAY '00:00:01'
INSERT T (int1, bit1) VALUES (3, 1)
```

The last code excerpt from WhoWhenDefaults.sql in the following code listing starts by listing the values in `T` generated by the preceding `INSERT` statements with a `SELECT` statement. The fourth item in the `SELECT` list uses a `CONVERT` function to represent the `binary(8)` value generated by the `timestamp` data type as an integer. Next, an `UPDATE` statement revises the `bit1` column value to 1 for the row with an `int1` column value of 2 (this is the second row). This `bit1` column value is set to 0 by the preceding codeblock. The concluding `SELECT` statement lists the `T` values after the revision of the `bit1` column value.

```
SELECT int1, bit1, usr1,
    CONVERT(int, rvr1) 'Timestamp as int',
    createtime
FROM T
GO

UPDATE T
set bit1 = 1
WHERE int1 = 2
GO
```

```
SELECT int1, bit1, usr1,
    CONVERT(int, rvr1) 'Timestamp as int',
    createtime
FROM T
GO
```

An excerpt from the result listing for running the script in WhoWhenDefaults.sql appears next.

- SQL Server Express generates the first result set from the first `SELECT` statement immediately after the `INSERT` statements.
    - Notice that the `timestamp` values increase by a single integer value from one row to the next. The actual values start at 3104 and end at 3106.
    - In contrast, the `createtime` column, which results from the `CURRENT_TIMESTAMP` function, increases by about 1 second for each successive row.
    - The `usr1` column values reflect the return value from the `USER` function. I ran the script as the `dbo` user.
- SQL Server Express generates the second result set after invoking the `UPDATE` statement, which changes the `bit1` column value for the row with an `int1` value of 2.
    - Notice that this second result set has a new value for the `timestamp` column as well as the `bit1` column in the second row. The new `timestamp` column value is 3107.
    - Also, notice that the `createtime` column value does not change. This distinction between the `timestamp` and `createtime` columns results from the fact that the `CURRENT_TIMESTAMP` function in the `CREATE TABLE` statement only operates when SQL Server Express inserts a new row in the table, but the `timestamp` column receives a new value whenever a row is initially inserted as well as for each subsequent row update.

```
int1  bit1  usr1  Timestamp as int createtime
----- ----- ----- ---------------- -----------------------
1     0     dbo   3104             2005-04-02 18:57:03.227
2     0     dbo   3105             2005-04-02 18:57:04.230
3     1     dbo   3106             2005-04-02 18:57:05.290
int1  bit1  usr1  Timestamp as int createtime
----- ----- ----- ---------------- -----------------------
1     0     dbo   3104             2005-04-02 18:57:03.227
2     1     dbo   3107             2005-04-02 18:57:04.230
3     1     dbo   3106             2005-04-02 18:57:05.290
```

## Designating a Column as a Primary Key

Primary key values provide a basis for uniquely identifying each row within a table. Use the `PRIMARY KEY` phrase in a column declaration to designate the column as the primary key for a table. A table can have only one primary key, which can be based on a single column or a set of columns. The values in a primary key column must

- Be unique for each row in the table
- Not be null for any row in the table

The primary key depends on an index created by SSE. This index can be clustered or nonclustered. Clustered indexes process faster than nonclustered indexes in the same way that columns with indexes process faster that columns without indexes. SQL Server Express permits 250 indexes per table. Only one table index can be clustered—that is, have rows arranged in storage according to the order of the index values. You can use the `CLUSTERED` keyword in a column declaration to make the index for a column clustered. However, the `PRIMARY KEY` phrase makes the index for a primary key clustered by default unless

- There is already a clustered index for another column in the table.
- You include the `NONCLUSTERED` keyword in the primary key's column declaration.

---

■**Note** The `PRIMARY KEY` phrase defines a constraint that restricts the values in the columns defining a primary key to conform to the rules of a primary key, which include uniquely defining a table's rows and not allowing null values. The `UNIQUE` constraint is another type of constraint that allows the specification of unique values across the rows of a table. `UNIQUE` constraint columns are independent of `PRIMARY KEY` constraint columns. In addition, a table can only have one `PRIMARY KEY` constraint, but it can have multiple `UNIQUE` constraints.

---

It is frequently recommended that you assign an integer data type to a primary key column. The smallest possible data type with enough distinct values for the rows in the table is the best choice for optimizing performance. Legitimate data types for a primary key column include `tinyint`, `smallint`, `int`, `bigint`, and `dec(p, 0)`. When you use an integer data type for a primary key column, you can further reduce the possibility for erroneous data by designating an `IDENTITY` property for the primary key column.

On the one hand, `IDENTITY` values are a relatively easy way of generating a unique value for every row in a table; recall that `IDENTITY` values are automatically generated. On the other hand, `IDENTITY` values have no structural relation to the other column values. If the natural primary key for a table is defined by multiple columns, you may frequently improve performance by using an `IDENTITY` column as the primary key instead.

- Some developers prefer the performance and automatic generation benefits of `IDENTITY` values.
- Other developers prefer to use meaningful column values to define a primary key even if the use of multiple columns negatively impacts performance.
- For a relatively small table of several hundred to a few thousand rows, performance differences rarely have a significant performance impact. As the rows in a table grow, performance tuning becomes more critical.

## Using the PRIMARY KEY Phrase

The following excerpt from the Ch04UsingPrimaryKey.sql file shows how simple it is to include a `PRIMARY KEY` phrase in a column declaration. No additional constraints apply to the `int1` column, but SSE treats the column as if it had both `NOT NULL` and `UNIQUE` constraints. The `bit1` column is the only other column in the table. This column has a `NOT NULL` and its `DEFAULT` property set.

```
CREATE TABLE T (
    int1 int PRIMARY KEY,
    bit1 bit NOT NULL DEFAULT 0
)
```

The next excerpt from Ch04UsingPrimaryKey.sql includes five `INSERT` statements for the table created with the preceding `CREATE TABLE` index.

- The first three INSERT statements are valid because they contain unique int1 column values.
- The fourth INSERT statement has a missing int1 column value. Because int1 serves as the primary key for the table, the missing value causes the rejection of the whole row.
- The fifth INSERT statement specifies an int1 column value of 3, but the third INSERT statement previously added a row to the table with this value. Therefore, SSE rejects the row for the fifth INSERT statement because it has a duplicate primary key value.

```
INSERT T (int1, bit1) VALUES (1, 1)
INSERT T (int1, bit1) VALUES (2, 0)
INSERT T (int1) VALUES (3)
INSERT T (bit1) VALUES (1)
INSERT T (int1, bit1) VALUES (3,1)
```

The following listing shows the result of running the script in Ch04UsingPrimaryKey.sql. A SELECT statement for all the column values in each row within the table created by the preceding CREATE TABLE statement generates the following output (SELECT * FROM T). The first three INSERT statements succeed. Before the result set for the SELECT statement, you can see error messages returned by SSE. These error messages explain why two of the INSERT statements did not succeed.

```
Msg 515, Level 16, State 2, Line 8
Cannot insert the value NULL into column 'int1', table
'ProSSEAPPSCh04.dbo.T'; column does not allow nulls.
INSERT fails.The statement has been terminated.
Msg 2627, Level 14, State 1, Line 9
Violation of PRIMARY KEY constraint 'PK__T__145C0A3F'.
Cannot insert duplicate key in object 'dbo.T'.
The statement has been terminated.
int1        bit1
----------- -----
1           1
2           0
3           0
```

## Using the IDENTITY Column Property

It is common to specify an IDENTITY property for a column with an integer data type that serves as a primary key. The IDENTITY property causes SSE to assign sequential column values starting from a seed value in fixed increments. Although the default seed and increment values for the IDENTITY keyword are both 1, you can override either or both. To change the default seed and increment settings, merely specify IDENTITY(seedvalue, incrementvalue) instead of IDENTITY.

When a column has an IDENTITY property setting, users do not have to specify a value for a column during the insertion of a new row because SSE automatically assigns a column value. If you attempt to insert a column value for a column with an IDENTITY property setting, SQL Server rejects the row by default. When you include an IDENTITY property setting for a column serving as the primary key, you do not have to worry about duplicate values because SQL Server automatically specifies sequentially unique values.

■ **Tip** The IDENTITY property applies only to columns that have an integer data type. You cannot use it with columns that have a char or varchar data type.

The following excerpt from Ch04UsingIDENTITY.sql demonstrates the syntax for specifying an `IDENTITY` property for the `int1` column. By not explicitly designating seed or increment values, the statement accepts the default value of 1 for both. The `int1` column also serves as the table's primary key, and it has an `int` data type. The `bit1` column has a `bit` data type with a `NOT NULL` constraint and its `DEFAULT` property set to 0.

```
CREATE TABLE T (
    int1 int IDENTITY PRIMARY KEY,
    bit1 bit NOT NULL DEFAULT 0
)
```

The next excerpt from Ch04UsingIDENTITY.sql shows three valid `INSERT` statements for the table generated by the `CREATE TABLE` in the sample file and one invalid `INSERT` statement.

- The first two statements merely assign values to the `bit1` column.
- The third statement shows the syntax for not specifying any column values. The `DEFAULT VALUES` phrase in the `INSERT` statement tells SSE to use the default value for `bit1`. This is legitimate in this case because
    - `int1` has an `IDENTITY` property setting, which causes its column values to be assigned by SSE.
    - `bit1` has its `DEFAULT` property set to 0, which does not conflict with the column's `NOT NULL` constraint.
- The fourth `INSERT` statement fails because it attempts to assign a value to `int1`, which has an `IDENTITY` property. SQL Server Express does not by default permit a client application to set a value for an `IDENTITY` column.

```
INSERT T (bit1) VALUES (1)
INSERT T (bit1) VALUES (0)
INSERT T DEFAULT VALUES
INSERT T (int1, bit1) VALUES (4,1)
```

The following listing shows the result of running the script in Ch04UsingIDENTITY.sql, which ends with `SELECT * FROM T`. Notice that the `int1` column values start at one and increment by one for each valid row through three. An error message before the result set explains the problem with the fourth `INSERT` statement: that while it is technically possible to insert a value to a column with an `IDENTITY` property, `IDENTITY_INSERT` must be set to `ON`. By default, `IDENTITY_INSERT` is set to `OFF`.

---

**■Tip** The syntax for turning `IDENTITY_INSERT` on is `SET IDENTITY_INSERT databasename.schemaname.tablename ON`. The syntax for restoring `IDENTITY_INSERT`'s default status is `SET IDENTITY_INSERT databasename.schemaname.tablename OFF`. Specifying `databasename` and `.schemaname` qualifiers for the `tablename` argument is optional. While `IDENTITY_INSERT` is `ON`, you enable the insertion of values to the `IDENTITY` column for only one table in a database—namely, the `tablename` argument value.

---

```
Msg 544, Level 16, State 1, Line 9
Cannot insert explicit value for identity column in
table 'T' when IDENTITY_INSERT is set to OFF.
int1        bit1
----------- -----
1           1
2           0
3           0
```

## Managing Data Integrity with Sophisticated Constraints

Constraining the values that can enter a table improves the ease of using a table by ensuring high-quality data in a table. For example, by not allowing nulls into some or all columns of a table, your queries for a table do not have to account for the existence of null values. In addition, a `PRIMARY KEY` constraint for a column guarantees that you cannot have duplicate values in that column of a table.

This section presents additional concepts and samples in three areas to help you manage the integrity of the data in your tables.

- First, you learn about `CHECK` constraints. This flexible type of constraint can set limits on what specific values and patterns for values are valid for a table.
- Second, this section presents the syntax and demonstrates solutions for creating constraints that are at the table level instead of being a part of the declaration for a single column. A table-level constraint allows you to manage your data across multiple columns within a table.
- The third and concluding topic for this section drills down on `FOREIGN KEY` constraints. Just as a primary key helps to manage data integrity for a single table, a foreign key facilitates the management of data integrity between a pair of tables.

### Using CHECK Constraints

When you write a column `CHECK` constraint for a specific column, you can limit the range of values and pattern of values for that column. The logic behind a `CHECK` constraint within a column is very simple.

- A Boolean expression specifies a condition for the insertion of values in the column.
- If the condition evaluates to `True`, the new value is valid.
- If the condition evaluates to `False`, the new value is invalid.

Any table can have multiple `CHECK` constraints. When the values in a row are valid for all constraints, SSE inserts the row in a table. If one or more values within a row fail to pass a constraint, SSE issues an error message describing the problem for the first failed constraint. The error message helps to guide users to refine their input so that it is valid for resubmitting to the table. Additional resubmittals may uncover other constraints for which the data are not valid.

There are two syntax conventions for adding a `CHECK` constraint to a column declaration in a `CREATE TABLE` statement. The easiest way to add a `CHECK` constraint is to follow the `CHECK` keyword with a Boolean expression. Insert the keyword and expression in the column declaration for which you want to constrain values. With this approach, SSE assigns a default name for the constraint that appears in error messages.

A second approach to declaring a `CHECK` constraint within a column allows you to assign a custom name for a constraint that will appear in error messages. A custom name will define the purpose for a constraint more precisely than the default name that SSE assigns. Custom constraint names can make it easier for users to determine how to fix their input so that it is valid. You can assign a custom name by preceding the `CHECK` keyword with `CONSTRAINT` followed by the custom name for the constraint.

## Blocking Empty and Missing Character Input

The Ch04CheckTooShort.sql file includes a script for verifying that `varchar` data type values do not contain an empty string (`''`). The `CREATE TABLE` statement in the script demonstrates the two syntax conventions for declaring a `CHECK` constraint within a column. The following `CREATE TABLE` statement from the file declares three columns.

- The first column serves as the primary key for the table. This column has an `int` data type with an `IDENTITY` property and a `PRIMARY KEY` constraint.
- The second column (`vch1`) illustrates the approach to defining a `CHECK` constraint where SQL Server Express assigns a default name to the constraint. The Boolean expression following the `CHECK` keyword uses the `LEN` function to specify that the number of characters in the `vch1` column must be greater than 0.
- The declaration for the third column (`vch2`) demonstrates the syntax for assigning a custom name to a constraint. The name for the `CHECK` constraint for the `vch2` column is `CK_LEN_TOO_SHORT`.

```
CREATE TABLE T (
    int1 int IDENTITY PRIMARY KEY,
    vch1 varchar(5)
        CHECK (LEN(vch1) > 0),
    vch2 varchar(5)
        CONSTRAINT CK_LEN_TOO_SHORT
        CHECK (LEN(vch2) > 0)
)
```

The following four `INSERT` statements from Ch04CheckTooShort.sql test the `CHECK` constraints in the preceding `CREATE TABLE` statement.

- The first statement inputs a row with valid `vch1` and `vch2` column values.
- The second and third statements input rows with invalid values for the `vch1` and `vch2` columns, respectively.
- The last `INSERT` statement highlights an issue that the `CHECK` constraints are not optimized to address, that is, the entry of rows with null values for `vch1` or `vch2`. If you want to block zero-length strings (`''`), chances are you also want to block null values. However, the `CHECK` constraints for the `vch1` and `vch2` columns do not block the input of null values.

```
INSERT T (vch1, vch2) VALUES('a','b')
INSERT T (vch1, vch2) VALUES('','b')
INSERT T (vch1, vch2) VALUES('a','')
INSERT T DEFAULT VALUES
```

An excerpt from the listing for running the script in Ch04CheckTooShort.sql appears next. The listing results from running `SELECT * FROM T` at the end of the script. Notice that SSE automatically assigns the name `CK_T_1B0907CE` to the `CHECK` constraint for the `vch1` column. This name by itself is not too informative. In contrast, the `CHECK` constraint in the `vch2` column declaration assigns a name (`CK_LEN_TOO_SHORT`) that conveys the meaning of the constraint (the length of the input is too short). An informative constraint name can make detecting and fixing erroneous input easier and faster.

```
Msg 547, Level 16, State 0, Line 3
The INSERT statement conflicted with the CHECK constraint
"CK__T__vch1__1B0907CE". The conflict occurred in database
"ProSSEAPPSCh04", table "dbo.T", column 'vch1'.
The statement has been terminated.
Msg 547, Level 16, State 0, Line 4
The INSERT statement conflicted with the CHECK constraint
"CK_LEN_TOO_SHORT". The conflict occurred in database
"ProSSEAPPSCh04", table "dbo.T", column 'vch2'.
The statement has been terminated.
(1 row(s) affected)

int1        vch1  vch2
----------- ----- -----
1           a     b
4           NULL  NULL
```

Notice from the preceding output that a CHECK constraint for a string need not have an impact on the input of null values (the second row contains two null values). If you want to block null values as well as zero-length strings (''), just insert the NOT NULL phrase in the declaration for a column. The following CREATE TABLE statement from Ch04CheckTooShortNotNull.sql illustrates the syntax for column declarations that block null values as well as zero-length strings ('').

```
CREATE TABLE T (
    int1 int IDENTITY PRIMARY KEY,
    vch1 varchar(5)
        CHECK (LEN(vch1) > 0)
        NOT NULL,
    vch2 varchar(5)
        CONSTRAINT CK_LEN_TOO_SHORT
        CHECK (LEN(vch2) > 0)
        NOT NULL
)
```

When you attempt to run the preceding four INSERT statements for the version of table T in Ch04CheckTooShortNotNull.sql, the final result set contains just one row (the one with values of a and b for vch1 and vch2). This outcome indicates that the addition of the NOT NULL phrase to the vch1 and vch2 column declarations blocks the entry of null values. For the specific input designated by the fourth INSERT statement, the NOT NULL phrase in the vch1 column blocks the input from the last INSERT statement. The NOT NULL phrase in the vch2 column declaration does not become active because the constraint blocking nulls for the vch1 column takes effect first.

## Specifying a Pattern for Character Input

One of the special advantages that CHECK constraints provide is the ability to limit the characters that can appear in a column as well as just the number of characters for a column. Using the LIKE operator with a pattern expression in a CHECK constraint provides a flexible means of validating character input for a column. Because numbers, such as five-digit U.S. zip codes and part "numbers," are often stored as characters, this approach can be very flexible.

You can designate patterns in the following ways:

- One character at a time in square brackets ([character])
- As starting (characters%) or ending (%characters) with one or more characters
- By denoting one or several individual wildcard characters (_) within a sequence of fixed or other wildcard characters (characters__characters)

Besides designating which characters should appear, you can even write pattern expressions to designate which characters should not appear by using a preceding caret symbol (^). You can also denote a range of values for a single character by an expression within square brackets ([beginningcharacter-endingcharacter]).

Here's a list that demonstrates some pattern expressions with selected values that match the expression from the CustomerID column in the Customers table of the Northwind database.

- LIKE 'A%': ALFKI, ANATR, ANTON, AROUT
- LIKE '[A-B][L-N]%': ALFKI, ANATR, ANTON, BLAUS, BLONP
- LIKE 'A%' OR LIKE '%A': ALFKI, ANATR, ANTON, AROUT, FAMIA, FISSA, HILAA, MAGAA, SAVEA, WOLZA
- LIKE '[A-D]___[E-R]': ALFKI, ANATR, ANTON, BLONP, BONAP, BOTTM, COMMI, CONSH, DUMON
- LIKE '[A-D]___[^E-R]': AROUT, BERGS, BLAUS, BOLID, BSFEV, CACTU, CENTC, CHOPS, DRACD

The following CREATE TABLE statement from the Ch04CheckZipPattern.sql file verifies five-digit zip codes with two constraints.

- First, a CHECK constraint verifies that
  - The length of each zip code column value (psc1) has a length of exactly five characters.
  - Each of the characters in a psc1 value is a number character from 0 through 9.
- Second, a NOT NULL phrase adds a second constraint to restrict the input of rows with a null psc1 value.

```
CREATE TABLE T (
    int1 int IDENTITY PRIMARY KEY,
    vch1 varchar(5)
        CHECK (LEN(vch1) > 0)
        NOT NULL,
    vch2 varchar(5)
        CONSTRAINT CK_LEN_TOO_SHORT
        CHECK (LEN(vch2) > 0)
        NOT NULL
)
```

The following set of four INSERT statements generates just one valid row for the table defined by the preceding CREATE TABLE:

- The first statement successfully adds a row to the table.
- The second statement fails the CHECK constraint because its length is four, instead of five, characters.
- The third statement fails because its first character (r) is not a numeric character from 0 through 9.
- The fourth statement fails because it attempts to insert a null value into the psc1 column; a null value is the default value for most columns that do not have a DEFAULT property set to another value.

```
INSERT T (psc1) VALUES('40222')
INSERT T (psc1) VALUES('4022')
INSERT T (psc1) VALUES('r0222')
INSERT T DEFAULT VALUES
```

## Using Multicolumn Constraints

It is sometimes convenient to define constraints across more than one column.

- One obvious reason for a multicolumn constraint is when you need to specify a primary key based on two or more columns, instead of just a single column. You'll need a multicolumn primary key whenever it takes two or more columns to uniquely specify any particular table row.
- In addition, the `CHECK` constraints discussed in the "Specifying a Pattern for Character Input" section apply to just one column, but some database modeling efforts require constraints specified by two or more table columns. For example, your application may approve an order amount column value over a fixed limit only if another column indicates the name of an approving manager.

Multicolumn constraints are sometimes referred to as table constraints because they are not bound to a single column. In fact, the syntax for declaring a multicolumn constraint applies to the table level. As you have seen, the `CREATE TABLE` statement comma delimits the specification for each column. This syntax declares each column at the table level—not within another column of the table. After specifying all the columns for a table within a `CREATE TABLE` statement, you can use commas to delimit additional specifications for each constraint that you want to apply to an overall table instead of one specific column.

---

■ **Note** Multicolumn primary keys can be clustered or nonclustered just like single-column primary keys. The ordering of rows on the storage device for clustered multicolumn keys applies to the primary key index—not to any one column defining the index. You can designate a clustered multicolumn primary key by specifying the termed `CLUSTERED` immediately after the `PRIMARY KEY` phrase. You can optionally replace the `CLUSTERED` keyword by the `NONCLUSTERED` keyword for a nonclustered primary key. The syntax for a clustered primary key index is: `CONSTRAINT PK_Name PRIMARY KEY CLUSTERED (Col1_Name, Col2_Name)`.

---

### Using a Multicolumn Primary Key

Multicolumn primary keys are common in tables that connect two other tables in a many-to-many relationship. A classic many-to-many relationship is the relationship of students and classes. Any one student can register for multiple classes. Most classes have multiple students. A table for storing student grades across multiple classes can have at least three columns. One column can identify the class, such as a `ClassID` number. A second column can identify individual students with another number, such as `StudentID`. A third column can store the grade for the student in the class.

The CH04PKStudentGrades.sql file shows the syntax for defining a multicolumn primary key in a table called `ClassGrades`. The `CREATE TABLE` for the table follows.

- The `ClassID` and `StudentID` columns each have an `int` data type.
- The `GradeLetter` column stores grades. This column has a `varchar(2)` type. The first character of the `GradeLetter` column denotes a letter for the grade, and the second column optionally records a + or – for grades such as A+ or A–.
- The name for the primary key constraint is `PK_ClassGrades`. The two columns defining the primary key are `ClassID` and `StudentID`. Notice the declaration for this constraint appears at the same level as the column declarations.

```
CREATE TABLE ClassGrades(
    ClassID int,
    StudentID int,
    GradeLetter varchar(2),
    Constraint PK_ClassGrades
        PRIMARY KEY(ClassID, StudentID)
)
```

The following three INSERT statements attempt to add three new rows to the ClassGrades table.

- The first statement adds a grade of A for the student with a StudentID of 1 in the class with a ClassID of 1.
- The second statement adds a grade of B– for another student with a StudentID of 2 in the same class as the first student.
- The third INSERT statement designates a grade of C– for a student in the class with a ClassID of 1. This statement does not specify a value for StudentID. Because StudentID is a part of the table's primary key, SSE automatically imposes a NOT NULL constraint for the column. As a result of the missing value for StudentID, the third INSERT statement fails and does not add a new row.

```
INSERT ClassGrades VALUES(1,1,'A')
INSERT ClassGrades VALUES(1,2,'B-')
INSERT ClassGrades (ClassID, GradeLetter)
VALUES(1,'C-')
```

An excerpt from the listing for running the script in CH04PKStudentGrades.sql describes the error message for the null value for StudentID in the third INSERT statement. Then, the listing enumerates the two rows added to the ClassGrades table. The statement generating the following output is SELECT * FROM ClassGrades.

```
Msg 515, Level 16, State 2, Line 4
Cannot insert the value NULL into column 'StudentID',
table 'ProSSEAPPSCh04.dbo.ClassGrades'; column does not
allow nulls. INSERT fails.
The statement has been terminated.
ClassID     StudentID   GradeLetter
----------- ----------- -----------
1           1           A
1           2           B-
```

## Using a Multicolumn CHECK Constraint

A CHECK constraint can also have multiple columns. In the context of the ClassGrades table, you can specify a set of valid letter grades and a range of valid integer values to denote classes. Rows with GradeLetter values outside the letter range or ClassID values outside of scope of valid numbers for classes can be rejected by a CHECK constraint.

The following CREATE TABLE statement is from the Ch04CheckPKStudentGrades.sql file. The file's contents are exactly the same as for the CREATE TABLE statement in the preceding sample, except for one constraint.

- The new constraint is the CK_GradeLetter_ClassID CHECK constraint at the end of the statement. Notice the new constraint is comma delimited from the preceding PRIMARY KEY constraint.
- The CHECK constraint has two expressions for restricting the content in two columns.
  - The first expression designates the range of legitimate letters for grades in the GradeLetter column, namely, A–F.
  - The second expression indicates that the most positive ClassID value is less than 1,000, namely, 999.
  - An AND operator joins the two expressions so that the data for a new row must comply with both expressions to be valid data according to the CK_GradeLetter_ClassID CHECK constraint.

```
CREATE TABLE ClassGrades(
    ClassID int,
    StudentID int,
    GradeLetter varchar(2),
    Constraint PK_ClassGrades
        PRIMARY KEY(ClassID, StudentID),
    Constraint CK_GradeRange_ClassID
        CHECK (LEFT(UPPER(GradeLetter),1)
        LIKE '[A-F]' AND ClassID < 1000)
)
```

The following set of five INSERT statements attempt to add five new rows, but only three attempts succeed. The third and fourth INSERT statements fail.

- The third statement fails because its GradeLetter value (V–) has a letter outside of the bounds for legitimate letter grades (A–F).
- The fourth statement fails because its ClassID value (1001) is greater than the upper limit for ClassID values (999).

```
INSERT ClassGrades VALUES(1, 1, 'C+')
INSERT ClassGrades VALUES(1, 2, 'A+')
INSERT ClassGrades VALUES(1, 3, 'V-')
INSERT ClassGrades VALUES(1001, 1, 'A')
INSERT ClassGrades VALUES(999, 2, 'A')
```

## Using Foreign Key Constraints

A foreign key constraint is one or more columns in a secondary table that points at corresponding columns in a primary table. For example, if Classes was a primary table, then a secondary table could be ClassGrades. The secondary table, ClassGrades, can use its ClassID column to point back at the ClassID column in the Classes table. Any secondary table can point at multiple primary tables, and therefore have multiple foreign keys. For example, the ClassGrades secondary table can also point at a Students primary table.

SQL Server Express enforces referential integrity by maintaining consistent values in the FOREIGN KEY columns of a secondary table and the PRIMARY KEY columns or the UNIQUE constraint columns of the primary table. Referential integrity rules restrict

- Column values in `FOREIGN KEY` columns to those that match values in the corresponding columns of the primary table
- The revision or deletion of column values in a primary table that will orphan rows in a secondary table

You can use a `FOREIGN KEY` constraint to designate a relationship between columns in a secondary table that points at columns that are a `PRIMARY KEY` or `UNIQUE` constraint in a primary table. SQL Server Express permits the declaration of `FOREIGN KEY` constraints at the column and table level. A table-level declaration of a `FOREIGN KEY` constraint is essential when a single foreign key extends across multiple columns. However, table-level declarations for single-column foreign keys can also improve the readability of the declarations for the associated columns.

Foreign keys optionally allow the specification of rules to define how foreign key columns in a secondary table will change when corresponding column values in the primary table are updated or deleted. To designate how the secondary table can change in response to updates and deletes in the primary table, you need to include the `ON UPDATE` or `ON DELETE` phrases in the `FOREIGN KEY` constraint declaration. Follow the `ON UPDATE` or `ON DELETE` keyword with a keyword to specify the action you wish to happen.

- The default rule prohibits changes to a primary table that require a corresponding change in a secondary table. In other words, the default rule allows "no action" to the primary table that requires a corresponding change to the secondary table. You can optionally specify the `NO ACTION` phrase to designate the default rule. In fact, `ON UPDATE` and `ON DELETE` are not necessary when the default action is acceptable.
- `CASCADE` extends the action for a row in the primary table to the corresponding rows in the secondary table.
  - `CASCADE` after `ON DELETE` causes the deletion of rows in a secondary table that correspond to a deleted row in a primary table.
  - `CASCADE` after `ON UPDATE` causes the revision of foreign key values after matching column values are updated in a primary table.
- `SET NULL` causes foreign key column values to be set to null values when updates or deletes occur to matching values in a primary table.
- `SET DEFAULT` causes foreign key column values to equal their `DEFAULT` property setting when updates or deletes occur to matching values in a primary table.

## Supporting Basic Referential Integrity with Foreign Keys

Foreign keys tightly link and synchronize a pair of tables through referential integrity. Unless you enable an action through a `FOREIGN KEY` constraint on the secondary table, you may not be able to perform the action on the primary table. This section systematically explores the barriers to actions. It also demonstrates how you can enable actions on both primary and secondary tables tied together by referential integrity.

The Ch04FKClassesClassGrades.sql file contains a lengthy script that illustrates the operation of referential integrity. The script works with two tables—a primary table and a secondary table. The primary table, named `Classes,` holds data about classes, including `ClassID` and `ClassTitle`. The following excerpt from Ch04FKClassesClassGrades.sql shows the syntax for creating `Classes`. Notice especially the `PRIMARY KEY` constraint specified in the `ClassID` column declaration. Recall that a primary table must have either a `PRIMARY KEY` or `UNIQUE` constraint.

```
CREATE TABLE Classes(
    ClassID int PRIMARY KEY,
    ClassTitle varchar(50)
)
```

The secondary table, named `ClassGrades` created in Ch04FKClassesClassGrades.sql, contains data about student grades in classes, including `ClassID`, `StudentID`, and `GradeLetter`. The `CREATE TABLE` statement for `ClassGrades` appears in the following code listing.

- The secondary table contains a `FOREIGN KEY` constraint.
- The `REFERENCES` keyword in the `ClassId` column declares the `FOREIGN KEY` constraint within the column. The `REFERENCES` clause ties the `ClassID` column in the `ClassGrades` table to the `ClassID` column in the `Classes` table.
- The `CASCADE` keyword after the `ON UPDATE` phrase allows updates for `ClassID` values in the `Classes` table to propagate to corresponding rows in the `ClassGrades` table. In effect, changes to the primary table cascade to the secondary table.
- The `ClassID` column in `ClassGrades` also participates in a `PRIMARY KEY` constraint for the secondary table declared at the end of the `CREATE TABLE` statement.

```
CREATE TABLE ClassGrades(
    ClassID int REFERENCES Classes(ClassID)
        ON UPDATE CASCADE,
    StudentID int,
    GradeLetter varchar(2),
    Constraint PK_ClassGrades
        PRIMARY KEY(ClassID, StudentID)
)
```

■ **Note** Referential integrity can have an impact on the order in which operations can be performed. Several examples in the book illustrate this point. Additional comments in the script within Ch04FKClassesClassGrades.sql have useful hints about the order in which operations can be performed. For example, after you bind a secondary table to a primary table with a `FOREIGN KEY` constraint, you cannot drop the primary table until you first drop the secondary table. If you plan on taking advantage of `FOREIGN KEY` constraints in your database development projects, I urge you to study the comments about the order of operations in the book and in the script to sharpen your understanding of referential integrity.

The next excerpt from Ch04FKClassesClassGrades.sql initially populates both the `Classes` and `ClassGrades` tables. Because of referential integrity, it is necessary to run the `INSERT` statements for the `Classes` table before the `INSERT` statements for the `ClassGrades` table. Running the `INSERT` statements in the reverse order would generate an error based on a violation of the `FOREIGN KEY` constraint. Recall that column values for `PRIMARY KEY` or `UNIQUE` constraints must exist in the primary table before you can reference them from the secondary table.

```
INSERT Classes VALUES(1,
    'Learning SQL Server Express')
INSERT Classes VALUES(999,
    'Biographies of Jesus Christ')
GO

INSERT ClassGrades VALUES(1, 1, 'C+')
INSERT ClassGrades VALUES(1, 2, 'A+')
INSERT ClassGrades VALUES(999, 2, 'A')
GO
```

The following excerpt from Ch04FKClassesClassGrades.sql attempts to insert a new row in the ClassGrades table that points at a class with a ClassID value of 998. The statement fails when it is run at this point. The reason for the failure is that the INSERT statement references a ClassID value in the secondary table that does not exist in the primary table. This action violates referential integrity.

```
INSERT ClassGrades VALUES(998, 1, 'B')
GO
```

The next excerpt from Ch04FKClassesClassGrades.sql illustrates a typical kind of change that you will make when working with a pair of tables. The script shows the result sets from a SELECT statement before and after an update to a ClassTitle column value in the Classes table for the row with a ClassID of 999. The SELECT statement joins columns from both the Classes and ClassGrades tables. C is an alias for Classes and CG is an alias for ClassGrades. These alias names are used as qualifiers for the column names in the SELECT statement. Referential integrity does not have an impact on the ability to make an update to a ClassTitle column value in the Classes table because referential integrity in this sample does not apply to the ClassTitle. Referential integrity applies to ClassID.

```
SELECT CG.StudentID, C.ClassTitle, CG.GradeLetter
FROM Classes C, ClassGrades CG
WHERE C.ClassID = CG.ClassID
GO

UPDATE Classes
SET ClassTitle = 'The Life of Jesus Christ'
WHERE ClassID = 999
GO

SELECT CG.StudentID, C.ClassTitle, CG.GradeLetter
FROM Classes C, ClassGrades CG
WHERE C.ClassID = CG.ClassID
GO
```

The listing from running the preceding excerpt follows.

- The three rows in the two result sets follow from the initial INSERT statements to populate the Classes and ClassGrades tables as well as from the UPDATE statement in the immediately preceding script.
    - The number of rows affected by the first SELECT statement is three.
    - The intervening UPDATE statement affects one row.
    - The second SELECT statement also affects three rows.
- Notice the ClassTitle column value switches from Biographies of Jesus Christ to The Life of Jesus Christ between the first and second result sets. This is a consequence of the UPDATE statement.

```
StudentID   ClassTitle                 GradeLetter
----------- -------------------------- -----------
1           Learning SQL Server Express C+
2           Learning SQL Server Express A+
2           Biographies of Jesus Christ A
```

```
(3 row(s) affected)



(1 row(s) affected)

StudentID   ClassTitle                 GradeLetter
----------- -------------------------- -----------
1           Learning SQL Server Express C+
2           Learning SQL Server Express A+
2           The Life of Jesus Christ    A

(3 row(s) affected)
```

The UPDATE statement in the next excerpt does depend on referential integrity and the CASCADE keyword following the ON UPDATE phrase in the CREATE TABLE statement for ClassGrades.

- This is because the UPDATE statement revises a ClassID column value in the Classes table, and the ClassID column is also referenced by the FOREIGN KEY constraint in the CREATE TABLE statement for ClassGrades.
- Without special settings, a FOREIGN KEY constraint blocks changes to ClassID in the primary table that affects rows in the secondary table.
- However, the ON UPDATE phrase followed by the CASCADE keyword in the REFERENCES clause for ClassID does the following:
  - Permits the update
  - Enables the changes in the Classes table to propagate to the ClassGrades table

```
SELECT * FROM Classes
GO

UPDATE Classes
SET ClassID = 998
WHERE ClassID = 999
GO

SELECT * FROM Classes
GO

SELECT CG.StudentID, C.ClassTitle, CG.GradeLetter
FROM Classes C, ClassGrades CG
WHERE C.ClassID = CG.ClassID
GO
```

The next listing shows the output from running the preceding script. This output assumes that you are running the inserts and updates sequentially as described in this section. The SELECT statements from the Classes table show the Classes table before and after the update to the ClassID column value for the row with an initial ClassID column value of 999. After the update, the ClassID column value switches successfully from 999 to 998. If this change did not propagate to the ClassGrades table, the concluding SELECT statement that joins the Classes and ClassGrades table would fail for one row in ClassGrades because ClassGrades would have a ClassID value of 999 that did not exist in Classes.

```
ClassID     ClassTitle
----------- --------------------------
1           Learning SQL Server Express
999         The Life of Jesus Christ

(2 row(s) affected)



(1 row(s) affected)

ClassID     ClassTitle
----------- --------------------------
1           Learning SQL Server Express
998         The Life of Jesus Christ

(2 row(s) affected)


StudentID   ClassTitle                 GradeLetter
----------- -------------------------- -----------
1           Learning SQL Server Express C+
2           Learning SQL Server Express A+
2           The Life of Jesus Christ    A

(3 row(s) affected)
```

The final excerpt from Ch04FKClassesClassGrades.sql contrasts an update and a delete to the `Classes` table. As you have already seen, an update succeeds.

- This is because of the `ON UPDATE` phrase and the `CASCADE` keyword.
- A delete to `Classes` fails for a similar reason—that is, the absence of an `ON DELETE` phrase followed by `CASCADE` in the `REFERENCES` clause for `ClassID` within the `CREATE TABLE` statement for `ClassGrades`.
- The following script sandwiches a `DELETE` statement for the row in `Classes` with a `ClassID` value of 999 between two `UPDATE` statements.
    - The statement preceding the `DELETE` statement ensures that the row with a `ClassID` of 999 is in the table to delete.
    - The statement trailing the `DELETE` statement restores the `ClassID` value of 998.

```
UPDATE Classes
SET ClassID = 999
WHERE ClassID = 998
GO

DELETE
FROM Classes
WHERE ClassID = 999
GO

UPDATE Classes
SET ClassID = 998
WHERE ClassID = 999
GO
```

## Implementing Many-to-Many Relationships with Foreign Keys

The concluding sample for this chapter demonstrates how to support a many-to-many relationship with foreign keys between two pairs of tables.

- The sample many-to-many relationship is between students and classes. Therefore, the demonstration application creates tables called Students and Classes. These two tables are primary tables.
- Many-to-many relationships require an intermediate table between the two main tables in the relationship. In our sample, the intermediate table will be ClassGrades.
- You can build the structure for the many-to-many relationship by using the intermediate table as a secondary table to each of the primary tables.

The following excerpt from Ch04FKMany-to-many.sql shows the CREATE TABLE statements for the two primary tables—namely, Students and Classes. The Students table has a StudentID column defined with an int data type that serves as the table's primary key. The Students table also has a computed column named FullName. This column is the concatenation of the FirstName and LastName column values with an intervening space. The Classes table in Ch04FKMany-to-many.sql has the same CREATE TABLE statement as in the preceding sample. The primary key for the Classes table is based on the ClassID column.

```
CREATE TABLE Students(
    StudentID int Primary Key,
    FirstName nvarchar(30),
    LastName nvarchar(50),
    FullName AS (FirstName + ' ' + LastName)
)
GO

CREATE TABLE Classes(
    ClassID int Primary Key,
    ClassTitle varchar(50)
)
GO
```

The intermediate table, ClassGrades, has three columns and three constraints. The three column names are ClassID, StudentID, and GradeLetter. The main purpose for the ClassGrades table is to store student grades in classes.

- The rows in the ClassGrades table are unique by the combination of ClassID and StudentID. Therefore, the PRIMARY KEY constraint for ClassGrades, PK_ClassGrades, relies on both its ClassID and StudentID column values.
- The FOREIGN KEY constraint that references the ClassID column values in the Classes table with the ClassID values in the ClassGrades table has the name FK_Classes_ClassID. This constraint enables cascading updates but no cascading deletes from the Classes table to the ClassGrades table.
- The FOREIGN KEY constraint that references the StudentID column values in the Students table with the StudentID values in the ClassGrades table has the name FK_Students_StudentID. This constraint also supports cascading updates, but it does not enable cascading deletes.

```
    CREATE TABLE ClassGrades(
        ClassID int,
        StudentID int,
        GradeLetter varchar(2),
        Constraint PK_ClassGrades
            PRIMARY KEY(ClassID, StudentID),
        Constraint FK_Classes_ClassID
            FOREIGN KEY(ClassID)
            REFERENCES Classes(ClassID) ON UPDATE CASCADE,
        Constraint FK_Students_StudentID
            FOREIGN KEY(StudentID)
            REFERENCES Students(StudentID) ON UPDATE CASCADE
    )
```

The next excerpt uses INSERT statements to populate the Classes, Students, and ClassGrades tables with an initial set of values. A concluding SELECT statement displays the values input with a result set that draws on all three tables. The SELECT statement's FROM clause defines aliases of C, CG, and S for the Classes, ClassGrades, and Students tables. The SELECT list starts with the reference to the FullName computed column from the Students tables. Other items in the SELECT list include ClassTitle from the Classes table and GradeLetter from the ClassGrades table.

```
--Insert classes rows
INSERT Classes VALUES(1,
    'Learning SQL Server Express')
INSERT Classes VALUES(999,
    'Biographies of Jesus Christ')
GO

--Insert Students rows
INSERT Students VALUES(1, 'Poor', 'DBA')
INSERT Students VALUES(2, 'Better', 'DBA')
GO

--Insert ClassGrades rows
INSERT ClassGrades VALUES(1, 1, 'C+')
INSERT ClassGrades VALUES(1, 2, 'A+')
INSERT ClassGrades VALUES(999, 2, 'A')
GO

--Show table values after initial population
SELECT S.FullName, C.ClassTitle, CG.GradeLetter
FROM Classes C, ClassGrades CG, Students S
WHERE C.ClassID = CG.ClassID AND
    S.StudentID = CG.StudentID
GO
```

The output from the preceding SELECT statement appears next. As you can see, there are just two students with names of Poor DBA and Better DBA. The student with a FullName of Poor DBA earned a C+ in the Learning SQL Server Express class. The other student, Better DBA, earned grades of A+ and A, respectively, in the Learning SQL Server Express and Biographies of Jesus Christ classes. These values follow from the INSERT statements preceding the SELECT statement.

```
FullName   ClassTitle                  GradeLetter
---------- --------------------------- -----------
Poor DBA   Learning SQL Server Express C+
Better DBA Learning SQL Server Express A+
Better DBA Biographies of Jesus Christ A
```

The next block of code, which is an excerpt from Ch04FKMany-to-many.sql, updates the column values for both `ClassTitle` and `ClassID` in the `Classes` table.

- The `ClassID` value changes to 998 for the row with a `ClassID` value of 999.
- The `ClassTitle` value for the row with an initial `ClassID` value of 999 updates to The Life of Jesus Christ.

You can designate changes for multiple columns within a single `UPDATE` statement by following a single `SET` clause with a separate comma-delimited assignment for each updated column value. The following code sample uses two assignment statements, one for `ClassID` and another for `ClassTitle`. The `UPDATE` statement occurs between two `SELECT` statements that show the `ClassID` and `ClassTitle` column values from the `Classes` tables.

```
SELECT * FROM Classes
GO

UPDATE Classes
SET ClassID = 998,
    ClassTitle = 'The Life of Jesus Christ'
WHERE ClassID = 999
GO

SELECT * FROM Classes
GO
```

The following listing shows the results from the preceding script as well as an additional `SELECT` statement that generates a result set showing `FullName`, `ClassTitle`, and `GradeLetter` column values for all combinations of student and class. The syntax for this additional `SELECT` statement is the same one that shows the initial values of the `Students`, `Classes`, and `ClassGrades` tables. The preceding `UPDATE` statement changed both the `ClassTitle` and the `ClassID` column values in the `Classes` table. However, a cascading update working through referential integrity revises the `ClassID` column value from the `ClassGrades` table to match the corresponding changed `ClassID` value from the `Classes` table.

```
ClassID     ClassTitle
----------- ---------------------------
1           Learning SQL Server Express
999         Biographies of Jesus Christ

(2 row(s) affected)



(1 row(s) affected)

ClassID     ClassTitle
----------- ---------------------------
1           Learning SQL Server Express
998         The Life of Jesus Christ

(2 row(s) affected)


FullName   ClassTitle                  GradeLetter
---------- --------------------------- -----------
Poor DBA   Learning SQL Server Express C+
Better DBA Learning SQL Server Express A+
Better DBA The Life of Jesus Christ    A

(3 row(s) affected)
```

The final portion of the script in Ch04FKMany-to-many.sql makes a change to the last name of the student named Better DBA. Although this change does not directly depend on referential integrity, it is typical of the kind of change that will be made in applications that implement referential integrity.

Let's assume Better DBA marries a preacher named Inspirational Minister. Because she has already attracted attention as an exceptional DBA, Better DBA does not want to totally give up her former last name. After her marriage, she changes her name to a hyphenated form (DBA-Minister). Then, she goes to the Office of the Registrar at her school and submits a request to change her last name. The following excerpt shows the code to implement the update and a new version of the result set to confirm the revision. The result set derives from the standard `SELECT` statement for this sample joining the `Classes`, `Students`, and `ClassGrades` tables.

```
UPDATE Students
SET LastName = 'DBA-Minister'
WHERE StudentID = 2
GO
```

```
FullName            ClassTitle                  GradeLetter
------------------- --------------------------- -----------
Poor DBA            Learning SQL Server Express C+
Better DBA-Minister Learning SQL Server Express A+
Better DBA-Minister The Life of Jesus Christ    A
```

## Summary

This chapter showed you how to create tables for the solutions that you build with SSE. Tables are at the heart of every custom database solution. If you can't build your own tables, you are limited to selecting from tables created by other developers and administrators. Even in this case, knowing about the table design issues covered in this chapter can help you to design smarter `SELECT` statements and to debug your `SELECT` statements faster.

- Because data types are so fundamental to your ability to create tables, the chapter commenced with a review that summarized and contrasted the more than two dozen SSE data types.
- Next, the focus shifted to creating tables with columns based on SQL Server data types. You also learned how to populate tables with data as well as how to recover data from an old version of the table and restore the data to a new version of the table.
- The chapter closed with two lengthy sections that drilled down on data integrity. By building tables that take advantage of the many features that SSE offers for managing the quality of data that enters a database, you ensure that your database solutions are never characterized as garbage in, garbage out.