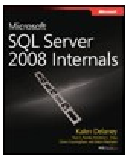


Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.
Microsoft Press. (c) 2009. Copying Prohibited.

Chapter 5: Tables

Kalen Delaney

Overview

In this chapter, we'll start with a basic introduction to tables and continue into some very detailed examinations of their internal structures. Simply put, a *table* is a collection of data about a specific *entity* (a person, place, or thing) that has a discrete number of named *attributes* (for example, quantity or type). Tables are at the heart of Microsoft SQL Server and the relational model in general. In SQL Server, a table is often referred to as a *base table* to emphasize where data is stored. Calling it a base table also distinguishes it from a *view*—a virtual table that's an internal query referencing one or more base tables or other views.

Attributes of a table's data (such as color, size, quantity, order date, and supplier's name) take the form of named *columns* in the table. Each instance of data in a table is represented as a single entry, or *row* (formally called a *tuple*). In a true relational database, each row in a table is unique and has a unique identifier called a *primary key*. (SQL Server, in accordance with the ANSI SQL standard, doesn't require you to make a row unique or declare a primary key. However, because both of these concepts are central to the relational model, I recommend that you always implement them.)

Most tables have some relationship to other tables. For example, in a typical order-entry system, the *orders* table has a *customer_number* column for keeping track of the customer number for an order, and *customer_number* also appears in the *customer* table. Assuming that *customer_number* is a unique identifier, or primary key, of the *customer* table, a foreign key relationship is established by which *orders* and *customer* tables can subsequently be joined.

So much for the 30-second database design primer. You can find plenty of books that discuss logical database and table design, but this isn't one of them. I'll assume that you understand basic database theory and design and that you generally know what your tables will look like. The rest of this chapter discusses the internals of tables in SQL Server 2008.

Creating Tables

To create a table, SQL Server uses the ANSI SQL standard *CREATE TABLE* syntax. SQL Server Management Studio provides a front-end, fill-in-the-blank table designer that can sometimes make your job easier. Ultimately, the SQL syntax is always sent to SQL Server to create a table, no matter what interface you use. In this chapter, I'll emphasize direct use of the Data Definition Language (DDL) rather than discuss the GUI tools. You should keep all DDL commands in a script so you can run them easily at a later time to re-create the table. (Even if you use one of the friendly front-end tools, it's critical that you are able to re-create the table later.) Management Studio and other front-end tools can create and save operating system files using the SQL DDL commands necessary to create any object. This DDL is essentially source code, and you should treat it as such. Keep a backup copy. You should also consider keeping these files under version control using a source control product such as Microsoft Visual SourceSafe.

At the basic level, creating a table requires little more than knowing what you want to name it, what columns it contains, and what range of values (domain) each column can store. Here's the basic syntax for creating the *customer* table in the *dbo* schema, with three fixed-length character (*char*) columns. (Note that this table definition isn't necessarily the most efficient way to store data because it always requires 46 bytes per entry for data plus a few bytes of overhead, regardless of the actual length of the data.)

```
CREATE TABLE dbo.customer
(
name          char(30),
phone         char(12),
emp_id        char(4)
);
```

This example shows each column on a separate line for readability. As far as the SQL Server parser is concerned, white spaces created by tabs, carriage returns, and the spacebar are identical. From the system's standpoint, the following *CREATE TABLE* example is identical to the preceding one, but it's harder to read from a user's standpoint:

```
CREATE TABLE customer (name char(30), phone char(12), emp_id char(4));
```

Naming Tables and Columns

A table is always created within one schema of one database. Tables also have owners, but unlike in versions of SQL

Server prior to 2005, the table owner is not used to access the table. The schema is used for all object access. Normally, a table is created in the default schema of the user who is creating it, but the *CREATE TABLE* statement can specify the schema in which the object is to be created. A user can create a table only in a schema for which the user has *ALTER* permissions. Any user in the *sysadmin*, *db_ddladmin*, or *db_owner* roles can create a table in any schema. A database can contain multiple tables with the same name, so long as the tables are in different schemas. The full name of a table has three parts, in the following form:

database_name.schema_name.table_name

The first two parts of the three-part name specification have default values. The default for the name of the database is whatever database context in which you're currently working. The *schema_name* actually has two possible defaults when querying. If no schema name is specified when you reference an object, SQL Server first checks for an object in your default schema. If there is no such table in your default schema, SQL Server then checks to see if there is an object of the specified name in the *dbo* schema.

Note To access a table or other object in a schema other than your default schema or the *dbo* schema, you must include the schema name along with the table name. In fact, you should get in the habit of always including the schema name when referring to any object in SQL Server 2008. Not only does this remove any possible confusion about which schema you are interested in, but it can lead to some performance benefits.

The *sys* schema is a special case. For compatibility views, such as *sysobjects,* SQL Server accesses the object in the *sys* schema prior to any object you might have created with the same name. Obviously, it is not a good idea to create an object of your own called *sysobjects,* as you will never be able to access it. Compatibility views can also be accessed through the *dbo* schema, so the objects *sys.sysobjects* and *dbo.sysobjects* are the same. For catalog views and Dynamic Management Objects, you must specify the *sys* schema to access the object.

You should make column names descriptive, and because you'll use them repeatedly, you should avoid wordiness. The name of the column (or any object in SQL Server, such as a table or a view) can be whatever you choose, so long as it conforms to the SQL Server rule for regular identifiers: it must consist of a combination of 1 through 128 letters, digits, or the symbols #*, $,* @*,* or _*.*

More Info Alternatively, you can use a delimited identifier that includes any characters you like. For more about identifier rules, see "Identifiers" in *SQL Server Books Online.* The discussion there applies to all SQL Server object names, not just column names.

In some cases, you can access a table using a four-part name, in which the first part is the name of the SQL Server instance. However, you can refer to a table using a four-part name only if the SQL Server instance has been defined as a linked server. You can read more about linked servers in *SQL Server Books Online;* I won't discuss them further here.

Reserved Keywords

Certain reserved keywords, such as *TABLE, CREATE, SELECT,* and *UPDATE,* have special meaning to the SQL Server parser, and collectively they make up the SQL language implementation. You should avoid using reserved keywords for your object names. In addition to the SQL Server reserved keywords, the SQL-92 standard has its own list of reserved keywords. In some cases, this list is more restrictive than the SQL Server list; in other cases, it's less restrictive. *SQL Server Books Online* includes both lists.

Watch out for the SQL-92 reserved keywords. Some of the words aren't reserved keywords in SQL Server yet, but they might become reserved keywords in a future SQL Server version. If you use a SQL-92 reserved keyword, you might end up having to alter your application before upgrading it if the word becomes a SQL Server reserved keyword.

Delimited Identifiers

You can't use keywords in your object names unless you use a delimited identifier. In fact, if you use a delimited identifier, not only can you use keywords as identifiers, but you can also use any other string as an object name—whether or not it follows the rules for identifiers. This includes spaces and other nonalphanumeric characters that are normally not allowed. Two types of delimited identifiers exist:

- Bracketed identifiers, which are delimited by square brackets ([*object name*])
- Quoted identifiers, which are delimited by double quotation marks ("*object name*")

You can use bracketed identifiers in any environment, but to use quoted identifiers, you must enable a special option using SET QUOTED_IDENTIFIER ON. If you turn on QUOTED_ IDENTIFIER, double quotes are interpreted as referencing an object. To delimit string or date constants, you must use single quotes.

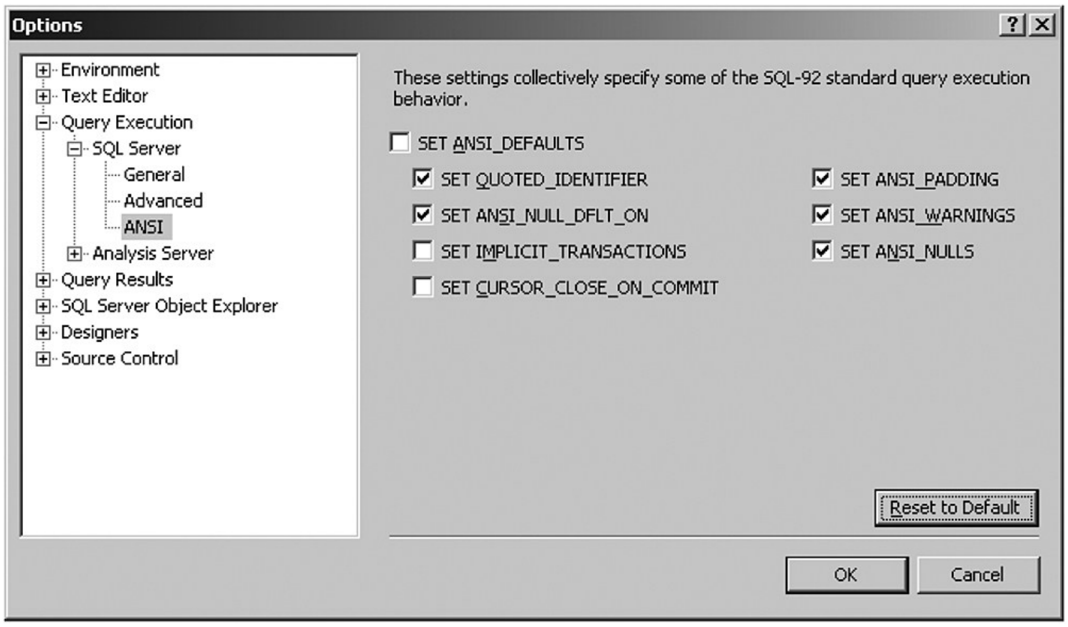
Let's look at some examples. Because *column* is a reserved keyword, the first statement that follows is illegal in all circumstances. The second statement is illegal unless QUOTED_ IDENTIFIER is on. The third statement is legal in any circumstance:

```
CREATE TABLE dbo.customer(name char(30), column char(12), emp_id char(4));

CREATE TABLE dbo.customer(name char(30), "column" char(12), emp_id char(4));

CREATE TABLE dbo.customer(name char(30), [column] char(12), emp_id char(4));
```

The SQL Native Client ODBC driver and SQL Native Client OLE DB Provider for SQL Server automatically set QUOTED_IDENTIFIER to ON when connecting. You can configure this in ODBC data sources, ODBC connection attributes, or OLE DB connection properties. You can determine whether this option is on or off for your session by executing the following query:

```
SELECT quoted_identifier
FROM sys.dm_exec_sessions
WHERE session_id = @@spid;
```

A result value of 1 indicates that QUOTED_IDENTIFIER is ON. If you're using Management Studio, you can check the setting by running the preceding command in a query window or by choosing Options from the Tools menu and then expanding the Query Execution/SQL Server node and examining the ANSI properties information, as shown in Figure 5-1.



These settings collectively specify some of the SQL-92 standard query execution behavior.

- [ ] SET ANSI_DEFAULTS
- [✓] SET QUOTED_IDENTIFIER
- [✓] SET ANSI_NULL_DFLT_ON
- [ ] SET IMPLICIT_TRANSACTIONS
- [ ] SET CURSOR_CLOSE_ON_COMMIT
- [✓] SET ANSI_PADDING
- [✓] SET ANSI_WARNINGS
- [✓] SET ANSI_NULLS

Figure 5-1: Examining the ANSI properties for a connection in Management Studio

Tip Technically, you can use delimited identifiers with all object and column names, so you never have to worry about reserved keywords. However, I don't recommend this. Many third-party tools for SQL Server don't handle quoted identifiers well, and they can make your code difficult to read. Using quoted identifiers might also make upgrading to future versions of SQL Server more difficult.

Rather than using delimited identifiers to protect against reserved keyword problems, you should simply adopt some simple naming conventions. For example, you can precede column names with the first few letters of the table name and an underscore. This naming style makes the column or object name more readable and also greatly reduces your chances of encountering a keyword or reserved word conflict.

Naming Conventions

Many organizations and multiuser development projects adopt standard naming conventions. This is generally a good practice. For example, assigning a standard moniker of *cust_id* to represent a customer number in every table clearly shows that all the tables share common data. If an organization instead uses several monikers in the tables to represent a customer number, such as *cust_id*, *cust_num*, *customer_number*, and *customer_#*, it won't be as obvious that these monikers represent common data.

One naming convention is the Hungarian-style notation for column names. Hungarian-style notation is a widely used practice in C programming, whereby variable names include information about their data types. This notation uses names such as *sint_nn_custnum* to indicate that the *custnum* column is a small integer (*smallint* of 2 bytes) and is NOT NULL (doesn't allow nulls). Although this practice makes good sense in C programming, it defeats the data type independence that SQL Server provides; therefore, I recommend against using it.

Data Types

SQL Server provides many data types, most of which are straightforward. Choosing the appropriate data type is simply a matter of mapping the domain of values you need to store to the corresponding data type. In choosing data types, you want to avoid wasting storage space while allowing enough space for a sufficient range of possible values over the life of your application. Discussing the details about all the possible considerations when programming with the various data types is beyond the scope of this book. For the most part, I'll just cover some of the basic issues related to dealing with the various data types.

Choosing a Data Type

The decision about what data type to use for each column depends primarily on the nature of the data the column holds and the operations you want to perform on the data. The five basic data type categories in SQL Server 2008 are numeric, character, date and time, Large Object (LOB), and miscellaneous. SQL Server 2008 also supports a variant data type called *sql_variant*. Values stored in a *sql_variant* column can be of almost any data type. I'll discuss LOB columns in Chapter 7, "Special Storage," because their storage format is different than that of other data types discussed in this chapter. In this section, I'll examine some of the issues related to storing data of different data types.

Numeric Data Types

You should use numeric data types for data on which you want to perform numeric comparisons or arithmetic operations. Your main decisions are the maximum range of possible values you want to be able to store and the accuracy you need. The tradeoff is that data types that can store a greater range of values take up more space.

Numeric data types can also be classified as either exact or approximate. Exact numeric values are guaranteed to store exact representations of your numbers. Approximate numeric values have a far greater range of values, but the values are not guaranteed to be stored precisely. The greatest range of values that exact numeric values can store data is –10^38 + 1 to 10^38 –1. Unless you need numbers with greater magnitude, I recommend that you not use the approximate numeric data types.

The exact numeric data types can be divided into two groups: integers and decimals. Integer types range in size from 1 to 8 bytes, with a corresponding increase in the range of possible values. The *money* and *smallmoney* data types are included frequently among the integer types because internally they are stored in the same way. For the *money* and *smallmoney* data types, it is understood that the rightmost four digits are after the decimal point. For the other integer types, no digits come after the decimal point. Table 5-1 lists the integer data types along with their storage size and range of values.

Table 5-1: Range and Storage Requirements for Integer Data Types

Data Type	Range	Storage (Bytes)
bigint	-2^{63} to $2^{63}-1$	8
int	-2^{31} to $2^{31}-1$	4
Smallint	-2^{15} to $2^{15}-1$	2
Tinyint	0 to 255	1
Money	–922,337,203,685,477.5808 to 922,337,203,685,477.5807, with accuracy of one ten-thousandth of a monetary unit	8
Smallmoney	–214,748.3648 to 214,748.3647, with accuracy of one ten-thousandth of a monetary unit	4

The decimal and numeric data types allow a high degree of accuracy and a large range of values. For those two synonymous data types, you can specify a *precision* (the total number of digits stored) and a *scale* (the maximum number of digits to the right of the decimal point). The maximum number of digits that can be stored to the left of the decimal point is precision – scale (that is, subtract the scale from precision to get the number of digits). Two different decimal values can have the same precision and very different ranges. For example, a column defined as decimal (8,4) can store values from –9,999.9999 to 9,999.9999, and a column defined as decimal (8,0) can store values from –99,999,999 to 99,999,999.

Table 5-2 shows the storage space required for decimal and numeric data based on the defined precision.

Table 5-2: Storage Requirements for Decimal and Numeric Data Types

Precision	Storage (Bytes)
1 to 9	5
10 to 19	9
20 to 28	13
29 to 38	17

Note SQL Server 2005 SP2 added a feature to allow decimal data to be stored in a variable amount of space. This can be useful when you have some values that need a high degree of precision, but most of the values in the column need only a few bytes, or are 0 or NULL. *Vardecimal,* unlike *varchar*, is not a data type, but rather a property of a table which is set by using the *sp_tableoption* procedure, and in SQL Server 2005, it must also be enabled for the database. In SQL Server 2008, all databases except the *master*, *model*, *tempdb,* and *msdb* databases always allow tables to have the *vardecimal storage format* property enabled.

Although the vardecimal storage format can reduce the storage size of the data, it comes at the cost of adding additional CPU overhead. Once the *vardecimal* property is enabled for a table, all *decimal* data in the table is stored as variable-length data. This includes all indexes on *decimal* data and all log records that include *decimal* data.

Changing the value of the *vardecimal storage format* property of a table is an offline operation and SQL Server exclusively locks the table that is being modified until all the decimal data is converted to the new format. The vardecimal storage format has been deprecated, so I will not be describing the details of the internal storage for *vardecimal* data. For new development, it is recommended that you use SQL Server's compression capabilities to minimize your storage requirement for data that requires a variable number of bytes. I will discuss data compression in Chapter 7.

Date and Time Data Types

SQL Server 2008 supports six data types for storing date and time information: *datetime* and *smalldatetime* have been available since the very first version and four new types were added in SQL Server 2008: *date*, *time*, *datetime2,* and *datetimeoffset*. The difference between these types is the range of possible dates, the number of bytes needed for storage, whether both date and time are stored (or just the date or just the time), and whether time zone information is incorporated into the stored value. Table 5-3, taken from *SQL Server 2008 Books Online,* shows the range and storage requirements for each of the date and time data types.

Table 5-3: SQL Server Date and Time Data Types Range and Storage Requirements

Type	Format	Range	Accuracy	Storage Size (bytes)	User-Defined Fractional Second Precision
time	*hh:mm:ss [.nnnnnnn]*	00:00:00.0000000 through 23:59:59.9999999	100 nanoseconds	3 to 5	Yes
date	*YYYY-MM-DD*	0001-01-01 through 9999-12-31	1 day	3	No
smalldatetime	*YYYY-MM-DD hh:mm:ss*	1900-01-01 through 2079-06-06	1 minute	4	No
datetime	*YYYY-MM-DD hh:mm:ss [.nnn]*	1753-01-01 through 9999-12-31	0.00333 second	8	No

datetime2	*YYYY-MM-DD hh:mm:ss [.nnnnnnn]*	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999	100 nanoseconds	6 to 8	Yes
datetimeoffset	*YYYY-MM-DD hh:mm:ss [.nnnnnnn] [+|-] hh:mm*	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999 (in UTC)	100 nanoseconds	8 to 10 (2 bytes for time zone data)	Yes

If no date is supplied, the default of January 1, 1900, is assumed; if no time is supplied, the default of 00:00:00.000 (midnight) is assumed.

Note If you're new to SQL Server date and time data, you might be surprised that for the original *datetime* data type, the earliest possible date that can be stored is January 1, 1753. This was done for historical reasons, and started with the original Sybase specification for the *datetime* data type. In what we sometimes refer to as the Western world, there have been two calendars in modern time: the Julian and the Gregorian calendars. These calendars were a number of days apart (depending on which century you look at), so when a culture that used the Julian calendar moved to the Gregorian calendar, they dropped between 10 to 13 days from the calendar. Great Britain made this shift in 1752, and in that year, September 2 was followed by September 14. Sybase decided not to store dates earlier than 1753 because the date arithmetic functions would be ambiguous. However, other countries made the change at other times, and in Turkey, the calendar was not shifted until 1927.

Internally, values for all the date and time data types are stored completely differently from how you enter them or how they are displayed. Dates and times are always stored as two separate components: a date component and a time component.

For the original *datetime* data types, *datetime* and *smalldatetime*, the data is stored internally as two separate components. For *datetime* values, the data is stored as two 4-byte values, the first (for the date) being the number of days before or after the base date of January 1, 1900, and the second (for the time) being the number of clock ticks after midnight, with each tick representing 3.33 milliseconds, or 1/300 of a second. You can actually see these two parts if you convert a *datetime* value to a binary string of 8 hexadecimal bytes. For *smalldatetime* values, each component is stored in 2 bytes. The date is stored as the number of days after January 1, 1900, and the time is stored as the number of minutes after midnight.

The following example shows how to see the component parts of the current date and time, stored in a variable of type *datetime*, retrieved using the parameterless system function *CURRENT_TIMESTAMP.* The first *CONVERT* operation shows the full hexadecimal byte string that stores the *datetime* value. The second *CONVERT* displays the first four bytes converted to an integer and the third *CONVERT* displays the second four bytes converted to an integer. Because we're storing current date and time in a local variable, we can be sure we're using the same value for all the *CONVERT* operations:

```
DECLARE @today datetime
SELECT @today = CURRENT_TIMESTAMP
SELECT @today AS [CURRENT TIMESTAMP];
SELECT CONVERT (varbinary(8), @today) AS [INTERNAL FORMAT];
SELECT CONVERT (int, SUBSTRING (CONVERT (varbinary(8), @today), 1, 4))
        AS [DAYS AFTER 1/1/1900];
SELECT CONVERT (int, SUBSTRING (CONVERT (varbinary(8), @today), 5, 4))
        AS [TICKS AFTER MIDNIGHT];
```

These are the results when the code runs on July 10, 2008:

```
CURRENT TIMESTAMP
-----------------------
2008-07-10 17:29:11.967

INTERNAL FORMAT
------------------
0x00009AD501202BD6

DAYS AFTER 1/1/1900
-------------------
39637

TICKS AFTER MIDNIGHT
--------------------
18885590
```

Microsoft used the opportunity when adding new *date* and *time* data types in SQL Server 2008 to change the internal representation of dates and times completely. Dates are now stored as a three-byte positive number, representing the number of days after January 1, 0001. For the *datetimeoffset* type, an additional two bytes are used to store a time offset, in hours and minutes, from UTC. Note that although internally, the base date for the new date and data types is January 1, 0001, when SQL Server is interpreting a date value where the actual date is not specified, January 1, 1900, is the default. For example, if you try to insert the string '01:15:00' into a *datetime2* column, SQL Server interprets this as a time of 1:15 on January 1, 1900.

All the new types that contain time information (*time*, *datetime2,* and *datetimeoffset*) allow you to specify the precision of the time component by following the data type name with a number between 1 and 7 indicating the desired scale. The default, if no scale is specified, is to assume a scale of 7. Table 5-4 shows what each of the possible scale values means in terms of the precision and storage requirement of the stored data values.

Table 5-4: Scale Values for Time Data with Storage Requirements and Precision

Specified Scale	Result (precision, scale)	Column Length (bytes)	Fractional Seconds (precision)
none	(16,7)	5	7
(0)	(8,0)	3	0–2
(1)	(10,1)	3	0–2
(2)	(11,2)	3	0–2
(3)	(12,3)	4	3–4
(4)	(13,4)	4	3–4
(5)	(14,5)	5	5–7
(6)	(15,6)	5	5–7
(7)	(16,7)	5	5–7

For a quick look at what the information in Table 5-4 means, you can run the following three conversions:

```
SELECT CAST(CURRENT_TIMESTAMP AS time);
SELECT CAST(CURRENT_TIMESTAMP AS time(2));
SELECT CAST(CURRENT_TIMESTAMP AS time(7));
```

I got the following results. Note that the scale value determines the number of decimal digits and that the value for *time* is identical to *time*(7):

```
17:39:43.0830000
17:39:43.08
17:39:43.0830000
```

Internally, the time is computed using the following formula, assuming *H* is hours, *M* is minutes, *S* is seconds, *F* is fractional sections, and *D* is scale (number of decimal digits):

$(((H * 60) + M) * 60 + S) * 10^D + F$

For example, the value 17:39:43.08, with *time(2)* format, would be stored internally as

$(((17 * 60) + 39) * 60 + 43) * 10^2 + 083$, or 6358383

The same time, stored as *time*(7) would be

$(((17 * 60) + 39) * 60 + 43) * 10^7 + 083$, or 635830000083

In the section entitled "Internal Storage," later in this chapter, we'll see what this data looks like when stored in a data row.

SQL Server 2008 provides dozens of functions for working with *date* and *time* data, as well as dozens of different formats that can be used for interpreting and displaying date and time values. It is beyond the scope of this book to cover date and time data in that level of detail. However, the most important thing to understand about these types is that what you see is not what is actually stored on disk. The on-disk format, whether you're using the old *datetime* and *smalldatetime* types or any of the new types, is completely unambiguous, but it is not very user-friendly. You need to make sure that you provide input data in a format that is also unambiguous. For example, the value '3/4/48' is *not* unambiguous. Does it represent March 4 or April 3, and is the year 1948, 2048, or perhaps 48 (almost 2,000 years ago)? The ISO 8601 format is an international standard with unambiguous specification. In addition this format is not affected by your session's *SET*

DATEFORMAT or *SET LANGUAGE* settings. Using this format, March 4, 1948, could be represented as 19480304 or 1948-03-04.

Character Data Types

Character data types come in four varieties. They can be fixed-length or variable-length strings of single-byte characters (*char* and *varchar*) or fixed-length or variable-length strings of Unicode characters (*nchar* and *nvarchar*). Unicode character strings need two bytes for each stored character; use them when you need to represent characters that can't be stored in the single-byte characters that are sufficient for storing most of the characters in the English and European alphabets. Single-byte character strings can store up to 8,000 characters, and Unicode character strings can store up to 4,000 characters. You should know the type of data that you are dealing with to decide between single-byte and double-byte character strings. Keep in mind that the catalog view *sys.types* reports the length in number of bytes, not in number of characters. In SQL Server 2005 and SQL Server 2008, you can also define a variable-length character string with a MAX length. Columns defined as *varchar(max)* are treated as normal variable-length columns when the actual length is less than or equal to 8,000 bytes, and they are treated as a large object value (mentioned later in this section and covered in detail in Chapter 7) when the actual length is greater than 8,000 bytes.

Deciding whether to use a variable-length or a fixed-length data type is a more difficult decision, and it isn't always straightforward or obvious. As a general rule, variable-length data types are most appropriate when you expect significant variation in the size of the data for a column and when the data in the column won't be changed frequently.

Using variable-length data types can yield important storage savings. It can sometimes result in a minor performance loss, and at other times it can result in improved performance. A row with variable-length columns requires special offset entries to be internally maintained. These entries keep track of the actual length of the column. Calculating and maintaining the offsets requires slightly more overhead than does a pure fixed-length row, which needs no such offsets. This task requires a few addition and subtraction operations to maintain the offset value. However, the extra overhead of maintaining these offsets is generally inconsequential, and this alone would not make a significant difference on most, if any, systems.

Another potential performance issue with variable-length fields is the cost of increasing the size of a row on a page that is almost full. If a row with variable-length columns uses only part of its maximum length and is later updated to a longer length, the enlarged row might no longer fit on the same page. If the table has a clustered index, the row must stay in the same position relative to the other rows, so the solution is to split the page and move some of the rows from the page with the enlarged row onto a newly linked page. This can be an expensive operation. Chapter 6, "Indexes: Internals and Management," describes the details of page splitting and moving rows. If the table has no clustered index, the row can move to a new location and leave a forwarding pointer in the original location. I'll talk about forwarding pointers later in this chapter.

On the other hand, using variable-length columns can sometimes improve performance because it can allow more rows to fit on a page. But the efficiency results from more than simply requiring less disk space. A data page for SQL Server is 8 KB (8,192 bytes), of which 8,096 bytes are available to store data. (The rest is for internal use to keep track of structural information about the page and the object to which it belongs.) One I/O operation brings back the entire page. If you can fit 80 rows on a page, a single I/O operation brings back 80 rows. But if you can fit 160 rows on a page, one I/O operation is essentially twice as efficient. In operations that scan for data and return lots of adjacent rows, this can amount to a significant performance improvement. The more rows you can fit per page, the better your I/O and cache-hit efficiency is.

For example, consider a simple customer table. Suppose that you could define it in two ways: fixed-length and variable-length, as shown in Figures 5-2 and 5-3.

```
USE testdb
GO

CREATE TABLE customer_fixed
(
cust_id                         smallint         NULL,
cust_name                       char(50)         NULL,
cust_addr1                      char(50)         NULL,
cust_addr2                      char(50)         NULL,
cust_city                       char(50)         NULL,
cust_state                      char(2)          NULL,
cust_postal_code                char(10)         NULL,
cust_phone                      char(20)         NULL,
cust_fax                        char(20)         NULL,
cust_email                      char(30)         NULL,
cust_web_url                    char(100)        NULL,
)
```

Figure 5-2: A customer table with all fixed-length columns

```
USE testdb
GO

CREATE TABLE customer_var
(
cust_id                       smallint          NULL,
cust_name                     varchar(50)       NULL,
cust_addr1                    varchar(50)       NULL,
cust_addr2                    varchar(50)       NULL,
cust_city                     varchar(50)       NULL,
cust_state                    char(2)           NULL,
cust_postal_code              varchar(10)       NULL,
cust_phone                    varchar(20)       NULL,
cust_fax                      varchar(20)       NULL,
cust_email                    varchar(30)       NULL,
cust_web_url                  varchar(100)      NULL
)
```

Figure 5-3: A customer table with variable-length columns

Columns that contain addresses, names, or URLs all have data that varies significantly in length. Let's look at the differences between choosing fixed-length columns and choosing variable-length columns. In Figure 5-2, which uses all fixed-length columns, every row uses 384 bytes for data regardless of the number of characters actually inserted in the row. SQL Server also needs an additional 10 bytes of overhead for every row in this table, so each row needs a total of 394 bytes for storage. But let's say that even though the table must accommodate addresses and names up to the specified size, the average row is only half the maximum size.

In Figure 5-3, assume that for all the variable-length (*varchar*) columns the average entry is actually only about half the maximum. Instead of a row length of 394 bytes, the average length is 224 bytes. This length is computed as follows: The *smallint* and *char(2)* columns total 4 bytes. The *varchar* columns' maximum total length is 380, half of which is 190 bytes. And a 2-byte overhead exists for each of nine *varchar* columns, for 18 bytes. Add 2 more bytes for any row that has one or more variable-length columns. In addition, these rows require the same 10 bytes of overhead that the fixed-length rows from Figure 5-2 require, regardless of the presence of variable-length fields. So the total is 4 + 190 + 18 + 2 + 10, or 224. (I'll discuss the actual meaning of each of these bytes of overhead later in this chapter.)

In the fixed-length example in Figure 5-2, you always fit 20 rows on a data page (8,096/394, discarding the remainder). In the variable-length example in Figure 5-3, you can fit an average of 36 rows per page (8,096/224). The table using variable-length columns will consume about half as many pages in storage, a single I/O operation retrieves almost twice as many rows, and a page cached in memory is twice as likely to contain the row you want.

More Info You need additional overhead bytes for each row if you are using snapshot isolation. I'll discuss this concurrency option, as well as the extra row overhead needed to support it, in Chapter 10, "Transactions and Concurrency."

When you choose lengths for columns, don't be wasteful—but don't be cheap, either. Allow for future needs, and realize that if the additional length doesn't change how many rows fit on a page, the additional size is free anyway. Consider again the examples in Figures 5-2 and 5-3. The *cust_id* is declared as a *smallint*, meaning that its maximum positive value is 32,767 (unfortunately, SQL Server doesn't provide any unsigned *int* or unsigned *smallint* data types), and it consumes 2 bytes of storage. Although 32,767 customers might seem like a lot to a new company, the company might be surprised by its own success and find in a couple of years that 32,767 is too limited.

The database designers might regret that they tried to save 2 bytes and didn't simply make the data type an *int*, using 4 bytes but with a maximum positive value of 2,147,483,647. They'll be especially disappointed if they realize they didn't really save any space. If you compute the rows-per-page calculations just discussed, increasing the row size by 2 bytes, you'll see that the same number of rows still fit on a page. The additional 2 bytes are free—they were simply wasted space before. They never cause fewer rows per page in the fixed-length example, and they'll rarely cause fewer rows per page even in the variable-length case.

So which strategy wins? Potentially better update performance? Or more rows per page? Like most questions of this nature, no one answer is right. It depends on your application. If you understand the tradeoffs, you can make the best choice. Now that you know the issues, this general rule merits repeating: Variable-length data types are most appropriate when you expect significant variation in the size of the data for that column and when the column won't be updated frequently.

Character Data Collation

For many data types, the rules to compare and sort are straightforward. No matter whom you ask, 12 is always greater than 11, and even if people may write dates in different ways, August 20, 2008, is never the same as August 21, 2007. But for character data, this principle doesn't apply. Most people would sort *csak* before *cukor*, but in an Hungarian dictionary, they come in the opposite order. And is *STREET* equal to *Street* or not? Also, how are characters with diacritic marks, such as accents or umlauts, sorted?

Because different users have different preferences and needs, character data in SQL Server are always associated with a *collation*. A collation is a set of rules that defines how character data are sorted and compared, and how language-dependent functions such as *UPPER* and *LOWER* work. The collation also determines the character repertoire for the single-byte data types, *char, varchar,* and *text.* Metadata in SQL Server (that is, names of tables, variables, etc.) are also subject to collation rules.

Determining Which Collation to Use You can define which collation to use at several levels in SQL Server. When you create a table, you can define the collation for each character column. If you don't supply a collation, the database collation is used.

The database collation also determines the collation for the metadata in the database. So in a database with a case-insensitive collation, you can use *MyTable* or *MYTABLE* to refer to a table which was created with the name *mytable*, but in a database with a case-sensitive collation, you must refer to it as *mytable*. The database collation also determines the collation for string literals and for data in character variables.

You can specify the database collation when you create a database. If you do not, the server collation is used. Under some fairly restricted circumstances, the *ALTER DATABASE* statement permits you to change the database collation. (Basically, if you have any CHECK constraints in the database, you cannot change the collation.) This will rebuild the system tables to reflect the new collation rules in the metadata. However, columns in user tables are left unchanged, and you need to change these yourself. For details on all restrictions, please see the *ALTER DATABASE* topic in *SQL Server Books Online*.

The server collation is used by the system databases *master, model, tempdb,* and *msdb*. (The resource database, on the other hand, always has the same collation, Latin1_General_CI_AI.) The server collation is also the collation for variable names, so on a server with a case-insensitive collation, @a and @A are the same variable, but they are two different ones if the server collation is case-sensitive. You select the server collation at setup.

Finally, you can use the COLLATE clause to force the collation in an expression. One situation where you need to do this is when the same expression includes two columns with different collations. This results in a *collation conflict*, and SQL Server requires you to resolve it with the COLLATE clause.

Available Collations To see the available collations, you can run the query

```
SELECT * FROM fn_helpcollations();
```

When running this query on a SQL Server 2008 instance, the result contains 2,397 collations. There are another 112 collations that are deprecated and not listed by fn_helpcollations.

Collations fall into two main groups: Windows collations and SQL Server collations. SQL Server collations are mainly former collations retained for compatibility reasons. Nevertheless, the collation SQL_Latin1_General_CP1_CI_AS is one of the most commonly used ones because it is the default collation when you install SQL Server on a machine with English (United States) as the system locale.

Windows Collations Windows collations take their definition from Microsoft Windows. SQL Server does not go out and query Windows for collation rules; rather, the SQL Server team has copied the collation definitions into SQL Server. The collations in Windows typically are modified with new releases of Windows to adapt to changes in the Unicode standard, and because collations determine in which order data appear in indexes, SQL Server cannot accept that the definition of a collation changes because you move a database to a different Windows version.

The Anatomy of a Collation Name Windows collations come in families, with 18 collations in each family. All collations in the same family start with the same *collation designator*, which indicates which language or group of languages the collation family supports.

The collation designator is followed by tokens that indicate the nature of the collation. The collation can be a binary collation, in which case the token is BIN or BIN2. For the other 16 collations, the tokens are CI/CS to indicate case sensitivity/insensitivity, AI/AS to indicate accent sensitivity/insensitivity, KS to indicate kanatype sensitivity, and WS to indicate width sensitivity.

If CI is part of the collation name, the strings *smith* and *SMITH* are equal, but they are different if CS is in the name. Likewise, if the collation is AI, *cote*, *coté, côte,* and *côté* are all equal, but in an AS collation, they are different. Kanatype relates to Japanese text only, and in a kanatype-sensitive collation, katakana and hiragana counterparts are considered different. Width sensitivity refers to East Asian languages for which there exists both half-width and full-width forms of some characters. KI and WI tokens do not exist, but kanatype and width insensitivity are implied if KS and WS are absent.

The following are some examples of collation names:

- **Latin1_General_CI_AS** A case-insensitive, accent-sensitive collation for Western European languages such as English, German, and Italian
- **Finnish_Swedish_CS_AS** A case-sensitive and accent-sensitive collation for Finnish and Swedish
- **Japanese_CI_AI_KS_WS** A collation that is insensitive to case and accent and sensitive to kanatype and width differences
- **Turkish_BIN2** A binary collation for Turkish

Different Versions of the Same Collation A collation designator may include a version number that indicates in which version of SQL Server the collation was added. The lack of a version number means that the collation was one of the original collations in SQL Server 2000; 90 indicates that the collation was added in SQL Server 2005; and 100 means that it was added in SQL Server 2008.

SQL Server 2008 added new collations for languages and language groups for which a collation already existed. So there is now both Latin1_General and Latin1_General_100, Finnish_Swedish and Finnish_Swedish_100, and other collation pairs.

These additions reflect the changes in Windows. The old collations are based on the collations in Windows 2000, and the new _100 collations are based on the collations in Windows 2008.

> **Caution** If you plan to access your SQL Server 2008 instance as a linked server from SQL Server 2005, you should avoid using the new _100 collations because if you try to access such a column from SQL Server 2005, you get the error message, "An invalid tabular data stream (TDS) collation was encountered."

The Single-Byte Character Types The single-byte character data types, *char, varchar,* and *text,* can represent only 255 possible characters, and the *code page* of the collation determines which 255 characters are available. In most code pages, the characters from 32 to 127 are always the same, taken from the ASCII standard, and remaining characters are selected to fit a certain language area. For instance, CP1252, also known as Latin-1, supports Western European languages such as English, French, Swedish, and others. CP1250 is for the Cyrillic script, CP1251 is for Eastern European languages, and so on.

When it comes to other operations—sorting, comparing, lower/upper, and so on—in a Windows collation, the rules are exactly the same for the single-byte data types and the double-byte Unicode data types. There is one exception to this: in a binary collation, sorting is done by character codes, and the order in the single-byte code page can be different from the order in Unicode. For instance, in a Polish collation, *char(209)* prints Ń (a capital N with an acute accent), whereas *unicode(N'Ń')* prints 323, which is the code point in Unicode for this character. (The code points in Unicode agree with the code points in Latin-1, but that applies only to the range 160-255.) Microsoft has added some extra characters to their version of Latin-1. One example of this is the Euro(€) character, which is *char(128)* in a collation based on CP1252, but in

Unicode, code point 128 is a nonprinting character, and *unicode(N'€')* prints 8364.

There are some collations that do not map to a single-byte code page. You can use these collations only with Unicode data types. For instance, if you run the code

```
CREATE TABLE NepaleseTest
    (abc char(5) COLLATE Nepali_100_CI_AS NOT NULL);
```

you get the following error message:

```
Msg 459, Level 16, State 2, Line 1
Collation 'Nepali_100_CI_AS' is supported on Unicode data types only and cannot be applied
to char, varchar or text data types.
```

To view the code page for a collation, you can use the *collationproperty* function, as in this example:

```
SELECT collationproperty('Latin1_General_CS_AS', 'CodePage');
```

This returns 1252. For a collation that supports Unicode only, you get 0 in return. (If you get NULL back, you have misspelled the collation name or the word *CodePage.*)

You cannot use Unicode-only collations as the server collation.

Sort Order The collation determines the sort order. When a Windows collation is insensitive (such as case or accents), this also applies to the sort order. For instance, in a case-insensitive collation, differences in case do not affect how the data is sorted. In a sensitive collation, case, accent, kanatype, and width affect the sorting, but only with a secondary weight. That is, these properties affect the sorting only when no other differences exist.

To illustrate this, consider this table:

```
CREATE TABLE #words (word   nvarchar(20) NOT NULL,
                     wordno tinyint PRIMARY KEY CLUSTERED);
INSERT #words
   VALUES(N'cloud',  1), (N'CSAK',    6), (N'cukor',   11),
         (N'Oblige', 2), (N'Opera',   7), (N'Öl',      12),
         (N'résumé', 3), (N'RESUME',  8), (N'RÉSUMÉ',  13),
         (N'resume', 4), (N'resumes', 9), (N'résumés', 14),
         (N'ŒIL',    5), (N'œil',    10);
```

To examine how a collation works, we use the query shown here. We start by looking at the commonly used collation Latin1_General_CI_AS:

```
WITH collatedwords (collatedword, wordno) AS (
   SELECT word COLLATE Latin1_General_CI_AS, wordno
   FROM   #words
)
SELECT collatedword, rank = dense_rank() OVER(ORDER BY collatedword),
       wordno
FROM   collatedwords
ORDER  BY collatedword;
```

When I ran the query, I got this result:

```
collatedword   rank   wordno
-------------- ------ ------
cloud          1      1
CSAK           2      6
cukor          3      11
Oblige         4      2
ŒIL            5      5
œil            5      10
Öl             6      12
Opera          7      7
RESUME         8      8
resume         8      4
résumé         9      3
RÉSUMÉ         9      13
resumes        10     9
résumés        11     14
```

The *rank* column gives the ranking in the sort order. We can see that for the words that differ only in case, the ranking is the same. We can also see from the output that sometimes the uppercase version comes first, and sometimes the lowercase version comes first. This is something that is entirely arbitrary, and it's perfectly possible that you will see a different order for these pairs if you run the query yourself.

If we change the collation to Latin1_General_CS_AS, we get this result:

```
collatedword   rank   wordno
------------- ------ ------
cloud          1      1
CSAK           2      6
cukor          3      11
Oblige         4      2
œil            5      10
ŒIL            6      5
Öl             7      12
Opera          8      7
resume         9      4
RESUME         10     8
résumé         11     3
RÉSUMÉ         12     13
resumes        13     9
résumés        14     14
```

All entries now have a different ranking. The lowercase forms come before the uppercase forms when no other difference exists because in Windows collations, lowercase always has a lower secondary weight than uppercase.

Let's now see what happens with a different language. Here's a test for the collation Hungarian_CI_AI:

```
collatedword   rank   wordno
------------- ------ ------
cloud          1      1
cukor          2      11
CSAK           3      6
Oblige         4      2
ŒIL            5      5
œil            5      10
Opera          6      7
Öl             7      12
RÉSUMÉ         8      13
RESUME         8      8
résumé         8      3
resume         8      4
resumes        9      9
résumés        9      14
```

The words *CSAK* and *öl* now sort after *cukor* and *Opera*. This is because in the Hungarian alphabet, CS and Ö are letters on their own. You can also see that in this CI_AI collation, all four forms of *résumé* have the same rank.

In these examples, the data type for the column was *nvarchar,* but if you change the table to use *varchar* and rerun the examples, you get the same results.

Character Ranges and Collations The sort order applies not only to ORDER BY clauses, but also to operators such as > and ranges in LIKE expressions. For instance, note the following code:

```
SELECT * FROM #words
WHERE word COLLATE Latin1_General_CI_AS > 'opera';
SELECT * FROM #words
WHERE word COLLATE Latin1_General_CS_AS > 'opera';
```

The first *SELECT* lists six words, whereas the second lists seven (because in a case-sensitive collation, *Opera* is > *opera*).

If you are used to character ranges from regular expressions in other languages, you may fall into the following trap when trying to select the words that start with an uppercase letter:

```
SELECT * FROM #words WHERE word LIKE '[A-Z]%';
```

But even with a case-sensitive collation, this code usually lists all 14 words. (In some languages, *Ö* sorts as a separate letter after *Z*, so it does not fall into the specified range.) The range *A–Z* is also subject to the collation rules. This also has another consequence: change *cloud* to *aloud* in the list. Using a case-sensitive collation, *SELECT* now returns only 13 rows. Because *a* sorts before *A*, the range *A–Z* does not include *a*.

As you can see, this can be a bit confusing. My advice is that you be very careful when using ranges with character data. If you need to do it, make sure that you really test the edge cases to ensure that you don't exclude any data inadvertently.

Binary Collations In a binary collation, no secondary weights exist, and characters sort by their code points in the character set. So with Latin1_General_BIN2 in the previous example, we get

```
collatedword   rank   wordno
-------------- ------ ------
CSAK           1      6
Oblige         2      2
Opera          3      7
RESUME         4      8
RÉSUMÉ         5      13
cloud          6      1
cukor          7      11
resume         8      4
resumes        9      9
résumé         10     3
résumés        11     14
Öl             12     12
ŒIL            13     5
œil            14     10
```

Now the words with the uppercase first letters *C*, *O,* and *R* come before those with the lowercase *c*, *o,* and *r,* as they do in the ASCII standard. *Öl* and the two forms of *œil* have code points beyond the first 127 ASCII codes and therefore come at the end of the list.

Because binary collations are based on the code points and they may be different in the single-byte code page and in Unicode, the order can be different for single-byte and Unicode data types. For instance, if you change the data type in #words to *varchar* and run the example with Latin1_General_BIN2 again, you find that *Öl* now comes last.

As you recall from the previous discussion, two types of binary collations exist, BIN and BIN2. Of these, the BIN collations are earlier collations, and if you need to use a binary collation in new development, you should use a BIN2 collation. To understand the difference between the two, we need to look at a Unicode string in its binary representation. For instance, consider

```
SELECT convert(varbinary, N'ABC');
```

This code returns 0x410042004300. The ASCII code for *A* is 65, or 41 in hexadecimal. And in Unicode, *A* is U+0041. (Unicode characters are often written as U+XXXX, where XXXX is the code point in hexadecimal notation.) But converted to *varbinary*, it appears as 4100. This is because PC architecture is *little endian*, which means that the least significant byte is stored first. (The reason for this is beyond the scope of this book to explain.)

Therefore, to sort *nvarchar* data by their code points properly, SQL Server should not just look at the byte string but swap each word to get the correct code points. And this is exactly what the BIN2 collations do. The older BIN collations perform this swap only for the first character, and then perform a byte-per-byte comparison for remaining characters. To illustrate the difference between the two types of binary collations and also true byte-sort, here is an example where we use the characters *Z* (U+005A) and *Ń* (N with grave accent; U+0143):

```
SELECT n, str, convert(binary(6), str) AS bytestr,
       row_number() OVER(ORDER BY convert(varbinary, str))
          AS bytesort,
       row_number() OVER(ORDER BY str COLLATE Latin1_General_BIN)
          AS collate_BIN,
       row_number() OVER(ORDER BY str COLLATE Latin1_General_BIN2)
          AS collate_BIN2
FROM  (VALUES(1, N'ZZZ'), (2, N'ZŃŃ'), (3, N'ŃZZ'), (4, N'ŃŃŃ'))
      AS T(n, str)
ORDER BY n;
```

Here is the result:

```
n           str    bytestr          bytesort    collate_BIN    collate_BIN2
----------- ------ ---------------- ----------- -------------  -----------
1           ZZZ    0x5A005A005A00   4           2              1
2           ZŃŃ    0x5A0043014301   3           1              2
```

```
3           ŃZZ    0x43015A005A00   2           4              3
4           ŃŃŃ    0x430143014301   1           3              4
```

You can see that in the *collate_BIN2* column, the rows are numbered according to their code points in Unicode. In the *bytesort* column, on the other hand, they are numbered in reverse order because the least significant byte in the character code takes precedence. Finally, in the *collate_BIN* column, the two entries that start with *Z* are sorted first, but in reverse order with regards to *collate_BIN2*.

SQL Server Collations The SQL Server collations (known as *SQL collations* for short) is a much smaller group than the Windows collations. In total, there are 76 SQL collations, of which 1 is deprecated.

A SQL collation uses two different rule sets. One is for single-byte data types, and the other is for Unicode data types. The rules for single-byte data types are defined by SQL Server itself, and derive from the days when SQL Server did not support Unicode. When you work with Unicode data, a SQL collation uses the same rules as the matching Windows collation. To see which Windows collation a certain SQL collation matches, you can view the *description* column in the output from *fn_helpcollations()*.

The name of a SQL collation always starts with *SQL_* followed by a language indicator, similar to the name of a Windows collation. Likewise, names for SQL collations include *CI/CS* and *AI/AS* to indicate case and accent sensitivity. Some binary SQL collations also exist. In contrast to names for Windows collations, SQL collations always include the code page for single-byte characters in the name. For some reason, though, CP1252, Windows Latin-1, appears as CP1 in the names.

Many SQL collations relate to American National Standards Institute (ANSI) code pages, that is, code pages used by non-Unicode Windows applications. But there are also SQL collations for the OEM code pages CP437 and CP850; that is, code pages used in the command-line window. There are even a few SQL collations for EBCDIC.

Sort Orders With a SQL collation, you can get different results depending on the data type. For instance, in the example with the 14 words, if we run it with *word* as *nvarchar* and with the commonly used SQL collation SQL_Latin1_General_CP1_CI_AS, the result is the same as when we used Latin1_General_CI_AS. But if you change *word* to be *varchar*, you get this result:

```
collatedword   rank   wordno
------------- ------ ------
ŒIL            1      5
œil            2      10
cloud          3      1
CSAK           4      6
cukor          5      11
Oblige         6      2
Öl             7      12
Opera          8      7
RESUME         9      8
resume         9      4
résumé         10     3
RÉSUMÉ         10     13
resumes        11     9
résumés        12     14
```

Now the two forms of *œil* come first and they have different ranks, despite the collation being case insensitive. In this collation, a few accented letters sort as if they were punctuation characters. (The others are *Š*, *Ÿ,* and *Ž*.) Other differences in SQL_Latin1_General_CP1_CI_AS between the single-byte and Unicode data types include how punctuation characters are sorted. However, so long as your data mainly consist of the digits 0–9 and the English letters A–Z, these differences likely will not be significant to you.

Tertiary Collations Just like Windows collations, SQL collations have primary and secondary weights, but it does not stop there. A total of 32 of the SQL collations also have *tertiary weights.* With one exception, the tertiary collations are all case insensitive. The purpose of the tertiary weight is to give preference to uppercase, so when everything else is equal in the entire ORDER BY clause, uppercase words sort first. In some tertiary collations, this is indicated by *Pref* appearing in the name, whereas in other tertiary collations, this is implicit. You find the full list of tertiary collations in *SQL Server Books Online* in the topic for the built-in function *TERTIARY_WEIGHTS*.

To study the tertiary collations, we use a different table with different words as follows:

```
CREATE TABLE #prefwords
           (word   char(3) COLLATE SQL_Latin1_General_Pref_CP1_CI_AS
                              NOT NULL,
              wordno int NOT NULL PRIMARY KEY NONCLUSTERED,
              tert   AS tertiary_weights(word));
CREATE CLUSTERED INDEX word_ix ON #prefwords (word);
--CREATE INDEX tert_ix on #prefwords(word, tert)
go
INSERT #prefwords (word, wordno)
   VALUES ('abc', 1), ('abC', 4), ('aBc', 7),
          ('aBC', 2), ('Abc', 5), ('ABc', 8),
          ('AbC', 3), ('ABC', 6);
go
SELECT word, wordno, rank = dense_rank() OVER(ORDER BY word),
       rowno = row_number() OVER (ORDER BY word)
FROM   #prefwords
ORDER  BY word--, wordno;
```

The output from this query is

```
word   wordno   rank   rowno
------ -------- ------ -----
ABC    6        1      8
ABc    8        1      6
AbC    3        1      7
Abc    5        1      5
aBC    2        1      4
aBc    7        1      3
abC    4        1      2
abc    1        1      1
```

You can see that all words have the same rank; nevertheless, uppercase letters consistently come before lowercase. And in the *rowno* column, rows are numbered in opposite order, which is likely to be by chance. That is, the tertiary weight affects only the ORDER BY at the end of the query, but not the ORDER BY for the *dense_rank* and *row_number* functions.

Now, if you look at the query plan for this query, you find a Sort operator, which is surprising, given there is a clustered index on *word*. If you go one step back in the plan, you find a Compute Scalar operator, and if you press F4, you can see that this operator defines [Expr1005] = Scalar Operator(tertiary_weights([tempdb].[dbo].[#prefwords].[word])), and if you look at the Sort operator, you see that it sorts by *word* and Expr1005. That is, the tertiary weight is not stored in the index, but computed at run time.

This is where the function *TERTIARY_WEIGHTS* comes in. This function accepts parameters of the types *char, varchar,* and *text* and returns a non-NULL value if the input value is not from a tertiary collation. *SQL Server Books Online* suggests that you can add a computed column with this function and then add an index on the character column and the computed column, like the *tert_ix* in the previous script. If you uncomment the creation of *tert_ix* in the previous script and also comment out the *rank* and *rowno* columns from the *SELECT* statement, you see a plan without any Sort operator. Thus the function *TERTIARY_WEIGHTS* can help to improve performance with tertiary collations.

Now see what happens if we uncomment *wordno* from the ORDER BY clause, so that the query now reads:

```
SELECT word, wordno
FROM   #prefwords
ORDER  BY word, wordno;
```

This is the output:

```
word   wordno
------ ------
abc    1
aBC    2
AbC    3
abC    4
Abc    5
ABC    6
aBc    7
ABc    8
```

That is, the tertiary weight only matters when there is no other difference in the entire ORDER BY clause. Needless to say, the query plan again includes the Sort operator.

Collations Defined During SQL Server Setup When you install SQL Server, you need to select a server collation. This is an important choice, because if you make an incorrect selection, you cannot easily change this later. You will essentially have to reinstall SQL Server.

The SQL Server Setup provides a default collation, and this will always be a CI_AS collation—that is, a collation that is sensitive to accents but insensitive to case, kanatype, and width.

Setup selects the collation designator for the default collation from the *system locale*—that is, the locale that applies on the system level, which may be different from the regional settings for your own Windows user. The default is always a Windows collation, except in one very notable case: if your system locale is English (United States), the default is SQL_Latin1_General_CP1_CI_AS. The reason for this seemingly odd default is backward compatibility.

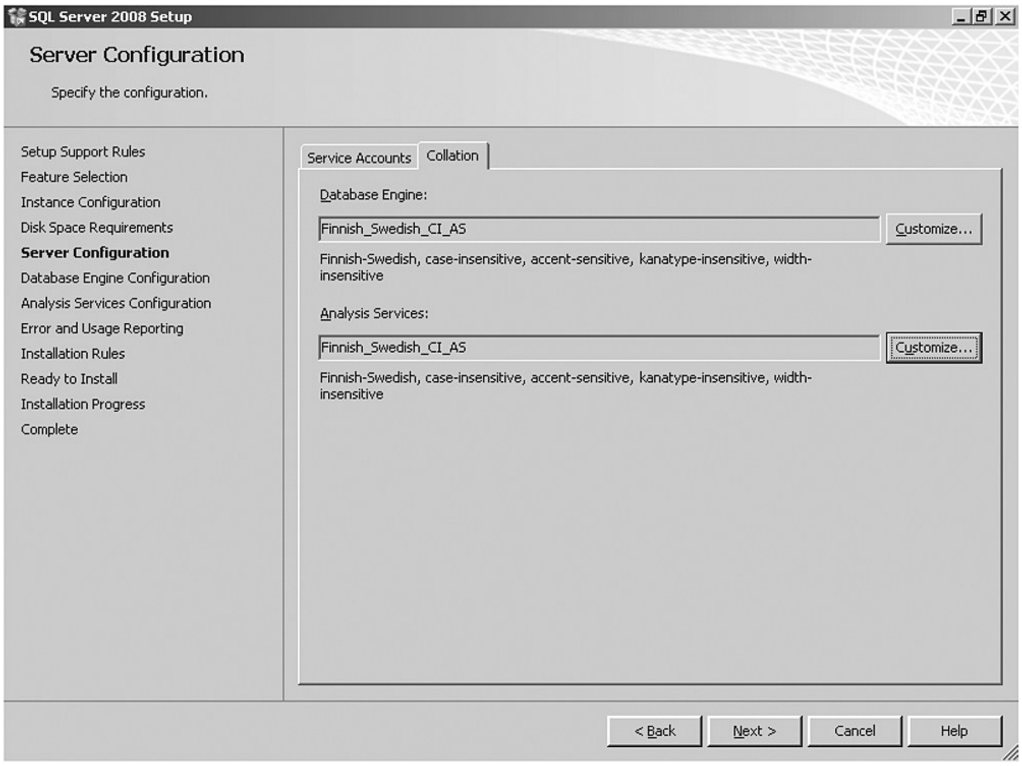
When different versions of the same language exist, the default depends on whether your system locale existed in previous versions of Windows or was added in Windows 2008. So, for instance, for English (United Kingdom) and German (Germany) the default is Latin1_General_ CI_AS, whereas for English (Singapore) and Swahili (Kenya), the default is Latin1_General_100_ CI_AS. Again, the reason for this variation is backward compatibility. For the full list of default collations, see the topic “Collation Settings in Setup” in *SQL Server Books Online*.

Although Setup suggests a default collation, it is far from certain that this default is the best for your server. You should make a conscious, deliberate decision. If you install a server to run a third-party product, you should consult the vendor’s documentation to see if it has any recommendations or requirements for the application. If you plan to migrate databases from an earlier version of SQL Server, you should probably select the same collation for the new server as for your existing server. As I noted earlier, if you plan to access the server as a linked server from SQL Server 2005, you should avoid the new _100 collations.

Another thing to beware of is that your Windows administrator may have installed a U.S. English version of Windows, leaving the system locale as English (United States) even if the local language is something else. If this is the case on your server, and you do not pay attention when you install SQL Server, you may end up with a collation that does not fit well with the language in your country.

Some languages have multiple appropriate choices. For instance, for German, the default is Latin1_General_CI_AS, but you can also use any of the German_Phonebook collations (in which *ä, ö,* and *ü* sort as *ae, oe,* and *ue*).

Running the Installation Wizard When you run the Installation Wizard for SQL Server 2008, you need to be observant because the collation selection is not on a page of its own but appears on a second tab on the Server Configuration page. You’ll have to watch carefully because the collation tab is not displayed when you get to the *Server Configuration* screen. You’ll see a screen asking for information about the service accounts to use. When you select the *collation* tab on that screen, you see something like Figure 5-4.



Database Engine:

Finnish_Swedish_CI_AS

Finnish-Swedish, case-insensitive, accent-sensitive, kanatype-insensitive, width-insensitive

Analysis Services:

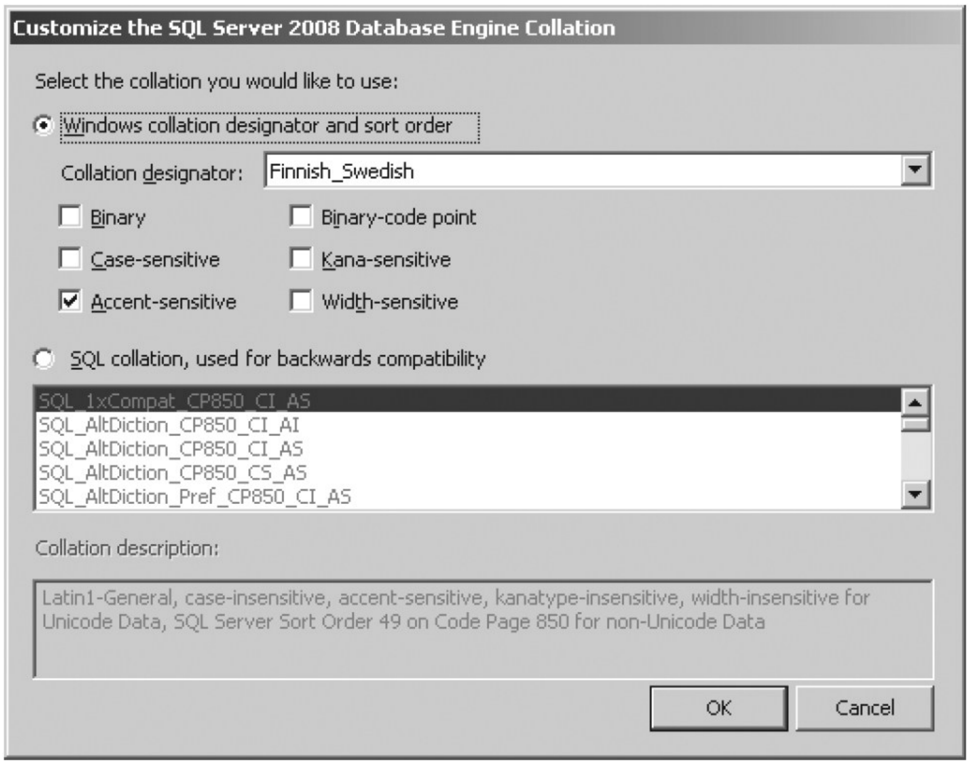
Finnish_Swedish_CI_AS

Finnish-Swedish, case-insensitive, accent-sensitive, kanatype-insensitive, width-insensitive

Figure 5-4: Setting the server configuration

This screenshot was taken on a machine with the system locale set to Swedish, and thus the default collation is Finnish_Swedish_CI_AS. (As you can see, you can also set the collation for Analysis Services on this tab, but that is beyond the scope of this book.)

Figure 5-5 shows you what you see when you press Customize.



Select the collation you would like to use:

[✓ Windows collation designator and sort order]

Collation designator: Finnish_Swedish

☐ Binary ☐ Binary-code point

☐ Case-sensitive ☐ Kana-sensitive

☑ Accent-sensitive ☐ Width-sensitive

SQL collation, used for backwards compatibility

SQL_1xCompat_CP850_CI_AS
SQL_AltDiction_CP850_CI_AI
SQL_AltDiction_CP850_CI_AS
SQL_AltDiction_CP850_CS_AS
SQL_AltDiction_Pref_CP850_CI_AS

Collation description:

Latin1-General, case-insensitive, accent-sensitive, kanatype-insensitive, width-insensitive for Unicode Data, SQL Server Sort Order 49 on Code Page 850 for non-Unicode Data

Figure 5-5: Customize the collation properties

You can use an option button to select whether to use a Windows collation or a SQL collation. If you select a Windows collation, there is a drop-down list where you can select the Collation designator. Below that are check boxes to select case sensitivity and other features. The choice *Binary* gives you a BIN collation, whereas *Binary-code point* gives you a BIN2 collation. If you select to use a SQL collation, there is a single list box that lists all SQL collations.

Performance Considerations Does the choice of collation affect performance? Yes, but in many cases only marginally, and your most important criteria should be to choose the collation that best meet your users' needs. However, there are a few situations where the collation can have quite drastic effects.

Generally, binary collations give you the best performance, but in most applications, they do not give a very good user experience.

So long as you work with *varchar* data, the SQL collations perform almost equally well. The SQL collations include rules only for the 255 characters in the code page covered by the collation. A Windows collation always works with the full rules of Unicode internally, even for single-byte data. Thus, the internal routines for SQL collations are far less complex than those for Unicode.

The Windows collations have some differences between collation families where some are faster than others. A special case is the case-insensitive Latin1_General and Latin1_General_100 collations, which appear to perform better than any other collation family when you work with Unicode data. Contrary to what you may expect, case-sensitive collations do not give better performance; rather, their rate is a few percentage points slower in many operations. But, again, this is not something that you should pay too much attention to. If your users expect to see data sorted according to, say, the Danish alphabet, there is no reason to select Latin1_General_CI_AS just because it operates a little faster. What's the point of a faster operation that doesn't do what your users need? Also, keep in mind that a typical query includes so many other components that the effect of the collation is likely to be lost in the noise.

A Trap with SQL Server Collations The collation really does matter in a few situations, though. Consider the following:

```
SELECT col FROM tbl WHERE indexedcol = @value;
```

For this query, the collation does not have much impact so long as the column and @value has the same data type. Neither is there an issue, if the column has a Unicode data type and @value is *char* or *varchar*. But if the column is single-byte and @value is Unicode, there is an issue because the data-type precedence rules in SQL Server. The *char* and *varchar* data types have lower precedence than *nchar* and *nvarchar*, so the column is converted to the type of the value, and this has ramifications for how the index can be used.

If the column has a Windows collation, SQL Server can still seek the index, albeit with a more complex filter, so compared to a query without conversion, you can expect the execution time to double or triple. But it is when the column has a SQL collation that this query becomes really problematic. The index does not serve any purpose after the conversion because in a SQL collation, the rules are entirely different for single-byte and Unicode data. SQL Server can at best scan the index. In a big table, performance can be drastically affected, with execution times that are 100 or 1,000 times more than for a properly written query. Thus, if you opt to use a SQL collation, you need to watch that you don't mix *varchar* and *nvarchar* casually.

Another case where the collation can make a huge difference is when SQL Server has to look at almost all characters in the strings. For instance, look at the following:

```
SELECT COUNT(*) FROM tbl WHERE longcol LIKE '%abc%';
```

This may execute 10 times faster or more with a binary collation than a nonbinary Windows collation. And with *varchar* data, this executes up to seven or eight times faster with a SQL collation than with a Windows collation. If you have a *varchar* column, you can speed this up by forcing the collation as follows:

```
SELECT COUNT(*) FROM tbl
WHERE longcol COLLATE SQL_Latin1_General_CP_CI_AS LIKE '%abc%';
```

If your column is *nvarchar*, you have to force a binary collation instead, but that would only be possible if users can accept a case-sensitive search.

The same considerations apply to the functions *CHARINDEX* and *PATINDEX.*

Special Data Types

I'll end this section on data types by showing you a few additional data types that you might find useful.

Binary Data Types These data types are *binary* and *varbinary*. They are used to store strings of bits, and the values are entered and displayed using their hexadecimal (hex) representation, which is indicated by a prefix of *0x*. So a hex value of 0x270F corresponds to a decimal value of 9,999 and a bit string of 0010011100001111. In hex, each two displayed characters represent a byte, so the value of 0x270F represents 2 bytes. You need to decide whether you want your data to be fixed or variable length, and you can use some of the same considerations discussed previously for deciding between *char* and *varchar* to make your decision. The maximum length of *binary* or *varbinary* data is 8,000 bytes.

bit Data Type The *bit* data type can store a 0 or a 1 and can consume only a single bit of storage space. However, if there is only one bit column in a table, it will take up a whole byte. Up to eight-bit columns are stored in a single byte.

LOB Data Types SQL Server 2008 allows you to define columns with the *MAX* attribute: *varchar(MAX), nvarchar(MAX),* and *varbinary(MAX).* If the number of bytes actually inserted into these columns exceeds the maximum of 8,000, these columns are stored using a special storage format for LOB data. The special storage format is the same one as used for the data types *text*, *ntext*, and *image,* but because those types will be discontinued in a future version of SQL Server, it is recommend that you use the variable-length data types with the MAX specifier for all new development*.* The *varchar(MAX)* (or *text)* data type can store up to 2^31 – 1 non-Unicode characters, *nvarchar(MAX)* (or *ntext)* can store up to 2^30 – 1 (half as many) Unicode characters, and *varbinary(MAX)* (or *image*) can store up to 2^31 – 1 bytes of binary data. In addition, *varbinary(MAX)* data can be stored as filestream data. We'll cover filestream data in more detail in Chapter 7, as well as look at the storage structures for LOB data.

cursor Data Type The *cursor* data type can hold a reference to a cursor. Although you can't declare a column in a table to be of type *cursor*, this data type can be used for output parameters and local variables. I've included the *cursor* data type in this list for completeness, but I won't be talking more about it.

rowversion Data Type The *rowversion* data type is a synonym for what was formerly called a *timestamp*. When using the *timestamp* data type name, many people might assume that the data has something to do with dates or times, but it doesn't. A column of type *rowversion* holds an internal sequence number that SQL Server automatically updates every time the row is modified. The value of any *rowversion* column is actually unique within an entire database, and a table can have only one column of type *rowversion*. Any operation that modifies any *rowversion* column in the database generates the next sequential value. The actual value stored in a *rowversion* column is seldom important by itself. The column is used to detect whether a row has been modified since the last time it was accessed by determining whether the *rowversion* value has changed.

sql_variant Data Type The *sql_variant* data type allows a column to hold values of any data type except *text*, *ntext*, *image*, *XML*, user-defined data types, variable-length data types with the MAX specifier, or *rowversi*on (*timestamp*). I'll describe the internal storage of *sql_variant* data later in this chapter.

Spatial Data Type SQL Server 2008 provides two data types for storing spatial data. The *geometry* data type supports planar, or Euclidean (flat-earth), data. The *geometry* data type conforms to the Open Geospatial Consortium (OGC) Simple Features for SQL Specification version 1.1.0. The *geography* data type stores ellipsoidal (round-earth) data, such as Global Positioning Satellite (GPS) latitude and longitude coordinates. These data types have their own methods for accessing and manipulating the data, as well as their own special extended index structures, which are different than the normal SQL Server indexes. Any further discussion of the access methods and storage of spatial data is beyond the scope of this book.

table Data Type The *table* data type can be used to store the result of a function and can be used as the data type of local variables. Columns in tables cannot be of type *table*.

xml Data Type The *xml* data type lets you store XML documents and fragments in a SQL Server database. You can use the *xml* data type as a column type when you create a table, or as the data type for variables, parameters, and the return value of a function. XML data has its own methods for retrieval and manipulation. I will not be covering details of working with *xml* data in this book.

uniqueidentifier Data Type The *uniqueidentifier* data type is sometimes referred to as a globally unique identifier (GUID) or universal unique identifier (UUID). A GUID or UUID is a 128-bit (16-byte) value generated in a way that, for all practical purposes, guarantees uniqueness among every networked computer in the world. It is becoming an important way to identify data, objects, software applications, and applets in distributed systems. Because there are some very interesting aspects to the way the *uniqueidentifier* data type is generated and manipulated, I'll give you a bit more detail about it.

The T-SQL language supports the system functions *NEWID* and *NEWSEQUENTIALID*, which you can use to generate *uniqueidentifier* values. A column or variable of data type *uniqueidentifier* can be initialized to a value in one of the following two ways:

- Using the system-supplied function *NEWID* or *NEWSEQUENTIALID* as a default value.
- Using a string constant in the following form (32 hexadecimal digits separated by hyphens): *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*. (Each *x* is a hexadecimal digit in the range 0 through 9 or *a* through *f*.)

This data type can be quite cumbersome to work with, and the only operations that are allowed against a *uniqueidentifier* value are comparisons (=, <>, <, >, <=, >=) and checking for NULL. However, using this data type internally can offer several advantages.

One reason to use the *uniqueidentifier* data type is that the values generated by *NEWID* or *NEWSEQUENTIALID* are guaranteed to be globally unique for any machine on a network because the last six bytes of a *uniqueidentifier* value make up the node number for the machine. When the SQL Server machine does not have an Ethernet/Token Ring (IEEE 802.*x*) address, there is no node number and the generated GUID is guaranteed to be unique among all GUIDs generated on that computer. However, the possibility exists that another computer without an Ethernet/Token Ring address will generate the identical GUID. The GUIDs generated on computers with network addresses are guaranteed to be globally unique.

The primary reason that SQL Server needed a way to generate a GUID was for use in merge replication, in which identifier values for the same table could be generated on any one of many different SQL Server machines. There needed to be a way to determine whether two rows really were the same row and there had to be no way that two rows not referring to the same entity would have the same identifier. Using GUID values provides that functionality. Two rows with the same GUID value must indicate that they really are the same row.

The difference between the *NEWSEQUENTIALID* and the *NEWID* functions is that *NEWSEQUENTIALID* creates a GUID that is greater than any GUID previously generated by this function on a specified computer and can be used to introduce a sequence to your GUID values. This turns out to increase greatly the scalability of systems using merge replication. If the *unqiueidentifer* values are being used as the clustered key for the replicated tables, the new rows are then inserted in random disk pages. (You'll see the details in Chapter 6, when clustered indexes are discussed in detail.) If the machines involved are performing a large amount of I/O operations, the nonsequential GUID generated by the *NEWID* function results in lots of random B-tree lookups and inefficient insert operations. The new function, *NEWSEQUENTIALID*, which is a wrapper around the Windows function *UuidCreateSequential*, does some byte scrambling and creates an ordering to the generated UUID values.

The list of *uniqueidentifier* values can't be exhausted. This is not the case with other data types frequently used as unique identifiers. In fact, SQL Server uses this data type internally for row-level merge replication. A *uniqueidentifier* column can have a special property called *ROWGUIDCOL*; at most, one *uniqueidentifier* column can have this property per table. The *ROWGUIDCOL* property can be specified as part of the column definition in *CREATE TABLE* and *ALTER TABLE ADD column*, or it can be added or dropped for an existing column using *ALTER TABLE ALTER COLUMN*.

You can reference a *uniqueidentifier* column with the *ROWGUIDCOL* property using the keyword *ROWGUIDCOL* in a query. This is similar to referencing an identity column using the *IDENTITYCOL* keyword. The *ROWGUIDCOL* property does not imply any automatic value generation, and if automatic value generation is needed, the *NEWID* function should be defined as the default value of the column. You can have multiple *uniqueidentifier* columns per table, but only one of them can have the *ROWGUIDCOL* property. You can use the *uniqueidentifier* data type for whatever reason you come up with, but if you're using one to identify the current row, an application must have a generic way to ask for it without needing to know the column name. That's what the *ROWGUIDCOL* property does.

Much Ado About NULL

The issue of whether to allow NULL has become a heated debate for many in the industry, and the discussion here may outrage a few people. However, my intention isn't to engage in a philosophical debate. Pragmatically, dealing with NULL brings added complexity to the storage engine because SQL Server keeps a special bitmap in every row to indicate which nullable columns actually *are* NULL. If NULLs are allowed, SQL Server must decode this bitmap for every row accessed. Allowing NULL also adds complexity in application code, which can often lead to bugs. You must always add special logic to account for the case of NULL.

As the database designer, you might understand the nuances of NULL and three-valued logic in aggregate functions when you do joins and when you search by values. In addition, you must also consider whether your development staff really understands how to work with NULLs. I recommend, if possible, that you use all NOT NULL columns and define *default* values for missing or unknown entries (and possibly make such character columns *varchar* if the default value is significantly different in size from the typical entered value).

In any case, it's good practice to declare NOT NULL or NULL explicitly when you create a table. If no such declaration exists, SQL Server assumes NOT NULL. (In other words, no NULLs are allowed.) This might surprise many people who assume that the default for SQL Server is to allow NULLs. The reason for this misconception is that most of the tools and interfaces for working with SQL Server enable a session setting that makes it the default to allow NULLs. However, you can set the default to allow NULLs by using a session setting or a database option, which, as I just mentioned, is what most tools and interfaces already do. If you script your DDL and then run it against another server that has a different default setting, you get different results if you don't declare NULL or NOT NULL explicitly in the column definition.

Several database options and session settings can control the behavior of SQL Server regarding NULL values. You can set database options using the *ALTER DATABASE* command, as I showed you in Chapter 3, "Databases and Database Files." And you can enable session settings for one connection at a time using the *SET* command.

Note The database option *ANSI null default* corresponds to the two session settings ANSI_ NULL_DFLT_ON and ANSI_NULL_DFLT_OFF. When the *ANSI null default* database option is false (the default setting for SQL Server), new columns created with the *ALTER TABLE* and *CREATE TABLE* commands are, by default, NOT NULL if the nullability status of the column isn't explicitly specified. SET ANSI_NULL_DFLT_OFF and SET ANSI_NULL_DFLT_ON are mutually exclusive options that indicate whether the database option should be overridden. When on, each option forces the opposite option off. Neither option, when off, turns the opposite option on—it only discontinues the current on setting.

You use the function *GETANSINULL* to determine the default nullability for your current session. This function returns 1 when new columns allow null values and the column or data type nullability wasn't defined explicitly when the table was created or altered. I strongly recommend declaring NULL or NOT NULL explicitly when you create a column. This removes all ambiguity and ensures that you're in control of how the table is built, regardless of the default nullability setting.

The database option *concat null yields null* corresponds to the session setting *SET CONCAT_ NULL_YIELDS_NULL.* When *CONCAT_NULL_YIELDS_NULL* is on, concatenating a NULL value with a string yields a NULL result. For example, *SELECT* 'abc' + NULL yields NULL. When *SET CONCAT_NULL_YIELDS_NULL* is off, concatenating a NULL value with a string yields the string itself. In other words, the NULL value is treated as an empty string. For example, *SELECT* 'abc' + NULL yields *abc*. If the session-level setting isn't specified, the value of the database option *concat null yields null* applies.

The database option *ANSI nulls* corresponds to the session setting *SET ANSI_NULLS.* When this option is set to ON, all comparisons to a NULL value evaluate to UNKNOWN. When it is set to OFF, comparisons of values to a NULL value evaluate to TRUE if both values are NULL. In addition, when this option is set to ON, your code must use the condition IS NULL to determine whether a column has a NULL value. When this option is set to OFF, SQL Server allows = NULL as a synonym for IS NULL and <> NULL as a synonym for IS NOT NULL.

A fourth session setting is *ANSI_DEFAULTS.* Setting this to ON is a shortcut for enabling both *ANSI_NULLS* and *ANSI_NULL_DFLT_ON,* as well as other session settings not related to NULL handling. The SQL Server ODBC driver and the SQL Server OLE DB provider automatically set *ANSI_DEFAULTS* to ON. You can change the *ANSI_NULLS* setting when you define your data source name (DSN). You should be aware that the tool you are using to connect to SQL Server might set certain options ON or OFF*.*

The following query shows the values for all the SET options in your current session, and if you have VIEW SERVER STATE permission, you can change or remove the WHERE clause to return information about other sessions as follows:

```
SELECT * FROM sys.dm_exec_sessions
WHERE session_id = @@spid;
```

As you can see, you can configure and control the treatment and behavior of NULL values in several ways, and you might think it would be impossible to keep track of all the variations. If you try to control every aspect of NULL handling separately within each individual session, you can cause immeasurable confusion and even grief. However, most of the issues become moot if you follow a few basic recommendations:

- Never allow NULL values in your tables.
- Include a specific NOT NULL qualification in your table definitions.
- Don't rely on database properties to control the behavior of NULL values.

If you must use NULLs in some cases, you can minimize problems by always following the same rules, and the easiest rules to follow are the ones that ANSI already specifies.

In addition, certain database designs allow for NULL values in a large number of columns and in a large number of rows. SQL Server 2008 introduces the concept of sparse columns. Sparse columns reduce the space requirements for NULL values at the cost of more overhead to retrieve NOT NULL values. So the biggest benefit from sparse columns is found when a large percentage of your data is NULL. I'll discuss sparse column storage in Chapter 7.

There are a couple of other storage considerations to be aware of when allowing your columns to be NULL. For fixed-length columns (that are not defined to be sparse), the column always uses the full defined length, even when storing NULL. For example, a column defined as *char(200)* always uses 200 bytes whether it is NULL or not. Variable-length columns are different and do not take up any space for the actual data storage of NULLs. That doesn't mean there is no space requirement at all, as we'll see later in this chapter when I describe the internal storage mechanisms.

User-Defined Data Types

A user-defined data type (UDT) provides a convenient way for you to guarantee consistent use of underlying native data types for columns known to have the same domain of possible values. For example, perhaps your database stores various phone numbers in many tables. Although no single, definitive way exists to store phone numbers, consistency is important in this database. You can create a *phone_number* UDT and use it consistently for any column in any table that keeps track of phone numbers to ensure that they all use the same data type. Here's how to create this UDT:

```
CREATE TYPE phone_number FROM varchar(20) NOT NULL;
```

And here's how to use the new UDT when you create a table:

```
CREATE TABLE customer
(
cust_id             smallint        NOT NULL,
cust_name           varchar(50)     NOT NULL,
cust_addr1          varchar(50)     NOT NULL,
cust_addr2          varchar(50)     NOT NULL,
cust_city           varchar(50)     NOT NULL,
cust_state          char(2)         NOT NULL,
cust_postal_code    varchar(10)     NOT NULL,
cust_phone          phone_number    NOT NULL,
cust_fax            varchar(20)     NOT NULL,
cust_email          varchar(30)     NOT NULL,
cust_web_url        varchar(100)    NOT NULL
);
```

When the table is created, internally the *cust_phone* data type is known to be *varchar(20)*. Notice that both *cust_phone* and *cust_fax* are *varchar(20)*, although *cust_phone* has that declaration through its definition as a UDT.

Information about the columns in your tables is available through the catalog view *sys. columns*, which we'll look at in more detail in the section entitled "Internal Storage," later in this chapter. For now, we'll just look at a basic query to show us two columns in *sys.columns*, one containing a number representing the underlying system data type and one containing a number representing the data type used when creating the table. The following query selects all the rows from *sys.columns* and displays the *column_id*, the column name, the data type values, and the maximum length, and then displays the results:

```
SELECT column_id, name, system_type_id, user_type_id,
        type_name(user_type_id) as user_type_name, max_length
FROM sys.columns
WHERE object_id=object_id('customer');

column_id   type_name         system_type_id   user_type_id   user_type_name   max_length
----------- ----------------- ---------------- -------------- ---------------- ----------
1           cust_id           52               52             smallint         2
2           cust_name         167              167            varchar          50
3           cust_addr1        167              167            varchar          50
4           cust_addr2        167              167            varchar          50
5           cust_city         167              167            varchar        50
6           cust_state        175              175            char           2
7           cust_postal_code  167              167            varchar        10
8           cust_phone        167              257            phone_number   20
9           cust_fax          167              167            varchar        20
```

```
10          cust_email        167              167             varchar        30
11          cust_web_url      167              167             varchar        100
```

You can see that both the *cust_phone* and *cust_fax* columns have the same *system_type_id* value, although the *cust_phone* column shows that the *user_type_id* is a UDT (*user_type_id* = 257). The type is resolved when the table is created, and the UDT can't be dropped or changed so long as a table is currently using it. Once declared, a UDT is static and immutable, so no inherent performance penalty occurs in using a UDT instead of the native data type.

The use of UDTs can make your database more consistent and clear. SQL Server implicitly converts between compatible columns of different types (either native types or UDTs of different types).

Currently, UDTs don't support the notion of subtyping or inheritance, nor do they allow a DEFAULT value or a CHECK constraint to be declared as part of the UDT itself. These powerful object-oriented concepts will likely make their way into future versions of SQL Server. These limitations notwithstanding, UDT functionality is a dynamic and often underused feature of SQL Server.

IDENTITY Property

It is common to provide simple counter-type values for tables that don't have a natural or efficient primary key. Columns such as *cust_id* are usually simple counter fields. The *IDENTITY* property makes generating unique numeric values easy. *IDENTITY* isn't a data type; it's a column property that you can declare on a whole number data type such as *tinyint*, *smallint*, *int*, *bigint*, or numeric/decimal (with which only a scale of zero makes any sense). Each table can have only one column with the *IDENTITY* property. The table's creator can specify the starting number (seed) and the amount that this value increments or decrements. If not otherwise specified, the seed value starts at 1 and increments by 1, as shown in this example:

```
CREATE TABLE customer
(
cust_id      smallint        IDENTITY  NOT NULL,
cust_name    varchar(50)     NOT NULL
);
```

To find out which seed and increment values were defined for a table, you can use the *IDENT_SEED(tablename*) and *IDENT_INCR*(*tablename*) functions. Take a look at this statement:

```
SELECT IDENT_SEED('customer'), IDENT_INCR('customer')
```

It produces the following result for the *customer* table because values weren't declared explicitly and the default values were used.

```
1   1
```

This next example explicitly starts the numbering at 100 (seed) and increments the value by 20:

```
CREATE TABLE customer
(
cust_id      smallint        IDENTITY(100, 20)  NOT NULL,
cust_name    varchar(50)     NOT NULL
);
```

The value automatically produced with the *IDENTITY* property is normally unique, but that isn't guaranteed by the *IDENTITY* property itself, nor are the *IDENTITY* values guaranteed to be consecutive. (I will expand on the issues of nonunique and nonconsecutive *IDENTITY* values later in this section.) For efficiency, a value is considered used as soon as it is presented to a client doing an *INSERT* operation. If that client doesn't ultimately commit the *INSERT,* the value never appears, so a break occurs in the consecutive numbers. An unacceptable level of serialization would exist if the next number couldn't be parceled out until the previous one was actually committed or rolled back. (And even then, as soon as a row was deleted, the values would no longer be consecutive. Gaps are inevitable.)

> **Note** If you need exact sequential values without gaps, *IDENTITY* isn't the appropriate feature to use. Instead, you should implement a *next_number*-type table in which you can make the operation of bumping the number contained within it part of the larger transaction (and incur the serialization of queuing for this value).

To temporarily disable the automatic generation of values in an identity column, you use the SET IDENTITY_INSERT tablename ON option. In addition to filling in gaps in the identity sequence, this option is useful for tasks such as bulk-loading data in which the previous values already exist. For example, perhaps you're loading a new database with customer data from your previous system. You might want to preserve the previous customer numbers but have new ones automatically assigned using *IDENTITY*. The SET option was created exactly for cases like this.

Because the SET option allows you to determine your own values for an *IDENTITY* column, the *IDENTITY* property alone doesn't enforce uniqueness of a value within the table. Although *IDENTITY* generates a unique number if IDENTITY_INSERT has never been enabled, the uniqueness is not guaranteed once you have used the SET option. To enforce uniqueness (which you'll almost always want to do when using *IDENTITY*), you should also declare a UNIQUE or PRIMARY KEY constraint on the column. If you insert your own values for an identity column (using SET IDENTITY_INSERT), when automatic generation resumes, the next value is the next incremented value (or decremented value) of the highest value that exists in the table, whether it was generated previously or explicitly inserted.

Tip If you use the *bcp* utility for bulk-loading data, be aware of the *-E* (uppercase) parameter if your data already has assigned values that you want to keep for a column that has the *IDENTITY* property. You can also use the *T-SQL BULK INSERT* command with the KEEPIDENTITY option. For more information, see the SQL Server documentation for *bcp* and *BULK INSERT.*

The keyword *IDENTITYCOL* automatically refers to the specific column in a table that has the *IDENTITY* property, whatever its name. If that column is *cust_id*, you can refer to the column as *IDENTITYCOL* without knowing or using the column name, or you can refer to it explicitly as *cust_id*. For example, the following two statements work identically and return the same data:

```
SELECT IDENTITYCOL FROM customer;
SELECT cust_id FROM customer;
```

The column name returned to the caller is *cust_id*, not *IDENTITYCOL,* in both cases.

When inserting rows, you must omit an identity column from the column list and VALUES section. (The only exception is when the IDENTITY_INSERT option is on.) If you do supply a column list, you must omit the column for which the value will be supplied automatically. Here are two valid *INSERT* statements for the *customer* table shown previously:

```
INSERT customer VALUES ('ACME Widgets');
INSERT customer (cust_name) VALUES ('AAA Gadgets');
```

Selecting these two rows produces this output:

```
cust_id     cust_name
-------     ---------
1           ACME Widgets
2           AAA Gadgets
```

In applications, it's sometimes desirable to know immediately the value produced by *IDENTITY* for subsequent use. For example, a transaction might first add a new customer and then add an order for that customer. To add the order, you probably need to use the *cust_id*. Rather than selecting the value from the *customer* table, you can simply select the special system function *@@IDENTITY*, which contains the last identity value used by that connection. It doesn't necessarily provide the last value inserted in the table, however, because another user might have subsequently inserted data. If multiple *INSERT* statements are carried out in a batch on the same or different tables, the variable has the value for the last statement only. In addition, if an *INSERT* trigger fires after you insert the new row, and if that trigger inserts rows into a table with an identity column, *@@IDENTITY* does not have the value inserted by the original *INSERT* statement. To you, it might look like you're inserting and then immediately checking the value, as follows:

```
INSERT customer (cust_name) VALUES ('AAA Gadgets');
SELECT @@IDENTITY;
```

However, if a trigger were fired for the *INSERT*, the value of *@@IDENTITY* might have changed.

You might find two other functions useful when working with identity columns: *SCOPE_IDENTITY* and *IDENT_CURRENT. SCOPE_IDENTITY* returns the last identity value inserted into a table in the same scope, which could be a stored procedure, trigger, or batch. So if we replace *@@IDENTITY* with the *SCOPE_IDENTITY* function in the preceding code snippet, we can see the identity value inserted into the *customer* table. If an *INSERT* trigger also inserted a row that contained an identity column, it would be in a different scope, like this:

```
INSERT customer (cust_name) VALUES ('AAA Gadgets');
SELECT SCOPE_IDENTITY();
```

In other cases, you might want to know the last identity value inserted in a specific table from any application or user. You can get this value using the *IDENT_CURRENT* function, which takes a table name as an argument:

```
SELECT IDENT_CURRENT('customer');
```

This doesn't always guarantee that you can predict the next identity value to be inserted because another process could insert a row between the time you check the value of *IDENT_CURRENT* and the time you execute your *INSERT* statement.

You can't define the *IDENTITY* property as part of a UDT, but you can declare the *IDENTITY* property on a column that uses a UDT. A column that has the *IDENTITY* property must always be declared NOT NULL (either explicitly or implicitly); otherwise, error number 8147 results from the *CREATE TABLE* statement and *CREATE* won't succeed. Likewise, you can't declare the *IDENTITY* property and a *DEFAULT* on the same column. To check that the current identity value is valid based on the current maximum values in the table, and to reset it if an invalid value is found (which should never be the case), use the *DBCC CHECKIDENT*(tablename) statement.

Identity values are fully recoverable. If a system outage occurs while an insert activity is taking place with tables that have identity columns, the correct value is recovered when SQL Server restarts. SQL Server does this during the checkpoint processing by flushing the current identity value for all tables. For activity beyond the last checkpoint, subsequent values are reconstructed from the transaction log during the standard database recovery process. Any inserts into a table that have the *IDENTITY* property are known to have changed the value, and the current value is retrieved from the last *INSERT* statement (post-checkpoint) for each table in the transaction log. The net result is that when the database is recovered, the correct current identity value is also recovered.

In rare cases, the identity value can get out of sync. If this happens, you can use the *DBCC CHECKIDENT* command to reset the identity value to the appropriate number. In addition, the RESEED option to this command allows you to set a new starting value for the identity sequence. See the online documentation for complete details.

Internal Storage

This section describes how SQL Server actually stores table data. In addition, it explores the basic system metadata that keeps track of data storage information. Although you can use SQL Server effectively without understanding the internals of data storage, a detailed knowledge of how SQL Server stores data helps you develop efficient applications.

When you create a table, one or more rows are inserted into a number of system tables to manage that table and SQL Server provides catalog views built on top of the system tables that allow you to explore their contents. At minimum, you can see metadata for your new table in the *sys.tables*, *sys.indexes*, and *sys.columns* catalog views. When you define the new table with one or more constraints, you also can see information in the *sys.check_constraints*, *sys.default_constraints, sys.key_constraints*, or *sys.foreign_keys* view. For every table created, a single row that contains the name, object ID, and ID of the schema containing the new table (among other items) is available through the *sys.tables* view. Remember that the *sys.tables* view inherits all the columns from *sys.objects* (which shows information relevant to all types of objects) and then includes additional columns pertaining only to tables. The *sys.columns* view shows you one row for each column in the new table, and each row contains information such as the column name, data type, and length. Each column receives a column ID, which initially corresponds to the order in which you specified the columns when you created the table— that is, the first column listed in the *CREATE TABLE* statement has a column ID of 1, the second column has a column ID of 2, and so on. Figure 5-6 shows the rows returned by the *sys.tables* and *sys.columns* views when you create a table. (Not all columns are shown for each view.)

```
CREATE TABLE dbo.employee (
                  emp_lname   varchar(15)   NOT NULL,
                  emp_fname   varchar(10)   NOT NULL,
                  address     varchar(30)   NOT NULL,
                  phone       char(12)      NOT NULL,
                  job_level   smallint      NOT NULL
)

sys.tables        object_id   name                 schema_id   type_desc
                  ----------- -------------------- ----------- ------------------
                  917578307   employee             1           USER_TABLE

sys.columns       object_id   column_id   name         system_type_id max_length
                  ----------- ----------- ------------ -------------- ----------
                  917578307   1           emp_lname      167            15
```

```
                    917578307    2            emp_fname       167              10
                    917578307    3            address         167              30
                    917578307    4            phone           175              12
                    917578307    5            job_level       52               2
```

Figure 5-6: Basic catalog information stored after a table is created

Note There can be gaps in the column ID sequence if the table is altered to drop columns. However, the information schema view (*INFORMATION_SCHEMA.COLUMNS*) gives you a value called *ORDINAL_POSITION* because that is what the ANSI SQL standard demands. The ordinal position is the order the column will be listed when you *SELECT ** on the table. So the *column_id* is not necessarily the ordinal position of that column.

The sys.indexes Catalog View

In addition to *sys.columns* and *sys.tables*, the *sys.indexes* view returns at least one row for each table. In versions of SQL Server prior to SQL Server 2005, the *sysindexes* table contains all the physical storage information for both tables and indexes, which are the only objects that actually use storage space. The *sysindexes* table has columns to keep track of the space used by all tables and indexes, the physical location of each index root page, and the first page of each table and index. (In Chapter 6, you'll see more about root pages and what the "first" page actually means.) In SQL Server 2008, the compatibility view sys.*sysindexes* contains much of the same information, but it is incomplete because of changes in the storage organization introduced in SQL Server 2005. The *sys.indexes* catalog view contains only basic property information about indexes, such as whether the index is clustered or nonclustered, unique or nonunique, and other properties, which are discussed in Chapter 6. To get all the storage information in SQL Server 2005 or SQL Server 2008 that previous versions provided in the *sysindexes* table, we have to look at two other catalog views in addition to *sys.indexes*: *sys.partitions* and *sys.allocation_units* (or alternatively, the undocumented *sys.system_internals_allocation_units*). I'll discuss the basic contents of these views shortly, but first let's focus on *sys.indexes*.

You might be aware that if a table has a clustered index, the table's data is actually considered part of the index, so the data rows are actually index rows. For a table with a clustered index, SQL Server has a row in *sys.indexes* with an *index_id* value of 1 and the *name* column in *sys.indexes* contains the name of the index. The name of the table that is associated with the index can be determined from the *object_id* column in *sys.indexes*. If a table has no clustered index, there is no organization to the data itself, and we call such a table a *heap*. A heap in *sys.indexes* table has an *index_id* value of 0, and the *name* column contains NULL. Every additional index has a row in *sys.indexes* with an *index_id* value between 2 and 250 or between 256 and 1,005. (The values 251 – 255 are reserved.) Because as many as 999 nonclustered indexes can be on a single table and there is one row for the heap or clustered index, every table has between 1 and 1,000 rows in the *sys.indexes* view for relational indexes. A table can have additional rows in *sys.indexes* for XML indexes. Metadata for XML indexes is available in the *sys.xml_indexes* catalog view, which inherits columns from the *sys.indexes* view. Two main features in SQL Server 2008 make it most efficient to use more than one catalog view to keep track of storage information. First, SQL Server has the ability to store a table or index on multiple partitions, so the space used by each partition, as well as the partition's location, must be kept track of separately. Second, table and index data can be stored in three different formats, which are regular row data, row-overflow data, and LOB data. Both row-overflow data and LOB data can be part of an index, so each index has to keep track of its special format data separately. So each table can have multiple indexes, and each table and index can be stored on multiple partitions, and each partition needs to keep track of data in up to three formats. I'll discuss indexes in Chapter 6, and I'll discuss the storage of row-overflow data and LOB data, as well as partitioned tables and indexes, in Chapter 7.

Data Storage Metadata

Each heap and index has a row in *sys.indexes*, and each table and index in a SQL Server 2008 database can be stored on multiple partitions. The *sys.partitions* view contains one row for each partition of each heap or index. Every heap or index has at least one partition, even if you haven't specifically partitioned the structure, but one table or index can have up to 1,000 partitions. So there is a one-to-many relationship between *sys.indexes* and *sys.partitions*. The *sys.partitions* view contains a column called *partition_id* as well as the *object_id* and *index_id*, so we can join *sys.indexes* to *sys.partitions* on the *object_id* and *index_id* columns to retrieve all the partition ID values for a particular table or index. The term used in SQL Server 2008 to describe a subset of a table or index on a single partition is *hobt*, which stands for Heap Or B-Tree and is pronounced (you guessed it) "hobbit." (A B-tree is the storage structure used for indexes.) The *sys.partitions* view includes a column called *hobt_id*, and in SQL Server 2008, there is always a one-to-one relationship between *partition_id* and *hobt_id*. In fact, you can see that these two columns in the *sys.partitions* table always have the same value.

Each partition (whether for a heap or an index) can have three types of rows, each stored on its own set of pages. These types are called *in-row data pages* (for our "regular" data or index information), *row-overflow data pages*, and *LOB data pages*. A set of pages of one particular type for one particular partition is called an *allocation unit*, so the final catalog view I need to tell you about is *sys.allocation_units*. The *sys.allocation_units* view contains one, two, or three rows per partition because each heap or index on each partition can have as many as three allocation units. There is always an allocation unit for regular in-row pages, but there might also be an allocation unit for LOB data and one for row-overflow data. Figure 5-7 shows the relationship between *sys.indexes*, *sys.partitions*, and *sys.allocation_units*.

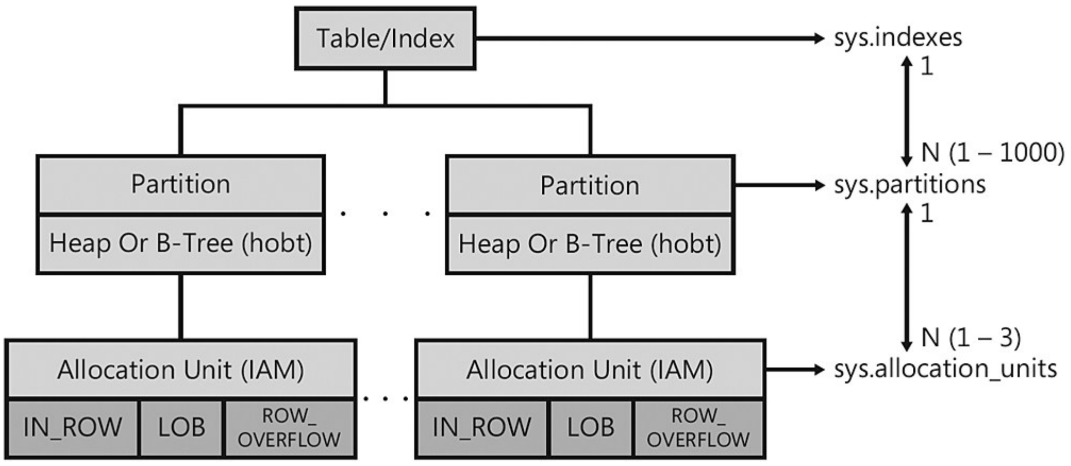


Figure 5-7: The relationship between sys.indexes, sys.partitions, and sys.allocation_units

Querying the Catalog Views

Let's look at a specific example now to see information in these three catalog views. Let's first create the table shown earlier in Figure 5-6. You can create it in any database, but I suggest either using *tempdb*, so the table is dropped automatically the next time you restart your SQL Server instance, or creating a new database just for testing. Many of my examples assume a database called *test:*

```
CREATE TABLE dbo.employee(
               emp_lname   varchar(15)   NOT NULL,
               emp_fname   varchar(10)   NOT NULL,
               address     varchar(30)   NOT NULL,
               phone       char(12)      NOT NULL,
               job_level   smallint      NOT NULL
);
```

This table has one row in *sys.indexes* and one in *sys.partitions*, as we can see when we run the following queries. I am including only a few of the columns from *sys.indexes,* but *sys.partitions* only has six columns, so I have retrieved them all:

```
SELECT  object_id, name, index_id, type_desc
FROM sys.indexes
WHERE object_id=object_id('dbo.employee');

SELECT *
FROM sys.partitions
WHERE object_id=object_id('dbo.employee');
```

Here are my results (yours might vary slightly because your ID values are probably different):

```
object_id   name    index_id    type_desc
----------- ------- ----------  ------------
5575058     NULL    0           HEAP

partition_id       object_id  index_id  partition_number  hobt_id            rows
------------------ ---------- --------- ----------------- -----------------  -----
72057594038779904  5575058    0         1                 72057594038779904  0
```

Each row in the *sys.allocation_units* view has a unique *allocation_unit_id* value. Each row also has a value in the column called *container_id* that can be joined with *partition_id* in *sys.partitions*, as shown in this query:

```
SELECT object_name(object_id) AS name,
    partition_id, partition_number AS pnum,  rows,
    allocation_unit_id AS au_id, type_desc as page_type_desc,
    total_pages AS pages
FROM sys.partitions p JOIN sys.allocation_units a
   ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.employee');
```

Again, for this simple table, I get only one row because there is only one partition, no nonclustered indexes, and only one type of data (IN_ROW_DATA). Here is the result:

```
name     partition_id        pnum   rows   au_id              page_type_desc pages
-------- ------------------  ------ ------ ------------------ -------------- ------
employee 72057594038779904   1      0      72057594043301888  IN_ROW_DATA    0
```

Now let's add some new columns to the table that need to be stored on other types of pages. *Varchar* data can be stored on row-overflow pages if the total row size exceeds the maximum of 8,060 bytes. By default, text data is stored on text pages. For *varchar* data that is stored on row-overflow pages, and for text data, there is additional overhead in the row itself to store a pointer to the off-row data. We'll look at the details of row-overflow and text data storage later in this section, and we'll look at *ALTER TABLE* at the end of this chapter, but now I just want to look at the additional rows in *sys.allocation_units:*

```
ALTER TABLE dbo.employee ADD resume_short varchar(8000);
ALTER TABLE dbo.employee ADD resume_long text;
```

If we run the preceding query that joins *sys.partitions* and *sys.allocation_units*, we get the following three rows:

```
name     partition_id           pnum   rows   au_id               page_type_desc      pages
-------- ---------------------- ------ ------ ------------------- ------------------- -----
employee 72057594038779904      1      0      72057594043301888   IN_ROW_DATA         0
employee 72057594038779904      1      0      72057594043367424   ROW_OVERFLOW_DATA   0
employee 72057594038779904      1      0      72057594043432960   LOB_DATA            0
```

You might also want to add an index or two and check the contents of these catalog views again. You should notice that just adding a clustered index does not change the number of rows in *sys.allocation_units*, but it does change the *partition_id* numbers because the entire table is rebuilt internally when you create a clustered index. Adding a nonclustered index adds at least one more row to *sys.allocation_units* to keep track of the pages for that index. The following query joins all three views—*sys.indexes*, *sys.partitions*, and *sys.allocation_units*—to show you the table name, index name and type, page type, and space usage information for the *dbo.employee* table:

```
SELECT  convert(char(8),object_name(i.object_id)) AS table_name,
    i.name AS index_name, i.index_id, i.type_desc as index_type,
    partition_id, partition_number AS pnum,  rows,
    allocation_unit_id AS au_id, a.type_desc as page_type_desc,
    total_pages AS pages
FROM sys.indexes i JOIN sys.partitions p
        ON i.object_id = p.object_id AND i.index_id = p.index_id
    JOIN sys.allocation_units a
        ON p.partition_id = a.container_id
WHERE i.object_id=object_id('dbo.employee');
```

Because I have not inserted any data into this table, you should notice that the values for rows and pages are all 0. When I discuss actual page structures, we'll insert data into our tables so we can look at the internal storage of the data at that time. The queries I've run so far do not provide us with any information about the location of pages in the various allocation units. In SQL Server 2000, the *sysindexes* table contains three columns that indicate where data is located; these columns are called *first*, *root*, and *firstIAM*. These columns are still available in SQL Server 2008 (with slightly different names: *first_page, root_page*, and *first_iam_page*), but they can be seen only in an undocumented view called *sys.system_internals_allocation_units*. This view is identical to *sys.allocation_units* except for the addition of these three additional columns, so you can replace *sys.allocation_units* with *sys.system_internals_allocation_units* in the preceding allocation query and add these three extra columns to the select list. Keep in mind that as an undocumented object, this view is for internal use only and is subject to change (as are other views starting with *system_internals*). Forward compatibility is not guaranteed.

Data Pages

Data pages are the structures that contain user data that has been added to a database's tables. As we saw earlier, there are three varieties of data pages, each of which stores data in a different format. There are pages for in-row data, pages for row-overflow data, and pages for LOB data. As with all other types of pages in SQL Server, data pages have a fixed size of 8 KB, or 8,192 bytes. They consist of three major components: the page header, data rows, and the row offset array, as shown in Figure 5-8.

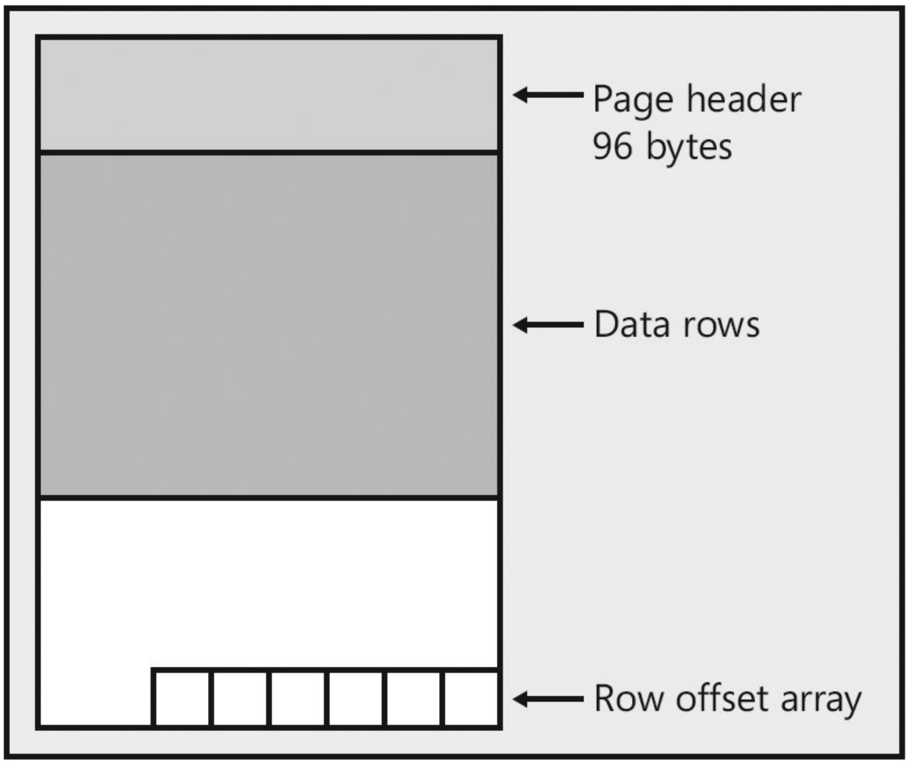


Figure 5-8: The structure of a data page

Page Header

As you can see in Figure 5-8, the page header occupies the first 96 bytes of each data page (leaving 8,096 bytes for data, row overhead, and row offsets). Table 5-5 shows some of the information shown when we examine the page header.

Table 5-5: Information Available by Examining the Page Header

Field	Meaning
pageID	The file number and page number of this page in the database
nextPage	The file number and page number of the next page if this page is in a page chain
prevPage	The file number and page number of the previous page if this page is in a page chain
Metadata: ObjectId	The ID of the object to which this page belongs
Metadata: PartitionId	The ID of the partition that this page is part of
Metadata: AllocUnitId	The ID of the allocation unit that contains this page
LSN	The Log Sequence Number (LSN) corresponding to the last log entry that changed this page
slotCnt	The total number of slots (rows) used on this page
Level	The level of this page in an index (which always is 0 for leaf pages)
indexId	The index ID of this page (always 0 for data pages)
freeData	The byte offset of the first free space on this page
Pminlen	The number of bytes in fixed-length portion of rows
freeCnt	The number of free bytes on the page

reservedCnt	The number of bytes reserved by all transactions
xactreserved	The number of bytes reserved by the most recently started transaction
tornBits	A bit string containing 1 bit per sector for detecting torn page writes (or checksum information if *torn_page_detection* is not on)
flagBits	A 2-byte bitmap that contains additional information about the page

Data Rows for In-Row Data

Following the page header is the area in which the table's actual data rows are stored. The maximum size of a single data row is 8,060 bytes of in-row data. Rows can also have row-overflow and LOB data stored on separate pages. The number of rows stored on a given page varies depending on the structure of the table and on the data being stored. A table that has all fixed-length columns is always able to store the same number of rows per page; variable-length rows can store as many rows that fit based on the actual length of the data entered. Keeping the row length shorter allows more rows to fit on a page, thus reducing I/O and improving the cache-hit ratio.

Row Offset Array

The row offset array is a block of 2-byte entries, each indicating the offset on the page at which the corresponding data row begins. Every row has a 2-byte entry in this array (as discussed earlier, when I mentioned the 10 overhead bytes needed by every row). Although these bytes aren't stored in the row with the data, they do affect the number of rows that fit on a page.

The row offset array indicates the logical order of rows on a page. For example, if a table has a clustered index, SQL Server stores the rows in the order of the clustered index key. This doesn't mean the rows are physically stored on the page in the order of the clustered index key. Rather, slot 0 in the offset array refers to the first row in the clustered index key order, slot 1 refers to the second row, and so forth. As we'll see shortly when we examine an actual page, the physical location of these rows can be anywhere on the page.

Examining Data Pages

You can view the contents of a data page by using the *DBCC PAGE* command, which allows you to view the page header, data rows, and row offset table for any given page in a database. Only a system administrator can use *DBCC PAGE.* But because you typically won't need to view the contents of a data page, you won't find information about *DBCC PAGE* in the SQL Server documentation. Nevertheless, in case you want to use it, here's the syntax:

```
DBCC PAGE ({dbid | dbname}, filenum, pagenum[, printopt])
```

The *DBCC PAGE* command includes the parameters shown in Table 5-6. The code and results following Table 5-6 show sample output from *DBCC PAGE* with a *printopt* value of 1. Note that *DBCC TRACEON(3604)* instructs SQL Server to return the results to the client. Without this traceflag, no output is returned for the *DBCC PAGE* command.

Table 5-6: Parameters of the DBCC Page Command

Parameter	Description
Dbid	The ID of the database containing the page
Dbname	The name of the database containing the page
Filenum	The file number containing the page
Pagenum	The page number within the file
Printopt	An optional print option; takes one of these values: ▪ 0 Default; prints the buffer header and page header ▪ 1 Prints the buffer header, page header, each row separately, and the row offset table ▪ 2 Prints the buffer and page headers, the page as a whole, and the offset table ▪ 3 Prints the buffer header, page header, each row separately, and the row offset table; each row is followed by each of its column values listed separately

```
DBCC TRACEON(3604);
GO
```

```
DBCC PAGE (pubs, 1, 157, 1);
GO
PAGE: (1:157)
BUFFER:
BUF @0x038E697C
bpage = 0x0C3AA000                   bhash = 0x00000000                   bpageno = (1:157)
bdbid = 11                           breferences = 0                      bUse1 = 60722
bstat = 0xc00009                     blog = 0x3212159                     bnext = 0x00000000

PAGE HEADER:
Page @0x0C3AA000
m_pageId = (1:157)                   m_headerVersion = 1                  m_type = 1
m_typeFlagBits = 0x4                 m_level = 0                          m_flagBits = 0x200
m_objId (AllocUnitId.idObj) = 27     m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594039697408
Metadata: PartitionId = 72057594038779904                                 Metadata: IndexId = 1
Metadata: ObjectId = 2105058535      m_prevPage = (0:0)                   m_nextPage = (0:0)
pminlen = 24                         m_slotCnt = 23                       m_freeCnt = 6010
m_freeData = 2136                    m_reservedCnt = 0                    m_lsn = (18:350:2)
m_xactReserved = 0                   m_xdesId = (0:0)                     m_ghostRecCnt = 0
m_tornBits = 1967525613

Allocation Status
GAM (1:2) = ALLOCATED                SGAM (1:3) = NOT ALLOCATED
PFS (1:1) = 0x60 MIXED_EXT ALLOCATED   0_PCT_FULL                         DIFF (1:6) = CHANGED
ML (1:7) = NOT MIN_LOGGED

DATA:
Slot 0, Offset 0x631, Length 88, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 88
Memory Dump @0x6292C631
00000000:   30001800 34303820 3439362d 37323233 †0...408 496-7223
00000010:   43413934 303235ff 09000000 05003300 †CA94025ÿ .....3.
00000020:   38003f00 4e005800 3137322d 33322d31 †8.?.N.X.172-32-1
00000030:   31373657 68697465 4a6f686e 736f6e31 †176WhiteJohnson1
00000040:   30393332 20426967 67652052 642e4d65 †0932 Bigge Rd.Me
00000050:   6e6c6f20 5061726b ††††††††††††††††††nlo Park

Slot 1, Offset 0xb8, Length 88, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 88
Memory Dump @0x6292C0B8
00000000:   30001800 34313520 3938362d 37303230 †0...415 986-7020
00000010:   43413934 363138ff 09000000 05003300 †CA94618ÿ .....3.
00000020:   38004000 51005800 3231332d 34362d38 †8.@.Q.X.213-46-8
00000030:   39313547 7265656e 4d61726a 6f726965 †915GreenMarjorie
00000040:   33303920 36337264 2053742e 20233431 †309 63rd St. #41
00000050:   314f616b 6c616e64 ††††††††††††††††††1Oakland

Slot 2, Offset 0x110, Length 85, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 85
Memory Dump @0x6292C110
00000000:   30001800 34313520 3534382d 37373233 †0...415 548-7723
00000010:   43413934 373035ff 09000000 05003300 †CA94705ÿ .....3.
00000020:   39003f00 4d005500 3233382d 39352d37 †9.?.M.U.238-95-7
00000030:   37363643 6172736f 6e436865 72796c35 †766CarsonCheryl5
00000040:   38392044 61727769 6e204c6e 2e426572 †89 Darwin Ln.Ber
00000050:   6b656c65 79†††††††††††††††††††††††††keley
```

/* Data for slots 3 through 20 not shown */

```
Slot 21, Offset 0x1c0, Length 89, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 89
Memory Dump @0x6292C1C0
00000000:   30001800 38303120 3832362d 30373532 †0...801 826-0752
00000010:   55543834 313532ff 09000000 05003300 †UT84152ÿ .....3.
```

```
00000020:   39003d00 4b005900 3839392d 34362d32 †9.=.K.Y.899-46-2
00000030:   30333552 696e6765 72416e6e 65363720 †035RingerAnne67
00000040:   53657665 6e746820 41762e53 616c7420 †Seventh Av.Salt
00000050:   4c616b65 20436974 79†††††††††††††††††Lake City

Slot 22, Offset 0x165, Length 91, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 91
Memory Dump @0x6292C165
00000000:   30001800 38303120 3832362d 30373532 †0...801 826-0752
00000010:   55543834 313532ff 09000000 05003300 †UT84152ÿ .....3.
00000020:   39003f00 4d005b00 3939382d 37322d33 †9.?.M.[.998-72-3
00000030:   35363752 696e6765 72416c62 65727436 †567RingerAlbert6
00000040:   37205365 76656e74 68204176 2e53616c †7 Seventh Av.Sal
00000050:   74204c61 6b652043 697479†††††††††††††t Lake City

OFFSET TABLE:
Row - Offset
22 (0x16) - 357 (0x165)
21 (0x15) - 448 (0x1c0)
20 (0x14) - 711 (0x2c7)
19 (0x13) - 1767 (0x6e7)
18 (0x12) - 619 (0x26b)
17 (0x11) - 970 (0x3ca)
16 (0x10) - 1055 (0x41f)
15 (0xf) - 796 (0x31c)
14 (0xe) - 537 (0x219)
13 (0xd) - 1673 (0x689)
12 (0xc) - 1226 (0x4ca)
11 (0xb) - 1949 (0x79d)
10 (0xa) - 1488 (0x5d0)
9 (0x9) - 1854 (0x73e)
8 (0x8) - 1407 (0x57f)
7 (0x7) - 1144 (0x478)
6 (0x6) - 96 (0x60)
5 (0x5) - 2047 (0x7ff)
4 (0x4) - 884 (0x374)
3 (0x3) - 1314 (0x522)
2 (0x2) - 272 (0x110)
1 (0x1) - 184 (0xb8)
0 (0x0) - 1585 (0x631)


DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

As you can see, the output from *DBCC PAGE* is divided into four main sections: BUFFER, PAGE HEADER, DATA, and OFFSET TABLE (really the offset array). The BUFFER section shows information about the buffer for the given page. A *buffer* in this context is the in-memory structure that manages a page, and the information in this section is relevant only when the page is in memory.

The PAGE HEADER section in the output from *DBCC PAGE* displays the data for all the header fields on the page. (Table 5-5 shows the meaning of most of these fields.) The DATA section contains information for each row. When *DBCC PAGE* is used with a *printopt* value of 1 or 3, *DBCC PAGE* indicates the slot position of each row, the offset of the row on the page, and the length of the row. The row data is divided into three parts. The left column indicates the byte position within the row where the displayed data occurs. The next section contains the actual data stored on the page, displayed in four columns of eight hexadecimal digits each. The right column contains an ASCII character representation of the data. Only character data is readable in this column, although some of the other data might be displayed.

The OFFSET TABLE section shows the contents of the row offset array at the end of the page. In the output from *DBCC PAGE*, you can see that this page contains 23 rows, with the first row (indicated by slot 0) beginning at offset 1585 (0x631). The first row physically stored on the page is actually row 6, with an offset in the row offset array of 96. *DBCC PAGE* with a *printopt* value of 1 displays the rows in slot number order, even though, as you can see by the offset of each of the slots, that it isn't the order in which the rows physically exist on the page. If you use *DBCC PAGE* with a *printopt* value of 2, you see a dump of all 8,096 bytes of the page (after the header) in the order they are stored on the page.

The Structure of Data Rows

A table's data rows have the general structure shown in Figure 5-9 (so long as the data is stored in uncompressed form). We call this format the FixedVar format, because the data for all fixed-length columns is stored first, followed by the data for all variable-length columns. Table 5-7 shows the information stored in each FixedVar row. (In Chapter 7, we'll see the format of rows stored in a different format, used when the data on the page is compressed.)

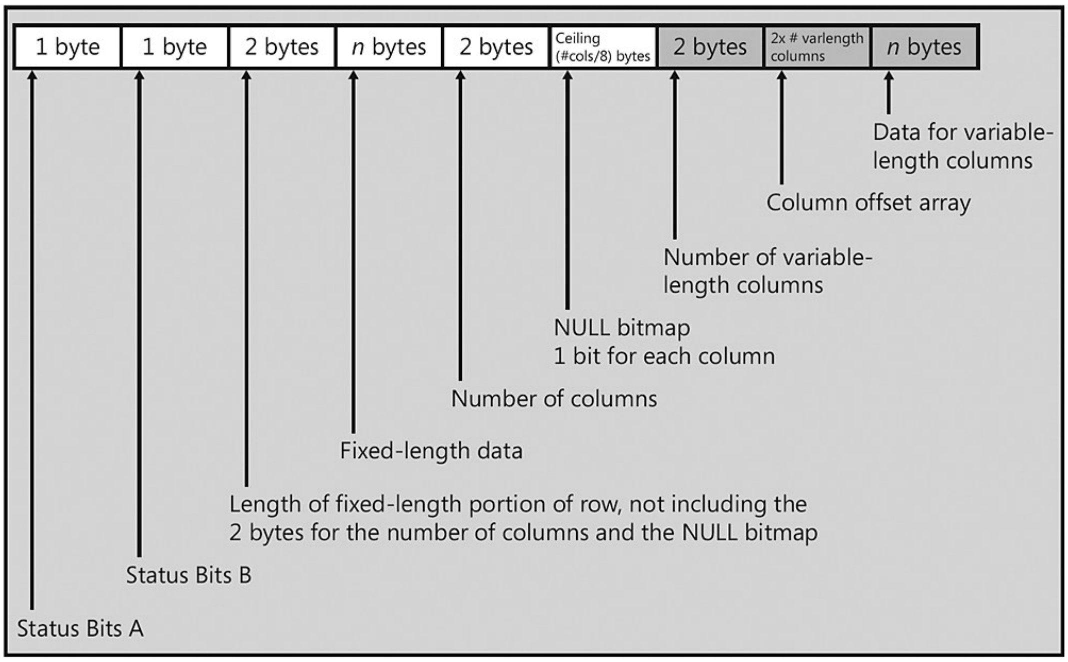

1 byte
1 byte
2 bytes
n bytes
2 bytes
Ceiling (#cols/8) bytes
2 bytes
2x # varlength columns
n bytes
Data for variable-length columns
Column offset array
Number of variable-length columns
NULL bitmap 1 bit for each column
Number of columns
Fixed-length data
Length of fixed-length portion of row, not including the 2 bytes for the number of columns and the NULL bitmap
Status Bits B
Status Bits A

Figure 5-9: The structure of data rows

Table 5-7: Information Stored in a Table's Data Rows

Information	**Mnemonic**	**Size**
Status Bits A	TagA	1 byte
Status Bits B	TagB	1 byte
Fixed-length size	Fsize	2 bytes
Fixed-length data	Fdata	Fsize – 4
Number of columns	Ncol	2 bytes
NULL bitmap (1 bit for each column in the table; a 1 indicates that the corresponding column is NULL or that the bit is unused.)	Nullbits	Ceiling (Ncol / 8)
Number of variable-length columns stored in row	VarCount	2 bytes
Variable column offset array	VarOffset	2 * VarCount
Variable-length data	VarData	VarOff[VarCount] - (Fsize + 4 + Ceiling (Ncol / 8) + 2 * VarCount)

Status Bits A contains a bitmap indicating properties of the row. The bits have the following meanings:

- **Bit 0** Versioning information. In SQL Server 2008, this is always 0.
- **Bits 1 through 3** Taken as a three-bit value, 0 indicates a primary record, 1 indicates a forwarded record, 2 indicates a forwarding stub, 3 indicates an index record, 4 indicates a blob fragment or row-overflow data, 5 indicates a ghost index record, 6 indicates a ghost data record, and 7 indicates a ghost version record. (I'll discuss forwarded records in the section entitled "Moving Rows," later in this chapter, and ghost records in Chapter 6.)
- **Bit 4** Indicates that a NULL bitmap exists. In SQL Server 2008, a NULL bitmap is always present, even if no NULLs are allowed in any column.
- **Bit 5** Indicates that variable-length columns exist in the row.

- **Bit 6** Indicates that the row contains versioning information.
- **Bit 7** Not used in SQL Server 2008.

Only one bit is used in the Status Bits B field, indicating that the record is a ghost forwarded record.

You can see in both Figure 5-9 and Table 5-7 that the third and fourth bytes indicate the length of the fixed-length portion of the row. As Figure 5-9 explains, it is the length excluding the 2 bytes for the number of columns, and the NULL bitmap, which is variable length depending on the total number of columns in the table. Another way to interpret the data in these bits is as the location in the row where the number of columns can be found. For example, if the third and fourth bytes (bytes 2-3) contain the value 0x0016, which is decimal 22, it means not only that there are 22 bytes in the row before the value for number of columns, but that the value for the number of columns can be found at byte 22. In some of the figures in this chapter and later ones, bytes 2-3 may be identified as the position to find the number of columns.

Within each block of fixed-length or variable-length data, the data is stored in the column order in which the table was created. For example, suppose a table is created with the following statement:

```
CREATE TABLE Test1
(
Col1 int                NOT NULL,
Col2 char(25)           NOT NULL,
Col3 varchar(60)        NULL,
Col4 money              NOT NULL,
Col5 varchar(20)        NOT NULL
);
```

The fixed-length data portion of this row contains the data for *Col1*, followed by the data for *Col2*, followed by the data for *Col4*. The variable-length data portion contains the data for *Col3*, followed by the data for *Col5*. For rows that contain only fixed-length data, the following is true:

- The first hexadecimal digit of the first byte of the data row is 1, indicating that no variable-length columns exist. (The first hexadecimal digit comprises bits 4 through 7; bits 6 and 7 are always 0, and if there are no variable-length columns, bit 5 is also 0. Bit 4 is always 1, so the value of the four bits is displayed as 1.)
- The data row ends after the NULL bitmap, which follows the fixed-length data (that is, the shaded portion shown in Figure 5-9 won't exist in rows with only fixed-length data).
- The total length of every data row is the same.

A data row that has any variable-length columns has a column offset array in the data row with a 2-byte entry for each non-NULL variable-length column, indicating the position within the row where the column ends. (The terms *offset* and *position* aren't exactly interchangeable. *Offset* is 0-based, and *position* is 1-based. A byte at an offset of 7 is in the eighth byte position in the row.) There are some special issues storing variable-length columns with a NULL value, and I'll discuss this issue in the section entitled "NULLs and Variable-Length Columns," later in this chapter.

Finding a Physical Page

Before we examine specific data, we need to digress a bit. The examples that follow use the *DBCC PAGE* command to examine the physical database pages. To run this command, I need to know what page numbers are used to store rows for a table. I mentioned previously that a value for *first_page* was stored in an undocumented view called *sys.system_internals_ allocation_units*, which is almost identical to the *sys.allocation_units* view. First, let me create a table (that will be used in the following section) and insert a single row into it:

```
USE tempdb;
CREATE TABLE Fixed
(
Col1 char(5)      NOT NULL,
Col2 int          NOT NULL,
Col3 char(3)      NULL,
Col4 char(6)      NOT NULL
);
INSERT Fixed VALUES ('ABCDE', 123, NULL, 'CCCC');
```

The following query gives me the value for *first_page* in the *Fixed* table:

```
SELECT object_name(object_id) AS name,
    rows, type_desc as page_type_desc,
    total_pages AS pages, first_page
FROM sys.partitions p  JOIN sys.system_internals_allocation_units a
   ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.Fixed');
```

```
RESULTS:
name    rows   page_type_desc   pages  first_page
------- ------ ---------------  ------ --------
Fixed   1      IN_ROW_DATA     2      0xCF0400000100
```

I can then take the value of *first_page* from the preceding *sys.system_internals_ allocation_units* output (0xCF0400000100) and convert it to a file and page address. (The value that you get for *first_page* most likely will be different than the one I got.) In hexadecimal notation, each set of two hexadecimal digits represents a byte. I first had to swap the bytes to get 00 01 00 00 04 CF. The first two groups represent the 2-byte file number; the last four groups represent the page number. So the file is 0x0001, which is 1, and the page number is 0x000004CF, which is 1231 in decimal.

Unless you particularly enjoy playing with hexadecimal conversions, you might want to use one of three other options for determining the actual page numbers associated with your SQL Server tables. First you can create the function shown here to convert a 6-byte hexadecimal page number value (such as 0xCF0400000100) to a *file_number:page_number* format:

```
CREATE FUNCTION convert_page_nums (@page_num binary(6))
   RETURNS varchar(11)
AS
  BEGIN
   RETURN(convert(varchar(2), (convert(int, substring(@page_num, 6, 1))
          * power(2, 8)) +
             (convert(int, substring(@page_num, 5, 1)))) + ':' +
               convert(varchar(11),
   (convert(int, substring(@page_num, 4, 1)) * power(2, 24)) +
   (convert(int, substring(@page_num, 3, 1)) * power(2, 16)) +
   (convert(int, substring(@page_num, 2, 1)) * power(2, 8)) +
   (convert(int, substring(@page_num, 1, 1)))) )
  END;
```

You can then execute this *SELECT* to call the function:

```
SELECT dbo.convert_page_nums(0xCF0400000100);
```

You should get back the result *1:1231*.

Warning SQL Server does not guarantee that the *first_page* column in *sys.system_internals_ allocation_units* always indicates the first page of a table. (The view is undocumented, after all.) I've found that *first_page* is reliable until you begin to perform deletes and updates on the data in the table.

The second option for determining the actual page numbers is to use another undocumented command called *DBCC IND.* Because most of the information returned is relevant only to indexes, I won't discuss this command in detail until Chapter 6. However, for a sneak preview, you can run the following command and note the values in the first two columns of output (labeled *PageFID* and *PagePID*) in the row where *PageType* = 1, which indicates that the page is a data page:

```
DBCC IND(tempdb, Fixed, -1);
```

If you weren't in *tempdb*, you would replace *tempdb* with the name of whatever database you were in when you created this table. The values for *PageFID* and *PagePID* should be the same value you used when you converted the hexadecimal string for the *first_page* value. In my case, I see that *PageFID* value is 1 and the *PagePID* value is 1231. So those are the values I use when calling *DBCC PAGE*.

The third method for obtaining file and page number information involves using an undocumented function, *sys.fn_PhysLocFormatter,* in conjunction with an undocumented value, *%%physloc%%,* to return the physical row location in your result rows along with data values from a table. This can be useful if you want to find which page in a table contains a particular value. *DBCC IND* can be used to find all the pages in a table but not specifically the pages containing a particular row. However, *sys.fn_PhysLocFormatter* can show you only data pages for the data that is returned in a *SELECT* statement. We can use this function to get the pages used by our data in the table *Fixed,* as follows:

```
SELECT sys.fn_PhysLocFormatter (%%physloc%%) AS RID, * FROM Fixed;
```

```
GO
```

Here are my results:

```
RID          Col1    Col2          Col3   Col4
------------ ------- ------------- ------ ------
(1:1231:1)   ABCDE   123           NULL   CCCC
```

Once you have the *FileID* and *PageID* values, you can use *DBCC PAGE*. For a larger table, we could use *sys.fn_PhysLocFormatter* to get the the pages only for the specific rows that were returned by the conditions in our WHERE clause.

Caution The *%%physloc%%* value is not understood by the relational engine, which means that if you use *%%physloc%%* in a WHERE clause, SQL Server has to examine every row to see which ones are on the page indicated by *%%physloc%%.* It is not able to use *%%physloc%%* to find the row. Another way of looking at this is that *%%physloc%%* can be returned as output to report on a physical row location, but cannot be used as input to find a particular location in a table. The *%%physloc%%* value was introduced as a debugging feature by the SQL Server product development team and is not intended to be used (and is not supported) in production applications.

The two examples that follow illustrate how fixed-length and variable-length data rows are stored.

Storage of Fixed-Length Rows

First, let's look at the simpler case of an all fixed-length row using the table I just built in the preceding section:

```
CREATE TABLE Fixed
(
Col1 char(5)     NOT NULL,
Col2 int         NOT NULL,
Col3 char(3)     NULL,
Col4 char(6)     NOT NULL
);
```

When this table is created, you should be able to execute the following queries against the *sys.indexes* and *sys.columns* views to receive the information similar to the results shown:

```
SELECT object_id,  type_desc,
    indexproperty(object_id, name, 'minlen') as min_row_len
    FROM sys.indexes where object_id=object_id('Fixed');

SELECT  column_id, name, system_type_id, max_length as max_col_len
FROM sys.columns
WHERE object_id=object_id('Fixed');

RESULTS:
object_id     type_desc    minlen
------------- ------------ -------
53575229      HEAP         22

column_id     name                 system_type_id   max_length
------------- -------------------- ---------------- ----------
1             Col1                 175              5
2             Col2                 56               4
3             Col3                 175              3
4             Col4                 175              6
```

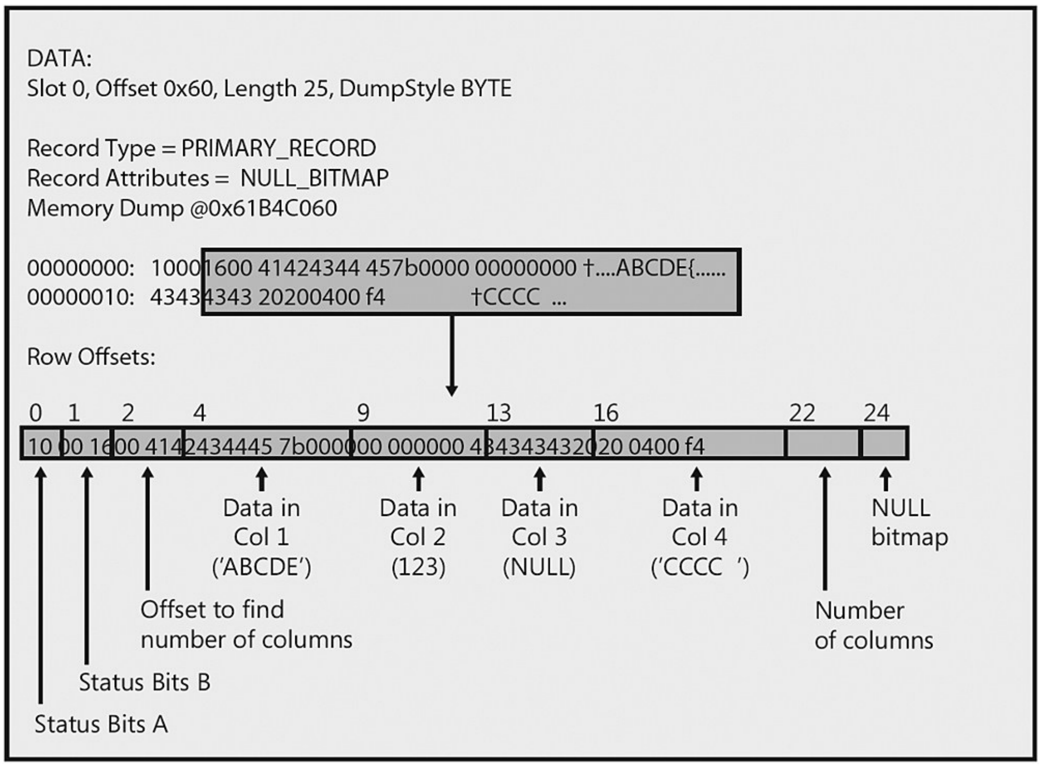
Note The *sysindexes* compatibility view contains columns called *minlen* and *xmaxlen*, which store the minimum and maximum length of a row. In SQL Server 2008, these values are not available in any of the catalog views, but you can get them by using undocumented parameters to the *indexproperty* function. As with all undocumented features, keep in mind that they are not supported by Microsoft and future compatibility is not guaranteed.

For tables containing only fixed-length columns, the value returned for *minlen* by the *indexproperty* function equals the sum of the column lengths (from *sys.columns.max_length*) plus 4 bytes. It doesn't include the 2 bytes for the number of columns, or the bytes for the NULL bitmap.

To look at a specific data row in this table, you must first insert a new row. If you didn't insert this row in the preceeding section, insert it now:

```
INSERT Fixed VALUES ('ABCDE', 123, NULL, 'CCCC');
```

Figure 5-10 shows this row's actual contents on the data page.



```
DATA:
Slot 0, Offset 0x60, Length 25, DumpStyle BYTE

Record Type = PRIMARY_RECORD
Record Attributes =  NULL_BITMAP
Memory Dump @0x61B4C060

00000000:   10001600 41424344 457b0000 00000000 †....ABCDE{......
00000010:   43434343 20200400 f4                †CCCC  ...

Row Offsets:

0  1   2    4          9         13        16              22    24
10 00 1600 4142434445 7b000000 000000 4343434320 20 0400 f4
```

Figure 5-10: A data row containing all fixed-length columns

I was able to get the page contents by running the *DBCC PAGE* command, using the file and page number obtained using one of the methods that I described previously:

```
DBCC PAGE(tempdb, 1, 1231, 1);
```

Reading the output of *DBCC PAGE* takes a bit of practice. First, note that the output shows the data rows in groups of 4 bytes at a time. The shaded area in Figure 5-10 has been expanded to show the bytes in an expanded form.

The first byte is Status Bits A, and its value (0x10) indicates that only bit 4 is on, and because bit 5 is not on, we know the row has no variable-length columns. The second byte in the row (Status Bits B) is unused. The third and fourth bytes (1600) indicate the length of the fixed-length fields, which is also the column offset in which the *Ncol* value can be found. (As a multibyte numeric value, this information is stored in a byte-swapped form, so the value is really 0x0016, which translates to 22.) To know where in the row between offsets 4 and 22 each column actually is located, we need to know the offset of each column. In SQL Server 2000, the *syscolumns* system table has a column indicating the offset within the row. Although you can still select from the compatibility view called *syscolumns* in SQL Server 2005, the results you get back are not reliable. The offsets can be found in an undocumented view called *sys.system_internals_partition_columns* that we can then join to *sys.partitions* to get the information about the referenced objects and join to *sys.columns* to get other information about each column.

Here is a query to return basic column information, including the offset within the row for each column. I will use the same query for other tables later in this chapter, and I will refer to it as the "column detail query."

```
SELECT  c.name AS column_name, column_id, max_inrow_length,
          pc.system_type_id, leaf_offset
 FROM sys.system_internals_partition_columns pc
    JOIN sys.partitions p
      ON p.partition_id = pc.partition_id
    JOIN sys.columns c
          ON column_id = partition_column_id
             AND c.object_id = p.object_id
WHERE p.object_id=object_id('Fixed');
```

```
RESULTS:
column_name   column_id     max_inrow_length   system_type_id   leaf_offset
------------- ------------- ------------------ ---------------- -----------
Col1          1             5                  175              4
Col2          2             4                  56               9
Col3          3             3                  175              13
Col4          4             6                  175              16
```

So now we can find the data in the row for each column simply by using the offset value in the preceding results: the data for column *Col1* begins at offset 4, the data for column *Col2* begins at offset 9, and so on. As an *int*, the data in *Col2* (7b000000) must be byte-swapped to give the value 0x0000007b, which is equivalent to 123 in decimal.

Note that the 3 bytes of data for *Col3* are all zeros, representing an actual NULL in the column. Because the row has no variable-length columns, the row ends 3 bytes after the data for column *Col4*. The 2 bytes starting right after the fixed-length data at offset 22 (0400, which is byte-swapped to yield 0x0004) indicate that four columns are in the row. The last byte is the NULL bitmap. The value of 0xf4 is 11110100 in binary, and bits are shown from high order to low order. The low-order four bits represent the four columns in the table, 0100, which indicates that only the third column actually IS NULL. The high-order four bits are 1111 because those bits are unused. The NULL bitmap must have a multiple of eight bits, and if the number of columns is not a multiple of 8, some bits are unused.

Storage of Variable-Length Rows

Now let's look at the somewhat more complex case of a table with variable-length data. Each row has three *varchar* columns and two fixed-length columns:

```
CREATE TABLE Variable
(
Col1 char(3)       NOT NULL,
Col2 varchar(250)  NOT NULL,
Col3 varchar(5)    NULL,
Col4 varchar(20)   NOT NULL,
Col5 smallint      NULL
);
```

When this table is created, you should be able to execute the following queries against the *sys.indexes*, *sys.partitions*, *sys.system_internals_partition_columns*, and *sys.columns* views to receive the information similar to the results shown here:

```
SELECT object_id,  type_desc,
    indexproperty(object_id, name, 'minlen') as minlen
    FROM sys.indexes where object_id=object_id('Variable');

SELECT  name, column_id, max_inrow_length, pc.system_type_id, leaf_offset
 FROM sys.system_internals_partition_columns pc
    JOIN sys.partitions p
          ON p.partition_id = pc.partition_id
    JOIN sys.columns c
         ON column_id = partition_column_id AND c.object_id = p.object_id
WHERE p.object_id=object_id('Variable');

RESULTS:
object_id   type_desc   minlen
----------- ----------- -----------
69575286    HEAP        9


column_name   column_id     max_inrow_length   system_type_id   leaf_offset
------------- ------------- ------------------ ---------------- -----------
Col1          1             3                  175               4
Col2          2             250                167              -1
Col3          3             5                  167              -2
Col4          4             20                 167              -3
Col5          5             2                  52                7
```

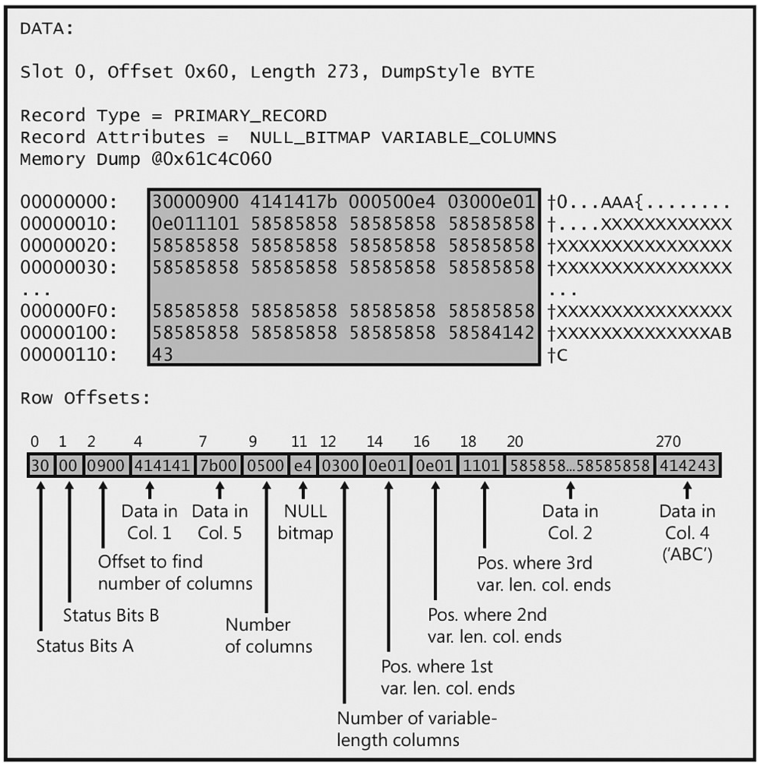
Now you can insert a row into the table as follows:

```
INSERT Variable VALUES
```

```
    ('AAA', REPLICATE('X', 250), NULL, 'ABC', 123);
```

The *REPLICATE* function is used here to simplify populating a column; this function builds a string of 250 *X*s to be inserted into *Col2*.

You can see the details of this row as stored on the page in the *DBCC PAGE* output in Figure 5-11. The location of the fixed-length columns can be found by using the *leaf_offset* value in *sys.system_internals_partition_columns*, in the preceding query results. In this table, *Col1* begins at offset 4 and *Col5* begins at offset 7. Variable-length columns are not shown in the query output with a specific byte offset because the offset can be different in each row. Instead, the row itself holds the ending position of each variable-length column within that row in a part of the row called the *Column Offset Array*. The query output shows that *Col2* has an *leaf_offset* value of –1, which means that *Col2* is the first variable-length column; an offset for *Col3* of –2 means that *Col3* is the second variable-length column, and an offset of –3 for *Col4* means that *Col4* is the third variable-length column.



```
DATA:

Slot 0, Offset 0x60, Length 273, DumpStyle BYTE

Record Type = PRIMARY_RECORD
Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Memory Dump @0x61C4C060

00000000:   30000900 4141417b 000500e4 03000e01 †0...AAA{........
00000010:   0e011101 58585858 58585858 58585858 †....XXXXXXXXXXXX
00000020:   58585858 58585858 58585858 58585858 †XXXXXXXXXXXXXXXX
00000030:   58585858 58585858 58585858 58585858 †XXXXXXXXXXXXXXXX
...                                             ...
000000F0:   58585858 58585858 58585858 58585858 †XXXXXXXXXXXXXXXX
00000100:   58585858 58585858 58585858 58584142 †XXXXXXXXXXXXXXAB
00000110:   43                                  †C

Row Offsets:
```

0 1 2 4 7 9 11 12 14 16 18 20 270
30 | 00 | 0900 | 414141 | 7b00 | 0500 | e4 | 0300 | 0e01 | 0e01 | 1101 | 585858...58585858 | 414243
Data in Col. 1
Data in Col. 5
NULL bitmap
Data in Col. 2
Data in Col. 4 ('ABC')
Offset to find number of columns
Pos. where 3rd var. len. col. ends
Status Bits B
Number of columns
Pos. where 2nd var. len. col. ends
Status Bits A
Pos. where 1st var. len. col. ends
Number of variable-length columns

Figure 5-11: A data row with variable-length columns

To find the variable-length columns in the data row itself, you first locate the column offset array in the row. Right after the 2-byte field indicating the total number of columns (0x0500) and the NULL bitmap with the value 0xe4, a 2-byte field exists with the value 0x0300 (or 3, decimal) indicating that three variable-length fields exist. Next comes the column offset array. Three 2-byte values indicate the ending position of each of the three variable-length columns: 0x0e01 is byte-swapped to 0x010e, so the first variable byte column ends at position 270. The next 2-byte offset is also 0x0e01, so that column has no length and has nothing stored in the variable data area. Unlike with fixed-length fields, if a variable-length field has a NULL value, it takes no room in the data row. SQL Server distinguishes between a *varchar* containing NULL and an empty string by determining whether the bit for the field is 0 or 1 in the NULL bitmap. The third 2-byte offset is 0x1101, which, when byte-swapped, gives us 0x0111. This means the row ends at position 273 (and is 273 bytes long).

The total storage space needed for a row depends on a number of factors. Variable-length columns add more overhead to a row, and their actual size is probably unpredictable. Even for fixed-length columns, the number of bytes of overhead can change depending on the number of columns in the table. In the example illustrated earlier in this chapter in Figure 5-2, I mentioned that 10 bytes of overhead exist if a row contains all fixed-length columns. For that table 10 is the correct number. The size of the NULL bitmap needs to be long enough to store a bit for every column in the row. In the Figure 5-2 example, the table has 11 columns, so the NULL bitmap needs to be 2 bytes. In the examples illustrated by Figures 5-10 and 5-11, the tables have fewer than eight columns, so the NULL bitmaps need only a single byte. Don't forget that the total row overhead must also include the 2 bytes for each row in the row offset table at the bottom of the page.

NULLS and Variable-Length Columns

As mentioned previously, fixed-length columns are always the same length, even if the column contains NULL. For variable-length columns, NULLs don't take any space in the variable-length data part of the row. However, as we saw in Figure 5-11, there is still a 2-byte column offset entry for each variable-length column, so we can't say that they take no space at all. However, if a zero-length value is stored at the end of the list of variable-length data columns, SQL Server does not store any information about it and does not include the 2 bytes in the column offset array. Let's look at an example.

The following table allows NULLs in each of its character columns, and they are all variable length. The only fixed-length column is the integer identity column:

```
CREATE TABLE dbo.null_varchar
    (
      id INT PRIMARY KEY IDENTITY(1,1),
      col1 VARCHAR(10) NULL,
      col2 VARCHAR(10) NULL,
      col3 VARCHAR(10) NULL,
      col4 VARCHAR(10) NULL,
      col5 VARCHAR(10) NULL,
      col6 VARCHAR(10) NULL,
      col7 VARCHAR(10) NULL,
      col8 VARCHAR(10) NULL,
      col9 VARCHAR(10) NULL,
      col10 VARCHAR(10) NULL
    );
GO
```

I'll insert four rows into this table. The first has a single character in the last *varchar* column, and NULLs in all the others. The second has a single character in the first *varchar* column, and NULLs in all the others. The third has a single character in the last *varchar* column, and empty strings in all the others. The fourth has a single character in the first *varchar* column, and empty strings in all the others:

```
SET NOCOUNT ON
INSERT INTO null_varchar(col10)
   SELECT 'a';
INSERT INTO null_varchar(col1)
   SELECT 'b';
INSERT INTO null_varchar
   SELECT '','','','','','','','','','c';
INSERT INTO null_varchar
   SELECT  'd','','','','','','','','','';
GO
```

Now I can use *DBCC IND* and *DBCC PAGE* (as shown previously) to look at the page containing these four rows.

Here is the first row (with the column offset array shaded):

```
Slot 0, Offset 0x60, Length 35, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 35
Memory Dump @0x66B4C060
00000000:   30000800 01000000 0b00fe03 0a002200 †0.........þ...".
00000010:   22002200 22002200 22002200 22002200 †".".".".".".".".
00000020:   230061††††††††††††††††††††††††††††††††#.a
```

There are nine entries in the column offset array with the value (after byte-swapping) of hex 22, or decimal 18, and one entry with the decimal value 19. The value of 18 for the first nine positions indicates that data ends in the same position as the column offset array ends, and SQL Server determines that this means those nine columns are empty. But empty could mean either NULL or an empty string. By looking at the NULL bitmap, in positions 11 and 12, we see fe03, which is hex 03fe after byte-swapping. Looking at this in binary we see 0000001111111110. The column positions are shown from right to left. This table has only 11 columns, so the last five bits in the NULL bitmap are ignored. The rest of the string indicates the first and last columns are not NULL, but all the other columns are NULL.

The 10th value in the column offset array is hex 23, or decimal 19, which means that data ends at offset 19, which contains the ASCII code 61, representing *a*.

Here is the second row (with the column offset array shaded):

```
Slot 1, Offset 0x83, Length 17, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 17
Memory Dump @0x66B4C083
00000000:   30000800 02000000 0b00fc07 01001100 †0.........ü.....
00000010:   62†††††††††††††††††††††††††††††††††††b
```

There are several important differences to note, between this row and the preceding one. First, the column offset array contains only a single value, which is the ending position for the first variable-length column. The 1100 bytes are byte-swapped to 0011, and converted to 17 decimal, which is the offset where the ASCII code for *b* (that is, 62) is located. Immediately preceding the column offset array is the 2-byte value indicating the number of variable-length columns. The first row had a hex value of 000a here, indicating 10 variable-length columns. The second row has 0001, which means only one of the variable-length columns is actually stored in the row. We just saw that zero-length columns prior to columns containing data do use the column offset array, but in this case, because all the zero-length columns are after the non-NULL, only the non-NULL column is represented here. If you look at the NULL bitmap, you'll see fc07, which is hex 07fc after byte-swapping. Looking at this in binary, we see 0000011111111100, indicating that the first two columns are not NULL, but all the rest are.

If you look at the rows containing empty strings instead of NULLs, the output should be exactly the same, except for the NULL bitmap. Here is the third row (slot 2) and the fourth row (slot 3), with the NULL bitmaps shaded:

```
Slot 2, Offset 0x94, Length 35, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 35
Memory Dump @0x66B4C094
00000000:   30000800 03000000 0b000000 0a002200 †0.............".
00000010:   22002200 22002200 22002200 22002200 †".".".".".".".".
00000020:   230063†††††††††††††††††††††††††††††††#.c


Slot 3, Offset 0xb7, Length 17, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 17
Memory Dump @0x66B4C0B7
00000000:   30000800 04000000 0b000000 01001100 †0...............
00000010:   63†††††††††††††††††††††††††††††††††††d
```

For both the third and fourth rows, the NULL bitmap is all zeros, indicating that none of the columns are NULL. The first and third rows differ only in the actual character value stored and in the NULL bitmap. The second and fourth rows differ in the same way.

If we insert a row with all NULLs in the *varchar* columns, the row storage changes a bit more. Here is what it would look like:

```
Slot 4, Offset 0xc8, Length 12, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Record Size = 12
Memory Dump @0x66B4C0C8
00000000:   10000800 05000000 0b000000 ††††††††††.......
```

This row looks just like an all fixed-length row and ends right after the NULL bitmap. Bit 5 in the first byte (Status Bits A) has been set to 0 to indicate that there are no variable-length columns stored in this row.

Storage of Date and Time Data

I described the storage of date and time data types earlier in this chapter, and now that we've had some practice looking at the actual on-disk storage, let's look at some date and time data. The following table stores all the different data and time data types in a single row, and all of the different possible scales for time data. (Remember that *datetime2* and *datetimeoffset* can also indicate a scale for the time component, but the time values look no different than the time values stored with the simple *time* data type.) The table also includes single-column character values, which I use just so I can find the other values easily in the single row of hex data that *DBCC PAGE* gives me:

```
CREATE TABLE times (
        a char(1),
        dt1 datetime,
        b char(1),
        sd smalldatetime,
        c char(1),
        dt2 datetime2,
        d char(1),
        dt date,
        e char(1),
        dto datetimeoffset,
        f char(1),
        t time,
        g char(1),
        t0 time(0),
        h char(1),
        t1 time(1),
        i char(1),
        t2 time(2),
        j char(1),
        t3 time(3),
        k char(1),
        t4 time(4),
        l char(1),
        t5 time(5),
        m char(1),
        t6 time(6),
        n char(1),
        t7 time(7));
GO
```

Now I'll insert one one-row data, with the same time value provided for each date or time column. The data types that need a date component assume a default date of January 1, 1900:

```
INSERT INTO times
SELECT
    'a', '01:02:03.123',
    'b', '01:02:03.123',
    'c', '01:02:03.123',
    'd', '01:02:03.123',
    'e', '01:02:03.123',
    'f', '01:02:03.123',
    'g', '01:02:03.123',
    'h', '01:02:03.123',
    'i', '01:02:03.123',
    'j', '01:02:03.123',
    'k', '01:02:03.123',
    'l', '01:02:03.123',
    'm', '01:02:03.123',
    'n', '01:02:03.123';
```

Here is the *DBCC PAGE* output for this row. I have shaded the single-character column data to use as dividers:

```
00000000:   10005800 61090b11 00000000 00623e00 †..X.a .......b>.
00000010:   00006330 7c27ab08 5b950a64 5b950a65 †..c0|'".[?.d[?.e
00000020:   307c27ab 085b950a 00006630 7c27ab08 †0|'".[?...f0|'".
00000030:   678b0e00 686f9100 6958ae05 6a73cf38 †g?..ho?.iX®.jsÏ8
00000040:   006b7e1a 38026cec 08311600 6d3859ea †.k~.8.lì.1..m8Yê
00000050:   dd006e30 7c27ab08 1c000000 0000††††††Ý.n0|'".......
```

Table 5-8 shows the translation into decimal format for each of these values. Here are some points to notice:

- For the *datetime* and *smalldatetime* data types, the date value is stored as 0, meaning that the date is the base of 'January 1, 1900'. For the other types that store a date, the date value is stored as 693595, which represents the number of days after the new internal base date of January 1, 0001. To compute the corresponding date, you can use the *dateadd* function:

  ```
  SELECT DATEADD(dd, 693595, CAST('0001/1/1' AS datetime2));
  ```

- This will return the value '1900-01-01' 00:00:00.00', which is the default when no date is specified.

- The fractional seconds component is the last *N* digits of the time component, where *N* is the scale of the time data, as listed in the table definition. So for the *time(7)* value, the fractional seconds are .1230000; for the *time(4),* the fractional seconds are .1230; for the *time(1)* value, the fractional seconds are .1; and for the *time(0)* value, there are no fractional sections.

- Whatever remains in the time portion after the appropriate number of digits are removed for the fractional seconds is the hours, minutes, and seconds value. Because the same time value was used for all the columns in the table, the time values all start with the same four digits: 3723. Previously, I showed you the formula for converting a time value to an integer; here, I'll do the reverse, using the modulo operator (%) and integer division. SQL Server uses the following conversions to determine the hours, minutes, and seconds from 3723:

```
SELECT hours =  (3723 / 60) / 60;
SELECT minutes = (3723 / 60) % 60;
SELECT seconds = 3723 % 60;

RESULT:
hours
-----------
1

minutes
-----------
2

seconds
-----------
3
```

- The column storing *datetimeoffset* data has 2 extra bytes to store the *timezone* offset. Two bytes are needed because the offset is stored as the number of hours and minutes (1 byte for each) from Coordinated Universal Time (UTC).

Table 5-8: Translation of Various Date and Time Values

Column Name	Data Type and Bytes Used	Value Stored in Row	Byte-Swapped Date	Time	Decimal Values Date	Time
dt1	*Datetime* -8-	090b110000 000000	00 00 00 00	00 11 0b 09	0	1116937
sd	*Small datetime* -4-	3e000000	00 00	00 3e	0	62
dt2	*datetime2* -8-	307c27ab0 85b950a	0a 95 5b	08 ab 27 7c 30	693595	37231230000
dt	*date* -3-	5b950a	0a 95 5b	(none)	693595	(none)
dto	*datetime offset* -10-	307c27ab08 5b950a00 00	0a 95 5b	08 ab 27 7c 30	693595	37231230000
t	*time* -5-	307c27ab08	(none)	08 ab 27 7c 30	(none)	37231230000
t0	time(0) -3-	8b0e00	(none)	00 0e 8b	(none)	3723
t1	time(1) -3-	6f9100	(none)	00 91 6f	(none)	37231
t2	time(2) -3-	58ae05	(none)	05 ae 58	(none)	372312
t3	time(3) -4-	73cf3800	(none)	00 38 cf 73	(none)	3723123
t4	time(4) -4-	7e1a3802	(none)	02 38 1a 7e	(none)	37231230
t5	time(5) -5-	ec08311600	(none)	00 16 31 08 ec	(none)	372312300

t6	time(6) -5-	3859eadd00	(none)	00 dd ea 59 38	(none)	3723123000
t7	time(7) -5-	307c27ab08	(none)	08 ab 27 7c 30	(none)	37231230000

Storage of sql_variant Data

The *sql_variant* data type provides support for columns that contain any or all of the SQL Server base data types except LOBs and variable-length columns with the MAX qualifier, *rowversion* (*timestamp*), XML, and the types that can't be defined for a column in a table, namely *cursor* and *table*. For instance, a column can contain a *smallint* value in some rows, a *float* value in others, and a *char* value in the remainder.

This feature was designed to support what appears to be semistructured data in products sitting above SQL Server. This semistructured data exists in conceptual tables that have a fixed number of columns of known data types and one or more optional columns whose type might not be known in advance. An example is e-mail messages in Microsoft Office Outlook and Microsoft Exchange. With the *sql_variant* data type, you can pivot a conceptual table into a real, more compact table with sets of property-value pairs. Here is a graphical example: the conceptual table shown in Table 5-9 has three rows of data. The fixed columns are the ones that exist in every row. Each row can also have values for one or more of the three different properties, which have different data types.

Table 5-9: A Conceptual Table with an Arbitrary Number of Columns and Data Types

Row	Fixed Columns	Property 1	Property 2	Property 3
row -1	XXXXXX	value-11		value -13
row -2	YYYYYY	value-22		
row -3	ZZZZZZ	value-31	value-32	

This can be pivoted into Table 5-10, where the fixed columns are repeated for each different property that appears with those columns. The column called *value* can be represented by *sql_variant* data and be a different data type for each different property.

Table 5-10: Semistructured Data Stored Using the sql_variant Data Type

Fixed Columns	Property	Value
XXXXXX	property-1	value-11
XXXXXX	property-3	value-13
YYYYYY	property-2	value-22
ZZZZZZ	property-1	value-31
ZZZZZZ	property-2	value-32

Internally, columns of type *sql_variant* are always considered variable length. Their storage structure depends on the type of data, but the first byte of every *sql_variant* field always indicates the actual data type being used in that row.

I'll create a simple table with a *sql_variant* column and insert a few rows into it so we can observe the structure of the *sql_variant* storage.

```
USE testdb;
GO
CREATE TABLE variant (a int, b sql_variant);
GO
INSERT INTO variant VALUES (1, 3);
INSERT INTO variant VALUES (2, 3000000000);
INSERT INTO variant VALUES (3, 'abc');
INSERT INTO variant VALUES (4, current_timestamp);
```

SQL Server decides what data type to use in each row based on the data supplied. For example, the *3* in the first *INSERT* is assumed to be an integer. In the second *INSERT*, the *3000000000* is larger than the biggest possible integer, so SQL Server assumes a decimal with a precision of 10 and a scale of 0. (It could have used a *bigint*, but that would need more storage space.) We can now use *DBCC IND* to find the first page of the table and use *DBCC PAGE* to see its contents as follows:

```
DBCC IND (testdb, variant, -1);
-- (I got a value of file 1, page 2508 for the data page in this table)
GO
DBCC TRACEON (3604);
DBCC PAGE (testdb, 1, 2508, 1);
```

Figure 5-12 shows the contents of the four rows. I won't go into the details of every single byte because most are the same as what we've already examined.

```
DATA:

Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 21

Memory Dump @0x62B7C060
00000000: 30000800 01000000 02000001 00150038 †0..............8
00000010: 01030000 00††††††††††††††††††††††††††††.....

Slot 1, Offset 0x75, Length 24, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 24
Memory Dump @0x62B7C075

00000000: 30000800 02000000 02000001 0018006c †0..............l
00000010: 010a0001 005ed0b2 †††††††††††††††††††††.....^Ð²

Slot 2, Offset 0x8d, Length 26, DumpStyle BYTE
Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 26
Memory Dump @0x62B7C08D

00000000: 30000800 03000000 02000001 001a00a7 †0..............§
00000010: 01401f08 d0003461 6263††††††††††††††††.@..Ð.4abc

Slot 3, Offset 0xa7, Length 25, DumpStyle BYTE

Record Type = PRIMARY_RECORD       Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 25
Memory Dump @0x62B7C0A7

00000000: 30000800 04000000 02000001 0019003d †0..............=
00000010: 0183de14 01299b00 00††††††††††††††††††..Þ..)...

OFFSET TABLE:

Row - Offset
3 (0x3) - 167 (0xa7)
2 (0x2) - 141 (0x8d)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

Figure 5-12: Rows containing sql_variant data

The difference between the three rows starts at bytes 13 to 14, which indicate the position where the first variable-length column ends. Because there is only one variable-length column, this is also the length of the row. The *sql_variant* data begins at byte 15. Byte 15 is the code for the data type. You can find the codes in the *system_type_id* column of the *sys.types* catalog view. I've reproduced the relevant part of that view here:

```
system_type_id   name
```

```
---------------- ----------------------
34               image
35               text
36               uniqueidentifier
40               date
41               time
42               datetime2
43               datetimeoffset
48               tinyint
52               smallint
56               int
58               smalldatetime
59               real
60               money
61               datetime
62               float
98               sql_variant
99               ntext
104              bit
106              decimal
108              numeric
122              smallmoney
127              bigint
165              varbinary
167              varchar
173              binary
175              char
189              timestamp
231              nvarchar
231              sysname
239              nchar
240              hierarchyid
240              geometry
240              geography
241              xml
```

In our table, we have the data types 38 hex (which is 56 decimal and *int*), 6C hex (which is 108 decimal, which is numeric), A7 hex (which is 167 decimal and *varchar*), and 3D hex (which is 61 decimal and *datetime*). Following the byte for data type is a byte representing the version of the *sql_variant* format, and that is always 1 in SQL Server 2008. Following the version, there can be one of the following four sets of bytes:

- For numeric and decimal: 1 byte for the precision and 1 byte for the scale
- For strings: 2 bytes for the maximum length and 4 bytes for the collation ID
- For *binary* and *varbinary*: 2 bytes for the maximum length
- For all other types: no extra bytes

These bytes are then followed by the actual data in the *sql_variant* column.

Constraints

Constraints provide a powerful yet easy way to enforce the data integrity in your database. Data integrity comes in three forms:

Entity integrity Ensures that a table has a primary key. In SQL Server 2008, you can guarantee entity integrity by defining PRIMARY KEY or UNIQUE constraints or by building unique indexes. Alternatively, you can write a trigger to enforce entity integrity, but this is usually far less efficient.

Domain integrity Ensures that data values meet certain criteria. In SQL Server 2008, domain integrity can be guaranteed in several ways. Choosing appropriate data types can ensure that a data value meets certain conditions—for example, that the data represents a valid date. Other approaches include defining CHECK constraints or FOREIGN KEY constraints, or writing a trigger. You might also consider DEFAULT constraints as an aspect of enforcing domain integrity.

Referential integrity Enforces relationships between two tables, a referenced table, and a referencing table. SQL Server allows you to define FOREIGN KEY constraints to enforce referential integrity, and you can also write triggers for enforcement. It's crucial to note that there are always two sides to referential integrity enforcement. If data is updated or deleted from the referenced table, referential integrity ensures that any data in the referencing table that refers to the changed or deleted data is handled in some way. On the other side, if data is updated or inserted into the referencing table, referential integrity ensures that the new data matches a value in the referenced table.

In this section, I'll briefly describe some of the internal aspects of managing constraints. Constraints are also called *declarative data integrity* because they are part of the actual table definition. This is in contrast to *programmatic data integrity*, which uses stored procedures or triggers.

Here are the five types of constraints:

- PRIMARY KEY
- UNIQUE
- FOREIGN KEY
- CHECK
- DEFAULT

You might also sometimes see the *IDENTITY* property and the nullability of a column described as constraints. I typically don't consider these attributes to be constraints; instead, I think of them as properties of a column, for two reasons. First, each constraint has its own row in the *sys.objects* catalog view, but *IDENTITY* and nullability information is not available in *sys.objects*, only in *sys.columns* and *sys.identity_columns*. This makes me think that these properties are more like data types, which are also viewable through *sys.columns*. Second, when you use the *SELECT INTO* command to make a copy of a table, all column names and data types are copied, as well as *IDENTITY* information and column nullability, but constraints are *not* copied to the new table. This makes me think that *IDENTITY* and nullability are more a part of the actual table structure than the constraints are.

Constraint Names and Catalog View Information

The following simple *CREATE TABLE* statement, which includes a primary key on the table, creates a PRIMARY KEY constraint along with the table, and the constraint has a very cryptic-looking name:

```
CREATE TABLE customer
(
cust_id      int          IDENTITY  NOT NULL  PRIMARY KEY,
cust_name    varchar(30) NOT NULL
);
```

If you don't supply a constraint name in the *CREATE TABLE* or *ALTER TABLE* statement that defines the constraint, SQL Server comes up with a name for you.

The constraint produced from the preceding simple statement has a name very similar to the nonintuitive name *PK__customer__3BD0198E35BCFE0A*. (The hexadecimal number at the end of the name most likely will be different for a *customer* table that you create.) All types of single-column constraints use this naming scheme, which I'll explain shortly. The advantage of explicitly naming your constraint rather than using the system-generated name is greater clarity. The constraint name is used in the error message for any constraint violation, so creating a name such as *CUSTOMER_PK* probably makes more sense to users than a name such as *PK__customer__0856260D*. You should choose your own constraint names if such error messages are visible to your users. The first two characters *(PK)* show the constraint type—*PK* for PRIMARY KEY, *UQ* for UNIQUE, *FK* for FOREIGN KEY, *CK* for CHECK, and *DF* for DEFAULT. Next are two underscore characters, which are used as a separator.

Tip You might be tempted to use one underscore to conserve characters and to avoid having to truncate as much. However, it's common to use a single underscore in a table name or a column name, both of which appear in the constraint name. Using two underscore characters distinguishes the kind of a name it is and where the separation occurs.

Note Constraint names are schema-scoped, which means they all share the same namespace and hence must be unique within a schema. Within a schema, you cannot have two tables with the same name for any of their constraints.

Next comes the table name (*customer*), which is limited to 116 characters for a PRIMARY KEY constraint and slightly fewer characters for all other constraint names. For all constraints other than PRIMARY KEY and UNIQUE, there are two more underscore characters for separation, followed by the next sequence of characters, which is the column name. The column name is truncated to five characters if necessary. If the column name has fewer than five characters, the length of the table name portion can be slightly longer.

Finally, the hexadecimal representation of the object ID for the constraint comes after another separator. This value is used in the *object_id* column of the *sys.objects* catalog view. Object names are limited to 128 characters in SQL Server 2008, so the total length of all the portions of the constraint name must also be less than or equal to 128.

Several catalog views contain constraint information. They all inherit the columns from the *sys.objects* view and include additional columns specific to the type of constraint. These views are

- *sys.key_constraints*
- *sys.check_constraints*
- *sys.default_constraints*
- *sys.foreign_keys*

The *parent_object_id* column, which indicates which object contains the constraint, is actually part of the base *sys.objects* view, but for objects that have no "parent," this column is 0.

Constraint Failures in Transactions and Multiple-Row Data Modifications

Many bugs occur in application code because developers don't understand how the failure of a constraint affects a multiple-statement transaction declared by the user. The biggest misconception is that any error, such as a constraint failure, automatically aborts and rolls back the entire transaction. On the contrary, after an error is raised, it's up to the transaction to proceed and ultimately commit or to roll back. This feature provides the developer with the flexibility to decide how to handle errors. (The semantics are also in accordance with the ANSI SQL-92 standard for COMMIT behavior.)

Because many developers have handled transaction errors incorrectly and because it can be tedious to add an error check after every command, SQL Server includes a SET option called XACT_ABORT that causes SQL Server to abort a transaction if it encounters any error during the transaction. The default setting is OFF, which is consistent with ANSI-standard behavior.

A final comment about constraint errors and transactions: a single data modification statement (such as an *UPDATE* statement) that affects multiple rows is automatically an atomic operation, even if it's not part of an explicit transaction. If such an *UPDATE* statement finds 100 rows that meet the criteria of the WHERE clause but one row fails because of a constraint violation, no rows will be updated. I discuss implicit and explicit transactions a bit more in Chapter 10.

The Order of Integrity Checks

The modification of a given row fails if any constraint is violated or if a trigger rolls back the operation. As soon as a failure in a constraint occurs, the operation is aborted, subsequent checks for that row aren't performed, and no triggers fire for the row. Hence, the order of these checks can be important, as the following list shows:

1. Defaults are applied as appropriate.
2. NOT NULL violations are raised.
3. CHECK constraints are evaluated.
4. FOREIGN KEY checks of *referencing* tables are applied.
5. FOREIGN KEY checks of *referenced* tables are applied.
6. The UNIQUE and PRIMARY KEY constraints are checked for correctness.
7. Triggers fire.

Altering a Table

SQL Server 2008 allows existing tables to be modified in several ways. Using the *ALTER TABLE* command, you can make the following types of changes to an existing table:

- Change the data type or the *NULL* property of a single column.
- Add one or more new columns, with or without defining constraints for those columns.
- Add one or more constraints.
- Drop one or more constraints.
- Drop one or more columns.
- Enable or disable one or more constraints (applies only to CHECK and FOREIGN KEY constraints).
- Enable or disable one or more triggers.
- Rebuild a table or a partition to change the compression settings or remove fragmentation. (Fragmentation is discussed in Chapter 6, and compression is discussed in Chapter 7.)
- Change the lock escalation behavior of a table. (Locks and lock escalation are discussed in Chapter 10.)

Changing a Data Type

By using the ALTER COLUMN clause of *ALTER TABLE*, you can modify the data type or the *NULL* property of an existing column. But be aware of the following restrictions:

- The modified column can't be a *text*, *image*, *ntext*, or *rowversion* (*timestamp)* column.
- If the modified column is the *ROWGUIDCOL* for the table, only *DROP ROWGUIDCOL* is allowed; no data type changes are allowed.
- The modified column can't be a computed or replicated column.
- The modified column can't have a PRIMARY KEY or FOREIGN KEY constraint defined on it.
- The modified column can't be referenced in a computed column.
- The modified column can't have the type changed to *timestamp*.
- If the modified column participates in an index, the only type changes that are allowed are increasing the length of a variable-length type (for example, *varchar*(10) to *varchar*(20)), changing nullability of the column, or both.
- If the modified column has a UNIQUE or CHECK constraint defined on it, the only change allowed is altering the length of a variable-length column. For a UNIQUE constraint, the new length must be greater than the old length.
- If the modified column has a default defined on it, the only changes that are allowed are increasing or decreasing the length of a variable-length type, changing nullability, or changing the precision or scale.
- The old type of the column should have an allowed implicit conversion to the new type.
- The new type always has ANSI_PADDING semantics if applicable, regardless of the current setting.
- If conversion of an old type to a new type causes an overflow (arithmetic or size), the *ALTER TABLE* statement is aborted.

Here's the syntax and an example of using the ALTER COLUMN clause of the *ALTER TABLE* statement:

```
SYNTAX:

ALTER TABLE table-name ALTER COLUMN column-name
        { type_name [ ( prec [, scale] ) ] [COLLATE <collation name> ]
          [ NULL | NOT NULL ]
          |  {ADD | DROP} {ROWGUIDCOL | PERSISTED}  }
EXAMPLE:

/* Change the length of the emp_lname column in the employee
   table from varchar(15) to varchar(30) */
ALTER TABLE employee
   ALTER COLUMN emp_name varchar(30);
```

Adding a New Column

You can add a new column, with or without specifying column-level constraints. If the new column doesn't allow NULLs, isn't an identity column, and isn't a *rowversion* (or *timestamp* column), the new column must have a default constraint defined (unless no data is in the table yet). SQL Server populates the new column in every row with a NULL, the appropriate identity or *rowversion* value, or the specified default. If the newly added column is nullable and has a default constraint, the existing rows of the table are not filled with the default value, but rather with NULL values. You can override this restriction by using the WITH VALUES clause so that the existing rows of the table are filled with the specified default value.

Adding, Dropping, Disabling, or Enabling a Constraint

You can use *ALTER TABLE* to add, drop, enable, or disable a constraint. The trickiest part of using *ALTER TABLE* to manipulate constraints is that the word *CHECK* can be used in three different ways:

- To specify a CHECK constraint.
- To defer the checking of a newly added constraint. In the following example, we're adding a constraint to validate that *cust_id* in *orders* matches a *cust_id* in *customer*, but we don't want the constraint applied to existing data:

  ```
  ALTER TABLE orders
     WITH NOCHECK
     ADD FOREIGN KEY (cust_id) REFERENCES customer (cust_id);
  ```

  > **Note** Instead of using WITH NOCHECK, I could use WITH CHECK to force the constraint to be applied to existing data, but that's unnecessary because it's the default behavior.

- To enable or disable a constraint. In this example, we enable all the constraints on the *employee* table:

  ```
  ALTER TABLE employee
     CHECK CONSTRAINT ALL;
  ```

The only types of constraints that can be disabled are CHECK constraints and FOREIGN KEY constraints, and disabling tells SQL Server not to validate new data as it is added or updated. You should use caution when disabling and re-enabling constraints. If a constraint was part of the table when the table was created or was added to the table using the WITH CHECK option, SQL Server knows that the data conforms to the data integrity requirements of the constraint. The SQL Server Query Optimizer can then take advantage of this knowledge in some cases. For example, if you have a constraint that requires *col1* to be greater than 0, and then an application submits a query looking for all rows where *col1* < 0, if the constraint has always been in effect, the Optimizer knows that no rows can satisfy this query and the plan is a very simple plan. However, if the constraint has been disabled and re-enabled without using the WITH CHECK option, there is no guarantee that some of the data in the table won't meet the integrity requirements. You might not have any data less than or equal to 0, but the Optimizer cannot know that when it is devising the plan; all the Optimizer knows is that the constraint cannot be trusted. The catalog views *sys.check_constraints* and *sys.foreign_keys* each have a column called *is_not_trusted*. If you re-enable a constraint and don't use the WITH CHECK option to tell SQL Server to revalidate all existing data, the *is_not_trusted* column is set to 1.

Although you cannot use *ALTER TABLE* to disable or enable a PRIMARY KEY or UNIQUE constraint, you can use the *ALTER INDEX* command to disable the associated index. I'll discuss *ALTER INDEX* in Chapter 6. You can use *ALTER TABLE* to drop PRIMARY KEY and UNIQUE constraints, but you need to be aware that dropping one of these constraints automatically drops the associated index. In fact, the only way to drop those indexes is by altering the table to remove the constraint.

> **Note** You can't use *ALTER TABLE* to modify a constraint definition. You must use *ALTER TABLE* to drop the constraint and then use *ALTER TABLE* to add a new constraint with the new definition.

Dropping a Column

You can use *ALTER TABLE* to remove one or more columns from a table. However, you can't drop the following columns:

- A replicated column
- A column used in an index
- A column used in a CHECK, FOREIGN KEY, UNIQUE, or PRIMARY KEY constraint
- A column associated with a default defined using the *DEFAULT* keyword or bound to a default object
- A column to which a rule is bound

You can drop a column using the following syntax:

```
ALTER TABLE table-name
    DROP COLUMN column-name [, next-column-name]...
```

Note Notice the syntax difference between dropping a column and adding a new column: the word *COLUMN* is required when dropping a column but not when you add a new column to a table.

Enabling or Disabling a Trigger

You can enable or disable one or more (or all) triggers on a table using the *ALTER TABLE* command.

Internals of Altering Tables

Note that not all the *ALTER TABLE* variations require SQL Server to change every row when the *ALTER TABLE* is issued. SQL Server can carry out an *ALTER TABLE* command in three basic ways:

- It might need to change only metadata.
- It might need to examine all the existing data to make sure it is compatible with the change but only needs to make changes to metadata.
- It might need to change every row physically.

In many cases, SQL Server can just change the metadata (primarily the data seen through *sys.columns*) to reflect the new structure. In particular, the data isn't touched when a column is dropped, when a new column is added and NULL is assumed as the new value for all rows, when the length of a variable-length column is increased, or when a non-nullable column is changed to allow NULLs. The fact that data isn't touched when a column is dropped means that the disk space of the column is not reclaimed. You might have to reclaim the disk space of a dropped column when the row size of a table approaches or has exceeded its limit. You can reclaim space by creating a clustered index on the table or rebuilding an existing clustered index by using *ALTER INDEX*, as we'll see in Chapter 6.

Some changes to a table's structure require that the data be examined but not modified. For example, when you change the nullability property to disallow NULLs, SQL Server must first make sure there are no NULLs in the existing rows. A variable-length column can be shortened when all the existing data is within the new limit, so the existing data must be checked. If any rows have data longer than the new limit specified in the *ALTER TABLE*, the command fails. So you do need to be aware that for a huge table, this can take some time. Changing a fixed-length column to a shorter type, such as changing an *int* column to *smallint* or changing a *char(10)* to *char(8)*, also requires examining all the data to verify that all the existing values can be stored in the new type. However, even though the new data type takes up fewer bytes, the rows on the physical pages are not modified. If you have created a table with an *int* column, which needs 4 bytes in each row, all rows will use the full 4 bytes. After altering the table to change the *int* to *smallint*, we are restricted in the range of data values we can insert, but the rows continue to use 4 bytes for each value, instead of the 2 bytes that *smallint* requires. You can verify this by using the *DBCC PAGE* command. Changing a *char(10)* to *char(8)* displays similar behavior, and the rows continue to use 10 bytes for that column, but only 8 are allowed to have new data inserted. It is not until the table is rebuilt by creating or re-creating a clustered index that the *char(10)* columns are actually re-created to become *char(8).*

Other changes to a table's structure require SQL Server to change every row physically, and as it makes the changes, it has to write the appropriate records to the transaction log, so these changes can be extremely resource intensive for a large table. One example of this type of change is adding a new column that doesn't allow NULL, in which case you must specify a default column value. SQL Server physically adds the column with the default to each row. Note that when adding a new column that allows NULLs, the change is a metadata-only operation.

Another negative side effect of altering tables happens when a column is altered to increase its length. In this case, the old column is not actually replaced. Rather, a new column is added to the table, and *DBCC PAGE* shows you that the old data is still there. I'll let you explore the page dumps for this situation on your own, but we can see some of this unexpected behavior by just looking at the column offsets using the column detail query that I showed you earlier in this chapter.

First, create a table with all fixed-length columns, including a *smallint* in the first position:

```
CREATE TABLE change
(col1 smallint, col2 char(10), col3 char(5));
```

Now look at the column offsets:

```
SELECT  c.name AS column_name, column_id, max_inrow_length, pc.system_type_id, leaf_offset
 FROM sys.system_internals_partition_columns pc
    JOIN sys.partitions p
      ON p.partition_id = pc.partition_id
    JOIN sys.columns c
         ON column_id = partition_column_id
            AND c.object_id = p.object_id
WHERE p.object_id=object_id('change');

RESULTS:
column_name   column_id   max_inrow_length   system_type_id leaf_offset
------------- ----------- ------------------ -------------- -----------
col1          1           2                 52            4
col2          2           10                175           6
col3          3           5                 175           16
```

Now change *smallint* to *int*:

```
ALTER TABLE change
   ALTER COLUMN col1 int;
```

Finally, run the column detail query again to see that *col1* now starts much later in the row and that no column starts at offset 4 immediately after the row header information. This new column creation due to an *ALTER TABLE* takes place even before any data has been placed in the table:

```
column_name   column_id   max_inrow_length   system_type_id   leaf_offset
------------- ----------- ------------------ ---------------- -----------
col1          1           4                  56               21
col2          2           10                 175              6
col3          3           5                  175              16
```

Another drawback to the behavior of SQL Server in not actually dropping the old column is that we are now more severely limited in the size of the row. The row size now includes the old column, which is no longer usable or visible (unless you use *DBCC PAGE*). For example, if I create a table with a couple of large fixed-length character columns, as shown here, I can then ALTER the *char(2000)* column to be *char(3000):*

```
CREATE TABLE bigchange
(col1 smallint, col2 char(2000), col3 char(1000));

ALTER TABLE bigchange
   ALTER COLUMN col2 char(3000);
```

At this point, the length of the rows should be just over 4,000 bytes because there is a 3,000-byte column, a 1,000-byte column, and a *smallint*. However, if I try to add another 3,000-byte column, it fails:

```
ALTER TABLE bigchange
   ADD col4 char(3000);

Msg 1701, Level 16, State 1, Line 1
Creating or altering table 'bigchange' failed because the minimum row size would be 9009,
including 7 bytes of internal overhead. This exceeds the maximum allowable table row size
of8060 bytes.
```

However, if I just create a table with two 3,000-byte columns and a 1,000-byte column, there is no problem:

```
CREATE TABLE nochange
```

```
(col1 smallint, col2 char(3000), col3 char(1000), col4 char(3000));
```

Note that there is no way to *ALTER* a table to rearrange the logical column order or to add a new column in a particular position in the table. A newly added column always gets the next highest *column_id* value. When you execute *SELECT ** on a table or look at the metadata with *sp_help*, the columns are always returned in *column_id* order. If you need a different order, you have several options:

- Don’t use *SELECT **; always *SELECT* a list of columns in the order that you want to have them returned.
- Create a view on the table that *SELECT*s the columns in the order you want them, and then you can *SELECT ** from the view or run *sp_help* on the view.
- Create a new table, copy the data from the old table, drop the old table, and rename the new table to the old name. Don’t forget to re-create all constraints, indexes, and triggers.

You might think that Management Studio can add a new column in a particular position or rearrange the column order, but this is not true. Behind the scenes, the tool is actually using the preceding third option and creating a completely new table with all new indexes, constraints, and triggers. If you wonder why simply adding a new column to an existing (large) table is taking a long time, this is probably the reason.

Heap Modification Internals

We’ve seen how SQL Server stores data in a heap. Now we’ll look at what SQL Server actually does internally when your heap data is modified. Modifying data in an index, which includes a table with a clustered index, is a completely separate topic and will be covered in detail in Chapter 6. As a rule of thumb, you should always have a clustered index on a table. There are some cases in which you might be better off with a heap, such as when the most important factor is the speed of *INSERT* operations, but until you do thorough testing to establish that you have one of these cases, it’s better to have a clustered index than to have no organization to your data at all. In Chapter 6, you’ll see the benefits and tradeoffs of clustered and nonclustered indexes and examine some guidelines for their use. For now, we’ll look only at how SQL Server deals with the data modifications on tables without clustered indexes.

Allocation Structures

As discussed in Chapter 3, SQL Server allocates one or more IAM pages for each object, to keep track of which extents in each file belong to that object. If the table is a heap, using the IAMs is the only way for SQL Server to find all the extents belonging to the table, because the individual data pages of a table are not connected in a doubly linked list, the way they are if the table has a clustered index. Pages at each level of an index are linked, and because the data is considered the leaf level of a clustered index, SQL Server does maintain the linkage. However, for a heap, no such linked list connects the pages to each other. The only way that SQL Server determines which pages belong to a table is by inspecting the IAMs for the table.

Another special allocation structure is particularly useful when SQL Server is performing data modification operations, and that is the Page Free Space (PFS) structure. PFS pages keep track of how much space is free on each page, so that *INSERT* operations in a heap know where space is available for the new data, and *UPDATE* operations know where a row can be moved. I briefly mentioned PFS pages in Chapter 3, and I told you that these pages contained 1 byte for each page in a 8,088-page range of a file. This is much less dense than Global Allocation Maps (GAMs), Shared Global Allocation Maps (SGAMs), and IAMs, which contain one bit per extent.) Figure 5-13 shows the structure of a byte on a PFS page. Only the last three bits are used to indicate the page fullness, and four of the other five bits each have a meaning.

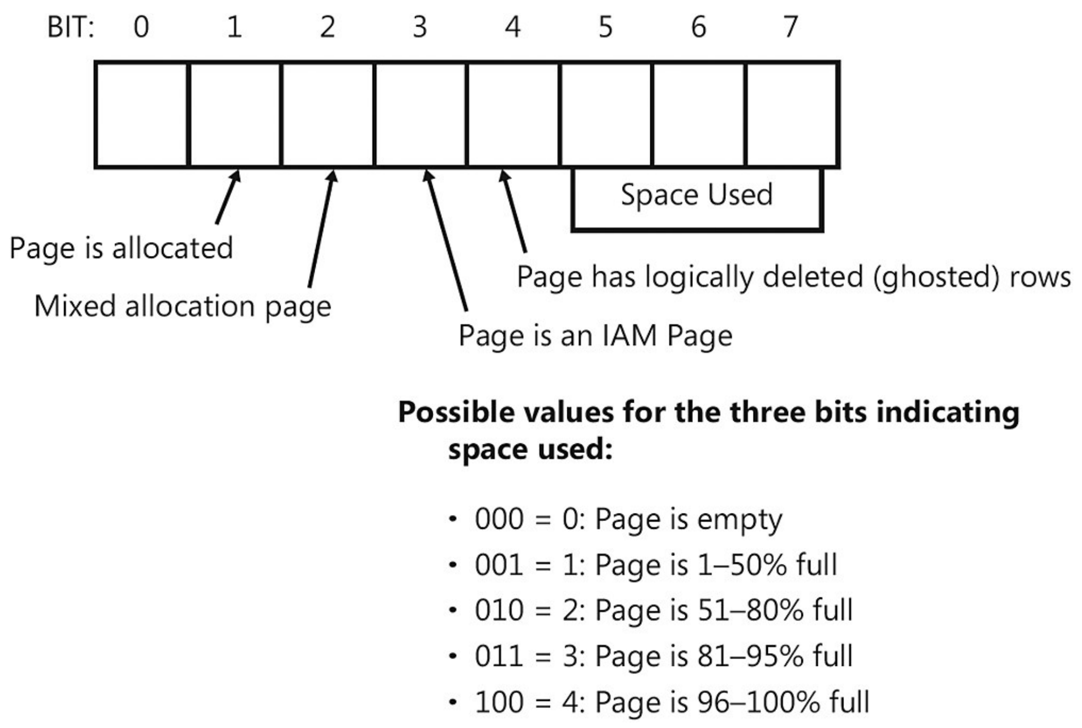

BIT: 0 1 2 3 4 5 6 7
Space Used
Page is allocated
Mixed allocation page
Page is an IAM Page
Page has logically deleted (ghosted) rows
Possible values for the three bits indicating space used:
• 000 = 0: Page is empty
• 001 = 1: Page is 1–50% full
• 010 = 2: Page is 51–80% full
• 011 = 3: Page is 81–95% full
• 100 = 4: Page is 96–100% full

Figure 5-13: Meaning of the bits in a PFS byte

Here is the way the bits are interpreted:

- **Bit 1** This bit indicates whether the page is actually allocated or not. For example, a uniform extent can be allocated to an object, but all of the pages in the extent might not be allocated. To tell which pages within an allocated extent are actually used, SQL Server needs to look at this bit in the appropriate byte in the PFS page.
- **Bit 2** Indicates whether or not the corresponding page is from a mixed extent.
- **Bit 3** Indicates that this page is an IAM page. Remember that IAM pages are not located at known locations in a file.
- **Bit 4** Indicates that this page contains ghost records. As we'll see, SQL Server uses a background cleanup thread to remove ghost records, and these bits on the PFS pages help SQL Server find those pages that need to be cleaned up. (Ghost records only show up in indexes or when using row-level versioning, so they won't be discussed further in this chapter.)
- **Bits 5 through 7** Taken as a three-bit value, the values 0 to 4 indicate the page fullness as follows:
  - 0: The page is empty.
  - 1: The page is 1–50 percent full.
  - 2: The page is 51–80 percent full.
  - 3: The page is 81–95 percent full.
  - 4: The page is 96–100 percent full.

PFS pages are at known locations within each data file. The second page (page 1) of a file is a PFS page, as is every 8,088th page thereafter.

Inserting Rows

When inserting a new row into a table, SQL Server must determine where to put it. When a table has no clustered index—that is, when the table is a heap—a new row is always inserted wherever room is available in the table. I've discussed how IAMs and the PFS pages keep track of which extents in a file already belong to a table and which of the pages in those extents have space available. Even without a clustered index, space management is quite efficient. If no pages with space are available, SQL Server tries to find unallocated pages from existing uniform extents that already belong to the object. If none exists, SQL Server must allocate a whole new extent to the table. Chapter 3 discussed how the GAMs and SGAMs were used to find extents available to be allocated to an object.

Deleting Rows

When you delete rows from a table, you have to consider what happens both to the data pages and the index pages. Remember that the data is actually the leaf level of a clustered index, and deleting rows from a table with a clustered index happens the same way as deleting rows from the leaf level of a nonclustered index. Deleting rows from a heap is managed in a different way, as is deleting from node pages of an index.

Deleting Rows from a Heap

SQL Server 2008 doesn't reorganize space on a page automatically when a row is deleted. As a performance optimization, the compaction doesn't occur until a page needs additional contiguous space for inserting a new row. You can see this in the following example, which deletes a row from the middle of a page and then inspects that page using *DBCC PAGE:*

```
USE testdb;
GO

CREATE TABLE smallrows
(
    a int identity,
    b char(10)
);
GO

INSERT INTO smallrows
    VALUES ('row 1');
INSERT INTO smallrows
    VALUES ('row 2');
INSERT INTO smallrows
    VALUES ('row 3');
INSERT INTO smallrows
    VALUES ('row 4');
INSERT INTO smallrows
    VALUES ('row 5');
GO

DBCC IND (testdb, smallrows, -1);
-- Note the FileID and PageID from the row where PageType = 1
--and use those values with DBCC PAGE (I got FileID 1 and PageID 4536)

DBCC TRACEON(3604);
GO
DBCC PAGE(testdb, 1, 4536,1);
```

Here is the output from *DBCC PAGE*:

```
DATA:

Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61D9C060
00000000:   10001200 01000000 726f7720 31202020 †........row 1
00000010:   20200200 fc†††††††††††††††††††††††††††  ...

Slot 1, Offset 0x75, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61D9C075
00000000:   10001200 02000000 726f7720 32202020 †........row 2
00000010:   20200200 fc†††††††††††††††††††††††††††  ...

Slot 2, Offset 0x8a, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61D9C08A
00000000:   10001200 03000000 726f7720 33202020 †........row 3
00000010:   20200200 fc†††††††††††††††††††††††††††  ...

Slot 3, Offset 0x9f, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
```

```
Memory Dump @0x61D9C09F
00000000:   10001200 04000000 726f7720 34202020 †........row 4
00000010:   20200200 fc††††††††††††††††††††††††††  ...

Slot 4, Offset 0xb4, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61D9C0B4
00000000:   10001200 05000000 726f7720 35202020 †........row 5
00000010:   20200200 fc††††††††††††††††††††††††††  ...

OFFSET TABLE:
Row - Offset
4 (0x4) - 180 (0xb4)
3 (0x3) - 159 (0x9f)
2 (0x2) - 138 (0x8a)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

Now we'll delete the middle row (`WHERE a = 3`) and look at the page again:

```
DELETE FROM smallrows
WHERE a = 3;
GO

DBCC PAGE(testdb, 1, 4536,1);
GO
```

Here is the output from the second execution of *DBCC PAGE:*

```
DATA:
Slot 0, Offset 0x60, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61B6C060
00000000:   10001200 01000000 726f7720 31202020 †........row 1
00000010:   20200200 fc††††††††††††††††††††††††††  ...

Slot 1, Offset 0x75, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61B6C075
00000000:   10001200 02000000 726f7720 32202020 †........row 2
00000010:   20200200 fc††††††††††††††††††††††††††  ...

Slot 3, Offset 0x9f, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61B6C09F
00000000:   10001200 04000000 726f7720 34202020 †........row 4
00000010:   20200200 fc††††††††††††††††††††††††††  ...

Slot 4, Offset 0xb4, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61B6C0B4
00000000:   10001200 05000000 726f7720 35202020 †........row 5
00000010:   20200200 fc††††††††††††††††††††††††††  ...

OFFSET TABLE:
Row - Offset
4 (0x4) - 180 (0xb4)
3 (0x3) - 159 (0x9f)
2 (0x2) - 0 (0x0)
1 (0x1) - 117 (0x75)
0 (0x0) - 96 (0x60)
```

Note that in the heap, the row offset array at the bottom of the page shows that the third row (at slot 2) is now at offset 0 (which means there really is no row using slot 2), and the row using slot 3 is at its same offset as before the delete. No data on the page is moved when the *DELETE* occurs. The row doesn't show up in the page when you use *printopt* 1 or 3 for *DBCC PAGE*. However, if you dump the page with *printopt* 2, you still see the bytes for 'row 3'. They are not physically removed from the page, but the 0 in the row offset array indicates that the space is not used now and can be used by a new row.

In addition to space on pages not being reclaimed, empty pages in heaps frequently cannot be reclaimed. Even if you delete all the rows from a heap, SQL Server does not mark the empty pages as unallocated, so the space is not available for other objects to use. The dynamic management view (DMV) *sys.dm_db_partition_stats* still shows the space as belonging to the heap table. One way to avoid this problem is to request a table lock when the delete is being performed, and we'll look at lock hints in Chapter 10. If this problem has already occurred, and you are showing more space belonging to a table than it really has, you can build a clustered index on the table to reorganize the space and then drop the index.

Reclaiming Pages

When the last row is deleted from a data page, the entire page is deallocated. The exception is if the table is a heap, as I discussed previously. (If the page is the only one remaining in the table, it isn't deallocated. A table always contains at least one page, even if it's empty.) Deallocation of a data page results in the deletion of the row in the index page that pointed to the deallocated data page. Index pages are deallocated if an index row is deleted (which, again, might occur as part of a delete/insert update strategy), leaving only one entry in the index page. That entry is moved to its neighboring page, and then the empty page is deallocated.

The discussion so far has focused on the page manipulation necessary for deleting a single row. If multiple rows are deleted in a single *DELETE* operation, you must be aware of some other issues.

Updating Rows

SQL Server can update rows in several different ways, automatically and invisibly choosing the fastest update strategy for the specific operation. In determining the strategy, SQL Server evaluates the number of rows affected, how the rows will be accessed (via a scan or an index retrieval, and via which index), and whether changes to the index keys will occur. Updates can happen either in place, by just changing one column's value to a new value in the original row, or as a delete followed by an insert. In addition, updates can be managed by the query processor or by the storage engine. In this section, we'll examine only whether the update happens in place or whether SQL Server treats it as two separate operations: delete the old row and insert a new row.

Moving Rows

What happens if a row has to move to a new location in the table? In SQL Server 2008, this can happen for a couple of different reasons. In Chapter 6, we'll look at the structure of indexes and see that the value in a table's clustered index column (or columns) determines the location of the row. So, if the value of the clustered key is changed, the row most likely has to move within the table.

If it will still have the same row locator (in other words, the clustering key for the row stays the same), no nonclustered indexes have to be modified. If a table has no clustered index (in other words, if it's a heap), a row may move because it no longer fits on the original page. This can happen whenever a row with variable-length columns is updated to a new, larger size so that it no longer fits in the original location. As you'll see when we cover index structures in Chapter 6, every nonclustered index on a heap contains pointers to the data rows that are the actual physical location of the row, including the file number, page number, and row number. So that the nonclustered indexes do not all have to be updated just because a row moves to a different physical location, SQL Server leaves a forwarding pointer in the original location when a row has to move.

Let's look at an example to see these forwarding pointers. I'll create a table that's much like the one I created for doing *DELETE* operations, but this table has a third column of variable length. After I populate the table with five rows, which fills the page, I'll update one of the rows to make its third column much longer. The row no longer fits on the original page and has to move. I can use *DBCC IND* to get the page numbers used by the table as follows:

```
USE testdb;
GO
DROP TABLE bigrows;
GO
CREATE TABLE bigrows
(   a int IDENTITY ,
    b varchar(1600),
    c varchar(1600));
GO
INSERT INTO bigrows
    VALUES (REPLICATE('a', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('b', 1600), '');
INSERT INTO bigrows
```

```
    VALUES (REPLICATE('c', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('d', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('e', 1600), '');
GO
UPDATE bigrows
SET c = REPLICATE('x', 1600)
WHERE a = 3;
GO

DBCC IND (testdb, bigrows, -1);

DBCC IND (testdb, bigrows, -1);
-- Note the FileID and PageID from the rows where PageType = 1
--and use those values with DBCC PAGE (I got FileID 1 and
--PageID values of 2252 and 4586.

RESULTS:
PageFID PagePID
------- -----------
1       2252
1       4586

DBCC TRACEON(3604);
GO
DBCC PAGE(testdb, 1, 2252, 1);
GO
```

I won't show you the entire output from the *DBCC PAGE* command, but I'll show you what appears in the slot where the row with a = 3 formerly appeared:

```
Slot 2, Offset 0x1feb, Length 9, DumpStyle BYTE
Record Type = FORWARDING_STUB        Record Attributes =
Memory Dump @0x61ADDFEB
00000000:   04ea1100 00010000 00††††††††††††††††.........
```

The value of 4 in the first byte means that this is just a forwarding stub. The 0011ea in the next 3 bytes is the page number to which the row has been moved. Because this is a hexadecimal value, we need to convert it to 4586 decimal. The next group of 4 bytes tells us that the page is at slot 0, file 1. If you then use *DBCC PAGE* to look at that page, page 4,586, you can see what the forwarded record looks like, and you can see that the Record Type indicates FORWARDED_RECORD.

Managing Forward Pointers

Forward pointers allow you to modify data in a heap without worrying about having to make drastic changes to the nonclustered indexes. If a row that has been forwarded must move again, the original forwarding pointer is updated to point to the new location. You'll never end up with a forwarding pointer pointing to another forwarding pointer. In addition, if the forwarded row shrinks enough to fit in its original place, the record might move back to its original place (if there is still room on that page), and the forward pointer would be eliminated.

A future version of SQL Server might include some mechanism for performing a physical reorganization of the data in a heap, which would get rid of forward pointers. Note that forward pointers exist only in heaps, and that the *ALTER TABLE* option to reorganize a table won't do anything to heaps. You can defragment a nonclustered index on a heap but not the table itself. Currently, when a forward pointer is created, it stays there forever—with only a few exceptions. The first exception is the case I already mentioned, in which a row shrinks and returns to its original location. The second exception is when the entire database shrinks. The bookmarks are actually reassigned when a file is shrunk. The shrink process never generates forwarding pointers. For pages that were removed because of the shrink process, any forwarded rows or stubs they contain are effectively "unforwarded." Other cases in which the forwarding pointers are removed are the obvious ones: if the forwarded row is deleted, or if a clustered index is built on the table so that it is no longer a heap.

More Info To get a count of forward records in a table, you can look at the output from the *sys.dm_db_index_physical_stats* function, which will be discussed in Chapter 6.

Updating in Place

In SQL Server 2008, updating a row in place is the rule rather than the exception. This means that the row stays in exactly the same location on the same page and only the bytes affected are changed. In addition, the log contains a single record for each such updated row unless the table has an update trigger on it or is marked for replication. In these cases, the update still happens in place, but the log contains a delete record followed by an insert record.

In cases where a row can't be updated in place, the cost of a not-in-place update is minimal because of the way the nonclustered indexes are stored and because of the use of forwarding pointers, as described previously. In fact, you can have an update not-in-place for which the row stays on the original page. Updates happen in place if a heap is being updated (and no forwarding pointer is required), or if a table with a clustered index is updated without any change to the clustering keys. You can also get an update in place if the clustering key changes but the row does not need to move at all. For example, if you have a clustered index on a last-name column containing consecutive key values of *Able*, *Becker*, and *Charlie*, you might want to update *Becker* to *Baker*. Because the row stays in the same location even after the clustered index key changes, SQL Server performs this as an update in place. On the other hand, if you update *Able* to *Buchner*, the update cannot occur in place but the new row might stay on the same page.

Updating Not in Place

If your update can't happen in place because you're updating clustering keys, the update occurs as a delete followed by an insert. In some cases, you'll get a hybrid update: some of the rows are updated in place and some aren't. If you're updating index keys, SQL Server builds a list of all the rows that need to change as both a *DELETE* and an *INSERT* operation. This list is stored in memory, if it's small enough, and is written to *tempdb* if necessary. This list is then sorted by key value and operator (*DELETE* or *INSERT*). If the index whose keys are changing isn't unique, the *DELETE* and *INSERT* steps are then applied to the table. If the index is unique, an additional step is carried out to collapse *DELETE* and *INSERT* operations on the same key into a single update operation.

More Info The Query Optimizer determines whether this special *UPDATE* method is appropriate, and this internal optimization, called *Split/Sort/Collapse,* is described in detail in Chapter 8, "The Query Optimizer."

Summary

Tables are at the heart of relational databases in general and SQL Server in particular. In this chapter, we looked at the internal storage issues of various data types, in particular comparing fixed- and variable-length data types. We saw that SQL Server 2008 provides multiple options for storing variable-length data, including data that is too long to fit on a single data page, and you saw that it's simplistic to think that using variable-length data types is either always good or always bad. SQL Server provides user-defined data types for support of domains, and it provides the *IDENTITY* property to make a column produce auto-sequenced numeric values. You also saw how data is physically stored in data pages, and we queried some of the metadata views that provide information from the underlying (and inaccessible) system tables. SQL Server also provides constraints, which offer a powerful way to ensure your data's logical integrity.