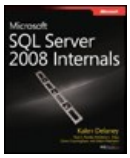

Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.

Chapter 7: Special Storage

Kalen Delaney

In Chapter 5, “Tables,” and Chapter 6, “Indexes: Internals and Management,” we discussed the storage of “regular rows” for both data and index information. I told you in Chapter 5 that regular rows are in a format called *FixedVar*. SQL Server provides ways of storing data in another format called *Column Descriptor (CD)*. It also can store special values in either the FixedVar or CD format that don’t fit on the regular 8-KB pages. In this chapter, I’ll describe data that exceeds the normal row size limitations and is stored as either row-overflow or Large Object (LOB) data. I’ll tell you about two additional methods for storing data on the actual data pages, introduced in Microsoft SQL Server 2008, one that uses a new type of complex column with a regular data row (sparse columns), and one that uses the new CD format (compressed data). I’ll also discuss filestream data, a new feature in SQL Server 2008, which allows you to access data from operating system files as if it were part of your relational tables.

Finally, I will discuss the ability of SQL Server to separate data into partitions. Although this doesn’t change the format of data in the rows on or the pages, it does change the metadata that keeps track of what space is allocated to which objects.

Large Object Storage

SQL Server 2008 has two special formats for storing data that doesn’t fit on the regular 8-KB data page. These formats allow you to have rows that exceed the maximum row size of 8,060 bytes. As discussed previously, this maximum row size value includes several bytes of overhead stored with the row on the physical pages, so the total size of all the table’s defined columns must be slightly less than this amount. In fact, the error message that you get if you try to create a table with more bytes than the allowable maximum is very specific. If you execute the following *CREATE TABLE* statement with column definitions that add up to exactly 8,060 bytes, you’ll get the error message shown here:

```
USE test;
CREATE TABLE dbo.bigrows_fixed
(   a char(3000),
    b char(3000),
    c char(2000),
    d char(60) ) ;

Msg 1701, Level 16, State 1, Line 1
Creating or altering table 'bigrows' failed because the minimum row size would be 8067,
including 7 bytes of internal overhead. This exceeds the maximum allowable table row size of
8060 bytes.
```

In this message, you can see the number of overhead bytes (7) that SQL Server wants to store with the row itself. There is also an additional 2 bytes for the row-offset bytes at the end of the page, but those bytes are not included in this total here.

Restricted-Length Large Object Data (Row-Overflow Data)

One way to exceed this size limit of 8,060 bytes is to use variable-length columns because for variable-length data, SQL Server 2005 and SQL Server 2008 can store the columns in special row-overflow pages, so long as all the fixed-length columns fit into the regular in-row size limit. So let’s take a look at a table with all variable-length columns. Note that although my example uses columns that are all *varchar*, columns of other data types can also be stored on row-overflow data pages. These other data types include *varbinary*, *nvarchar*, and *sqlvariant* columns, as well as columns that use the CLR user-defined data types. The following code creates a table with rows that have a maximum defined length that is much longer than 8,060 bytes:

```
USE test;
CREATE TABLE dbo.bigrows
  (a varchar(3000),
   b varchar(3000),
   c varchar(3000),
   d varchar(3000) );
```

In fact, if you run this *CREATE TABLE* statement in SQL Server 7.0, you get an error, and the table is not created at all. In SQL Server 2000, the table is created but you get a warning that inserts or updates might fail if the row size exceeds the maximum.

With SQL Server 2005 and SQL Server 2008, not only can the preceding *dbo.bigrows* table be created, but you can insert

a row with column sizes that add up to more than 8,060 bytes with a simple *INSERT,* as shown here:

```
INSERT INTO dbo.bigrows
     SELECT REPLICATE('e', 2100), REPLICATE('f', 2100),
      REPLICATE('g', 2100),  REPLICATE('h', 2100);
```

To determine whether SQL Server is storing any data in row-overflow data pages for a particular table, you can run the following allocation query from Chapter 5:

```
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.bigrows');
```

This query should return output similar to that shown here:

```
name    partition_id      pnum rows au_id             page_type_desc      pages
------------------- ---- ---- ---------------- --------------------
bigrows 72057594039238656 1    1    72057594043957248 IN_ROW_DATA         2
bigrows 72057594039238656 1    1    72057594044022784 ROW_OVERFLOW_DATA   2
```

You can see that there are two pages for the one row of regular in-row data and two pages for the one row of row-overflow data. Alternatively, you can use the command *DBCC IND(test, bigrows, -1)* and see the four pages individually. Looking at only four pages with *DBCC IND* is not too awkward, but once a table starts growing and contains hundreds or thousands of pages (or even more), the output of *DBCC IND* can be very difficult to work with, as *DBCC IND* returns one row per page. Chapter 6 provided you a script to build a table called *sp_tablepages,* which can be used to capture your *DBCC IND* output into a table, and then it is much easier to find just the rows you're interested in, to count the rows, to group them by page type, or to display just a subset of the columns. To populate this table with the information about the *bigrows* table, I can run the following *INSERT* statement:

```
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (test, bigrows, -1)');
```

Once the table is populated, you can select only the columns of interest, as follows:

```
SELECT PageFID, PagePID, ObjectID, PartitionID, IAM_chain_type, PageType
FROM sp_tablepages;
```

You should see the four rows shown, one for each page, as follows:

```
PageFID PagePID   ObjectID   PartitionID         IAM_chain_type     PageType
------- --------  ---------  -----------------   ---------------    --------
1       2252      85575343   72057594039238656   Row-overflow data  3
1       2251      85575343   72057594039238656   Row-overflow data  10
1       2254      85575343   72057594039238656   In-row data        1
1       2253      85575343   72057594039238656   In-row data        10
```

Two pages are for the row-overflow data, and two are for the in-row data. As you saw in Chapter 6, the *PageType* values have the following meanings:

- *PageType* = 1, Data page.
- *PageType* = 2, Index page.
- *PageType* = 3, LOB or row-overflow page, TEXT_MIXED.
- *PageType* = 4, LOB or row-overflow page, TEXT_DATA.
- *PageType* = 10, IAM page.

I'll tell you more about the different types of LOB pages in the section entitled " Unrestricted-Length Large Object Data," later in this chapter.

We can see that there is one data page and one IAM page for the in-row data, and one data page and one IAM page for the row-overflow data. With the results from *DBCC IND*, we could then look at the page contents with *DBCC PAGE.* On the data page for the in-row data, we would see three of the four *varchar* column values, and the fourth column would be

stored on the data page for the row-overflow data. If you run *DBCC PAGE* for the data page storing the in-row data (page 1:2254 in my example), you'll notice that it isn't necessarily the fourth column in the column order that is stored off the row. I won't show you the entire contents of the rows because the single row fills almost the entire page. When I look at the in-row data page using *DBCC PAGE*, I see the column with *e*, the column with *g*, and the column with *h*, and it is the column with *f* that has moved to the new row. In the place of that column, we can see the bytes shown here:

```
65020000 00010000 00290000 00340800 00cc0800 00010000 0067
```

I have included the last byte with *e* (ASCII code hexadecimal 65) and the first byte with *g* (ASCII code hexadecimal 67), and in between, there are 24 other bytes. Bytes 16 through 23 (the 17th through the 24th bytes) of those 24 bytes are treated as an 8-byte numeric value: cc08000001000000. We need to reverse the byte order and break it into a 2-byte hex value for the slot number, a 2-byte hex value for the file number, and a 4-byte hex value for the page number. So the file number is 0x0000 for slot 0 because this overflowing column is the first (and only) data on the row-overflow page. We have 0x0001, or 1, for the file number, and 0x000008cc, or 2252, for the page number. These are the same file and page numbers that we saw using *DBCC IND*.

The first 16 bytes in the row have the meanings indicated in Table 7-1.

Table 7-1: The First 16 Bytes of a Row-Overflow Pointer

Bytes	Hex Value	Decimal Value	Meaning
0	0x02	2	Type of special field: 1 = LOB 2 = overflow
1–2	0x0000	0	Level in the B-tree (always 0 for overflow)
3	0x00	0	Unused
4–7	0x00000001	1	Sequence: a value used by optimistic concurrency control for cursors that increases every time a LOB or overflow column is updated
8–11	0x00000029	2686976	Timestamp: a random value used by *DBCC CHECKTABLE* that remains unchanged during the lifetime of each LOB or overflow column
12–15	0x00000834	2100	Length

SQL Server stores variable-length columns on row-overflow pages only under certain conditions. The determining factor is the length of the row itself. It doesn't matter how full the regular page is into which SQL Server is trying to insert the new row. SQL Server constructs the row normally, and stores some of its columns on overflow pages only if the row itself needs more than 8,060 bytes.

Each column in the table is either completely on the row or completely off the row. This means that a 4,000-byte variable-length column cannot have half its bytes on the regular data page and half on a row-overflow page. If a row is less than 8,060 bytes and there is no room on the page where SQL Server is trying to insert it, normal page splitting algorithms (described in Chapter 6) are applied.

One row can span many row-overflow pages if it contains many large variable-length columns. For example, you can create the table *dbo.hugerows* and insert a single row into it as follows:

```
CREATE TABLE dbo.hugerows
  (a varchar(3000),
   b varchar(8000),
   c varchar(8000),
   d varchar(8000));
INSERT INTO dbo.hugerows
     SELECT REPLICATE('a', 3000), REPLICATE('b', 8000),
         REPLICATE('c', 8000),  REPLICATE('d', 8000);
```

Now if I run the allocation query shown previously, substituting *hugerows* for *bigrows*, I get the results shown here:

```
name     partition_id       pnum rows au_id               page_type_desc     pages
-------- -----------------  ---- ---- -----------------  -----------------  -----
hugerows 72057594039304192  1    1    72057594044088320  IN_ROW_DATA        2
hugerows 72057594039304192  1    1    72057594044153856  ROW_OVERFLOW_DATA  4
```

There are four pages for the row-overflow information, one for the row-overflow IAM page, and three for the columns that

didn't fit in the regular row. The number of large variable-length columns that a table can have is not unlimited, although it is quite large. There is a limit of 1,024 columns in any table. (The 1,024-column limit can be exceeded when you are using sparse columns, which is discussed later in this chapter.) But another limit is reached before that. When a column has to be moved off a regular page onto a row-overflow page, SQL Server keeps a pointer to the row-overflow information as part of the original row, which we saw in the DBCC output before is 24 bytes, and the row still needs 2 bytes in the column-offset array for each variable-length column, whether or not the variable-length column is stored in the row. So it turns out that 308 is the maximum number of overflowing columns we can have, and such a row needs 8,008 bytes just for the 26 overhead bytes for each overflowing column in the row.

Note Just because SQL Server can store lots of large columns on row-overflow pages doesn't mean it's always a good idea to do so. This capability does allow you more flexibility in the organization of your tables, but you might pay a heavy performance price if many additional pages need to be accessed for every row of data. Row-overflow pages are intended to be a solution in the situation where most rows fit completely on your data pages and you have row-overflow data only occasionally. Using row-overflow pages, SQL Server can handle the extra data effectively, without requiring a redesign of your table.

In some cases, if a large variable-length column shrinks, it can be moved back to the regular row. However, for efficiency reasons, if the decrease is just a few bytes, SQL Server does not bother checking. Only when a column stored in a row-overflow page is reduced by more than 1,000 bytes does SQL Server even consider checking to see whether the column can now fit on the regular data page. You can observe this behavior if you previously created the *dbo.bigrows* table for the previous example and inserted only the one row with 2,100 characters in each column.

The following update reduces the size of the first column by 500 bytes, reducing the row size to 7,900 bytes, which should all fit on one data page:

```
UPDATE bigrows
SET a = replicate('a', 1600);
```

However, if you run the allocation query again, you'll still see two row-overflow pages: one for the row-overflow data and one for the IAM page. Now reduce the size of the first column by more than 1,000 bytes and run the allocation query once more:

```
UPDATE bigrows
SET a = 'aaaaa';
```

You should see only three pages for the table now, because there is no longer a row-overflow data page. The IAM page for the row-overflow data pages has not been removed, but you no longer have a data page for row-overflow data.

Keep in mind that row-overflow data storage applies only to columns of variable-length data, which are defined as no longer than the normal variable-length maximum of 8,000 bytes per column. In addition, to store a variable-length column on a row-overflow page, you must meet the following conditions:

- All the fixed-length columns, including overhead bytes, must add up to no more than 8,060 bytes. (The pointer to each row-overflow column adds 24 bytes of overhead to the row.)
- The actual length of the variable-length column must be more than 24 bytes.
- The column must not be part of the clustered index key.

If you have single columns that might need to store more than 8,000 bytes, you should use either LOB (*text*, *image*, or *ntext*) columns or use the *MAX* data types.

Unrestricted-Length Large Object Data

If a table contains the older LOB data types (*text*, *ntext*, or *image* types), by default the actual data is not stored on the regular data pages. Like row-overflow data, LOB data is stored in its own set of pages, and the allocation query shows you pages for LOB data as well as pages for regular in-row data and row-overflow data. For LOB columns, SQL Server stores a 16-byte pointer in the data row that indicates where the actual data can be found. Although the default behavior is to store all the LOB data off the data row, SQL Server allows you to change the storage mechanism by setting a table option to allow LOB data to be stored in the data row itself if it is small enough. Note that there is no database or server setting to control storing small LOB columns on the data pages; it is managed as a table option.

By default no LOB data is stored in the data row. Instead, the data row contains only a 16-byte pointer to a page (or the first of a set of pages) where the data can be found. These pages are 8 KB in size, like any other page in SQL Server, and

individual *text*, *ntext*, and *image* pages aren't limited to storing data for only one occurrence of a *text*, *ntext*, or *image* column. A *text*, *ntext*, or *image* page can hold data from multiple columns and from multiple rows; the page can even have a mix of *text*, *ntext*, and *image* data. However, one *text* or *image* page can hold only *text* or *image* data from a single table. (Even more specifically, one *text* or *image* page can hold only *tex*t or *image* data from a single partition of a table, which will become clear when I discuss partitioning metadata at the end of this chapter.)

The collection of 8-KB pages that make up a LOB column aren't necessarily located next to each other. The pages are logically organized in a B-tree structure, so operations starting in the middle of the LOB string are very efficient. The structure of the B-tree varies slightly depending on whether the amount of data is less than or more than 32 KB. (See Figure 7-1 for the general structure.) B-trees were discussed in detail when describing indexes in Chapter 6.

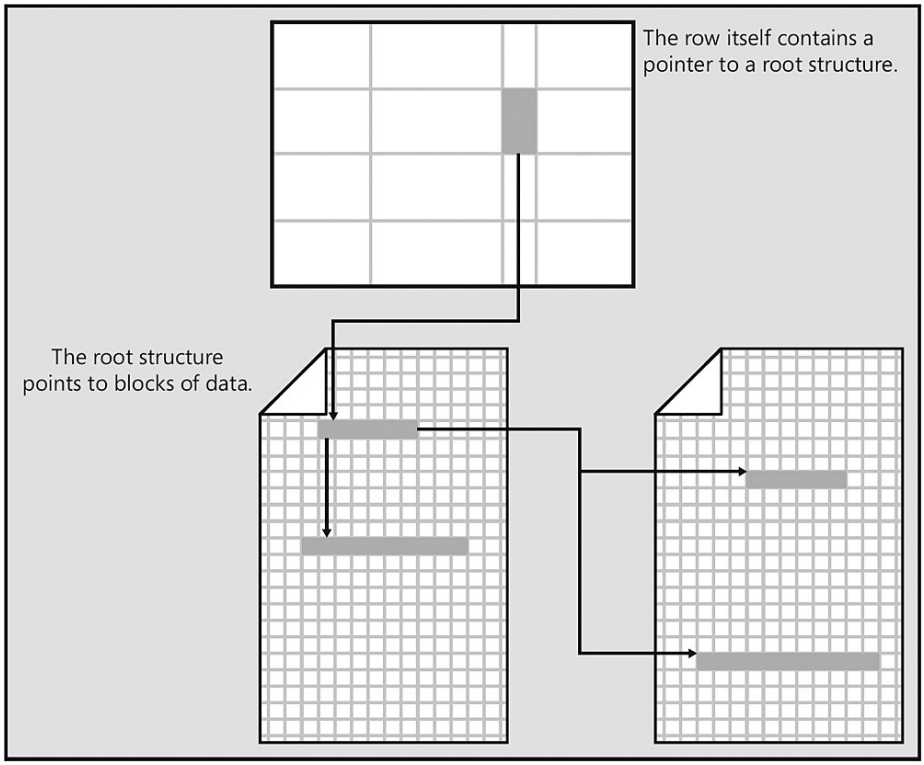

Figure 7-1: A text column pointing to a B-tree that contains the blocks of data

If the amount of LOB data is less than 32 KB, the text pointer in the data row points to an 84-byte text root structure. This forms the root node of the B-tree structure. The root node points to the blocks of *text, ntext,* or *image* data. Although the data for LOB columns is arranged logically in a B-tree, both the root node and the individual blocks of data are spread physically throughout LOB pages for the table. They're placed wherever space is available. The size of each block of data is determined by the size written by an application. Small blocks of data are combined to fill a page. If the amount of data is less than 64 bytes, it's all stored in the root structure.

If the amount of data for one occurrence of a LOB column exceeds 32 KB, SQL Server starts building intermediate nodes between the data blocks and the root node. The root structure and the data blocks are interleaved throughout the *text* and *image* pages. The intermediate nodes, however, are stored in pages that aren't shared between occurrences of *text* or *image* columns. Each page storing intermediate nodes contains only intermediate nodes for one *text* or *image* column in one data row.

SQL Server can store the LOB root and the actual LOB data on two different types of pages. One of these, referred to as TEXT_MIXED, allows LOB data from multiple rows to share the same pages. However, once your text data gets larger than about 40 KB, SQL Server starts devoting whole pages to a single LOB value. These pages are referred to as TEXT_DATA pages.

You can see this behavior by creating a table with a *text* column, inserting a value of less than 40 KB and then a value of greater than 40 KB, and examining the output of *DBCC IND*. The following script uses the *sp_tablepages* table created previously:

```
IF EXISTS (SELECT * FROM sys.tables
```

```
                        WHERE name = 'textdata')
    DROP TABLE textdata;
GO
CREATE TABLE textdata
 (bigcol text);
GO
INSERT INTO textdata
   SELECT REPLICATE(convert(varchar(MAX), 'a'), 38000);
GO
TRUNCATE TABLE sp_tablepages;
GO
INSERT INTO sp_tablepages
  EXEC('DBCC IND(test, textdata, -1)');
GO
SELECT PageFID, PagePID, ObjectID, IAM_chain_type, PageType
FROM sp_tablepages;
GO
INSERT INTO textdata
   SELECT REPLICATE(convert(varchar(MAX), 'a'), 41000);
GO
TRUNCATE TABLE sp_tablepages;
GO
INSERT INTO sp_tablepages
  EXEC('DBCC IND(test, textdata, -1)');
GO
SELECT PageFID, PagePID, ObjectID, IAM_chain_type, PageType
FROM sp_tablepages;
```

The first time that you select from *sp_tablepages*, you should have *PageType* values of 1, 3, and 10. The second time, once we have inserted data greater than 40 KB in size, we should also see *PageType* values of 4. *PageType* 3 indicates a TEXT_MIXED page, and *PageType* 4 indicates a TEXT_DATA page.

LOB Data Stored in the Data Row

If you store all your LOB data type values outside your regular data pages, SQL Server needs to perform additional page reads every time you access that data, just as it does for row-overflow pages. In some cases, you might notice a performance improvement by allowing some of the LOB data to be stored in the data row. You can enable a table option called *text in row* for a particular table by setting the option to 'ON' (including the quote marks) or by specifying a maximum number of bytes to be stored in the data row. The following command enables up to 500 bytes of LOB data to be stored with the regular row data in a table called *employee*:

```
EXEC sp_tableoption employee, 'text in row', 500;
```

Note that the value is in bytes, not characters. For *ntext* data, each character needs 2 bytes so that any *ntext* column is stored in the data row if it is less than or equal to 250 characters. Once you enable the *text in row* option, you never get just the 16-byte pointer for the LOB data in the row, as is the case when the option is not 'ON'. If the data in the LOB field is more than the specified maximum, the row holds the root structure containing pointers to the separate chunks of LOB data. The minimum size of a root structure is 24 bytes, and the possible range of values that *text in row* can be set to is 24 to 7,000 bytes. (If you specify the option 'ON' instead of a specific number, SQL Server assumes the default value of 256 bytes.)

To disable the *text in row* option, you can set the value to either 'OFF' or 0. To determine whether a table has the *text in row* property enabled, you can inspect the *sys.tables* catalog view as follows:

```
SELECT name, text_in_row_limit
FROM sys.tables
WHERE name = 'employee';
```

This *text_in_row_limit* value indicates the maximum number of bytes allowed for storing LOB data in a data row. If a 0 is returned, the *text in row* option is disabled.

Let's create a table very similar to the one we created to look at row structures, but we'll change the *varchar(250)* column to the *text* data type. We'll use almost the same *INSERT* statement to insert one row into the table:

```
CREATE TABLE HasText
(
Col1 char(3)       NOT NULL,
```

```
Col2 varchar(5)    NOT NULL,
Col3 text          NOT NULL,
Col4 varchar(20)   NOT NULL
);

INSERT HasText VALUES
    ('AAA', 'BBB', REPLICATE('X', 250), 'CCC');
```

Now let's find the basic information for this table using the allocation query and also look at the *DBCC IND* values for this table:

```
SELECT convert(char(7), object_name(object_id))  AS name,
     partition_id, partition_number AS pnum,  rows,
     allocation_unit_id AS au_id, convert(char(17), 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.HasText');

DBCC IND (test, HasText, -1);

name    partition_id      pnum rows  au_id             page_type_desc  pages
------- ----------------- ---- ----- ----------------- --------------- -----
HasText 72057594039435264 1    1     72057594044350464 IN_ROW_DATA     2
HasText 72057594039435264 1    1     72057594044416000 LOB_DATA        2

PageFID  PagePID  ObjectID   PartitionID         IAM_chain_type   PageType
-------  -------  --------   -----------------   --------------   --------
1        2197     133575514  72057594039435264   LOB data         3
1        2198     133575514  72057594039435264   LOB data         10
1        2199     133575514  72057594039435264   In-row data      1
1        2200     133575514  72057594039435264   In-row data      10
```

You can see two LOB pages (the LOB data page and the LOB IAM page) and two pages for the in-row data (again, the data page and the IAM page). The data page for the in-row data is 2199, and the LOB data is on page 2197. Figure 7-2 shows the output from running *DBCC PAGE* on page 2199. The row structure is very similar to the row structure shown in Chapter 5, in Figure 5-10, except for the text field itself. Bytes 21 to 36 are the 16-byte text pointer, and you can see the value 9508 starting at offset 29. When we reverse the bytes, it becomes 0x0895, or 2197 decimal, which is the page containing the text data, as we saw in the *DBCC IND* output.

```
DATA:


Slot 0, Offset 0x60, Length 40, DumpStyle BYTE

Record Type = PRIMARY_RECORD   Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Record Size = 40
Memory Dump @0x625BC060

00000000:  30000700 41414104 00600300 15002580 †0...AAA..`....%.
00000010:  28004242 420000e1 07000000 00950800 †(.BBB..á.....?..
00000020:  00010001 00434343 ††††††††††††††††††.....CCC
```

Figure 7-2: A row containing a text pointer

Now let's enable text data in the row, for up to 500 bytes:

```
EXEC sp_tableoption HasText, 'text in row', 500;
```

Enabling this option does not force the text data to be moved into the row. We have to update the text value to actually force the data movement:

```
UPDATE HasText
SET col3 =  REPLICATE('Z', 250);
```

If you run *DBCC PAGE* on the original data page, you see that the text column of 250 *z*'s is now in the data row and that

the row is practically identical in structure to the row containing *varchar* data that we saw in Figure 5-10.

Although enabling *text in row* does not move the data immediately, disabling the option does. If you turn off *text in row*, the LOB data moves immediately back onto its own pages, so you must make sure you don't turn this off for a large table during heavy operations.

A final issue when working with LOB data and the *text in row* option is dealing with the situation where *text in row* is enabled but the LOB is longer than the maximum configured length for some rows. If you change the maximum length for *text in row* to 50 for the *HasText* table we've been working with, this also forces the LOB data for all rows with more than 50 bytes of LOB data to be moved off the page immediately, just as when you disable the option completely:

```
EXEC sp_tableoption HasText, 'text in row', 50;
```

However, setting the limit to a smaller value is different than disabling the option in two ways. First, some of the rows might still have LOB data that is under the limit, and for those rows, the LOB data is stored completely in the data row. Second, if the LOB data doesn't fit, the information stored in the data row itself is not simply the 16-byte pointer, as it would be if *text in row* were turned off. Instead, for LOB data that doesn't fit in the defined size, the row contains a root structure for a B-tree that points to chunks of the LOB data. So long as the *text in row* option is not 'OFF' (or 0), SQL Server never stores the simple 16-byte LOB pointer in the row. It stores either the LOB data itself, if it fits, or the root structure for the LOB data B-tree.

A root structure is at least 24 bytes long (which is why 24 is the minimum size for the *text in row* limit) and the meaning of the bytes is similar to the meaning of the 24 bytes in the row-overflow pointer. The main difference is that no length is stored in bytes 12–15. Instead, bytes 12–23 constitute a link to a chunk of LOB data on a separate page. If multiple LOB chucks are accessed via the root, multiple sets of 12 bytes can be here, each pointing to LOB data on a separate page.

As indicated previously, when you first enable *text in row*, no data movement occurs until the text data is actually updated. The same is true if the limit is increased—that is, even if the new limit is large enough to accommodate the LOB data that was stored outside the row, the LOB data is not moved onto the row automatically. You must update the actual LOB data first.

Another point to keep in mind is that even if the amount of LOB data is less than the limit, the data is not necessarily stored in the row. You're still limited to a maximum row size of 8,060 bytes for a single row on a data page, so the amount of LOB data that can be stored in the actual data row might be reduced if the amount of non-LOB data is large. In addition, if a variable-length column needs to grow, it might push LOB data off the page so as not to exceed the 8,060-byte limit. Growth of variable-length columns always has priority over storing LOB data in the row. If no variable-length *char* fields need to grow during an update operation, SQL Server checks for growth of in-row LOB data, in column offset order. If one LOB needs to grow, others might be pushed off the row.

Finally, you should be aware that SQL Server logs all movement of LOB data, which means that reducing the limit of or turning OFF the *text in row* option can be a very time-consuming operation for a large table.

Although large data columns using the LOB data types can be stored and managed very efficiently, using them in your tables can be problematic. Data stored as *text*, *ntext*, or *image* cannot always be manipulated using the normal data manipulation commands, and in many cases, you need to resort to using the operations *readtext*, *writetext*, and *updatetext*, which require dealing with byte offsets and data-length values. Prior to SQL Server 2005, you had to decide whether to limit your columns to a maximum of 8,000 bytes or deal with your large data columns using different operators than you used for your shorter columns. SQL Server 2005 and SQL Server 2008 provide a solution that gives you the best of both worlds, as we'll see in the next section.

Storage of MAX-Length Data

SQL Server 2005 and SQL Server 2008 give us the option of defining a variable-length field using the MAX specifier. Although this functionality is frequently described by referring only to *varchar(MAX),* the MAX specifier can also be used with *nvarchar* and *varbinary*. You can indicate the MAX specifier instead of an actual size when you define a column, variable, or parameter using one of these types. By using the MAX specifier, you leave it up to SQL Server to determine whether to store the value as a regular *varchar*, *nvarchar*, or *varbinary* value or as a LOB. In general, if the actual length is 8,000 bytes or less, the value is treated as if it were one of the regular variable-length data types, including possibly overflowing onto row-overflow pages. However, if the *varchar(MAX)* column does need to spill off the page, the extra pages required are considered LOB pages and show the *IAM_chain_type* LOB when examined using *DBCC IND*. If the actual length is greater than 8,000 bytes, SQL Server stores and treats the value exactly as if it were *text, ntext*, or *image*.

Because variable-length columns with the MAX specifier are treated either as regular variable-length columns or as LOB columns, no special discussion of their storage is needed.

The size of values specified with MAX can reach the maximum size supported by LOB data, which is currently 2 GB. By using the MAX specifier, though, you are indicating that the maximum size should be the maximum the system supports. If you upgrade a table with a *varchar(MAX)* column to a later version of SQL Server in the future, the MAX length will be whatever the new maximum is in the new version.

Tip Because the MAX data types can store LOB data as well as regular row data, it is recommended that you use these data types in future development in place of the *text*, *ntext,* or *image* types, which Microsoft has indicated will be removed in a future version.

Note Although the acronym *LOB* can be expanded to mean "Large Object," I will be using these two terms to mean two different things. I will use *LOB* only when I want to refer to the data using the special storage format shown in Figure 7-1. I will use the term *large object* when referring to any of the methods for storing data that might be too large for a regular data page. This includes row-overflow columns, the actual LOB data types, the MAX data types, and filestream data.

Appending Data into a LOB Column

In the storage engine, each LOB column is broken into fragments of a maximum size of 8,040 bytes each. When you append data to a large object, SQL Server finds the append point and looks at the current fragment where the new data will be added. It calculates the size of the new fragment (including the newly appended data). If the size is more than 8,040 bytes, SQL Server allocates new large object pages until a fragment is left that is less than 8,040 bytes, and then it finds a page that has enough space for the remaining bytes.

When SQL Server allocates pages for LOB data, it has two allocation strategies:

1. For data that is less than 64 KB in size, it randomly allocates a page. This page comes from an extent that is part of the large object IAM, but the pages are not guaranteed to be continuous.
2. For data that is more than 64 KB in size, it uses an append-only page allocator that allocates one extent at a time and writes the pages continuously in the extent.

So from a performance standpoint, it is beneficial to write fragments of 64 KB at a time. It might be beneficial to allocate 1 MB in advance if you know that the size will be 1 MB. However, you need to take into account the space required for the transaction log as well. If you a create a 1-MB fragment first with any random contents, SQL Server logs the 1 MB, and then all the changes are logged as well. When you perform large object data updates, no new pages need to be allocated, but the changes still need to be logged.

So long as the large object values are small, they can be in the data page. In this case, some preallocation might be a good idea so that the large object data doesn't become too fragmented. A general recommendation might be that if the amount of data to be inserted into a large object column in a single operation is relatively small, that you insert a large object value of the final expected value, and then replace substrings of that initial value as needed. For larger sizes, try to append or insert in chunks of 8 * 8,040 bytes. This allocates a whole extent each time, and 8,040 bytes are stored on each page.

If you do find that your large object data is becoming fragmented, there is an option to ALTER INDEX REORGANIZE to defragment your large object data. In fact, this option (WITH LOB_ COMPACTION) is on by default, so you just need to make sure that you don't set it to 'OFF'.

Filestream Data

Although the flexible methods that SQL Server uses to store large object data in the database give you many advantages over data stored in the file system, they also have many disadvantages. Some of the benefits of storing large objects in your database include the following:

- Transactional consistency of your large object data can be guaranteed.
- Your backup and restore operations include the large object data, allowing you integrated, point-in-time recovery of your large objects.
- All data can be stored using a single storage and query environment.

Some of the disadvantages of storing large objects in your database include the following:

- Large objects can take a very large number of buffers in cache.
- Updating large objects can cause extensive database fragmentation.
- Database files can become extremely large.

SQL Server 2008 allows you to manage file system objects as if they were part of your database to provide the benefits of having large objects in the database while minimizing the disadvantages. The data stored in the file system is referred to as *filestream* data. As you start evaluating whether filestream data is beneficial for your applications, you must consider both the benefits and the drawbacks. Some of the benefits of filestream data include the following:

- The large object data is stored in the file system but rooted in the database as a 48-byte file pointer value in the column containing the filestream data.
- The large object data is kept transactionally consistent with structured data.
- The large object data is accessible through both Transact-SQL (T-SQL) and the NTFS streaming APIs, which can provide great performance benefits.
- The large object size is limited only by the NTFS volume size, not the old 2-GB limit for LOB data.

Some of the drawbacks of using filestream data include the following:

- Database mirroring cannot be used on databases containing filestream data.
- Database snapshots cannot include the filestream filegroups, so the filestream data is unavailable. A *SELECT* statement in a database snapshot that requests a filestream column generates an error.
- Filestream data can't be encrypted natively by SQL Server.

Enabling Filestream Data for SQL Server

The capability to access filestream data must be enabled both outside and inside your SQL Server 2008 instance, which I mentioned in Chapter 1, “SQL Server 2008 Architecture and Configuration,” when discussing configuration. Through the SQL Server Configuration Manager, you must enable T-SQL access to filestream data, and if that has been enabled, you can also enable file I/O streaming access. If file I/O streaming access is allowed, you can allow remote clients to have access to the streaming data if you want. Once the SQL Server Configuration Manager is opened, make sure you have selected SQL Server Services in the left pane. In the right pane, right-click the SQL instance that you want to configure and select Properties from the drop-down menu. The Properties dialog box has four tabs, including one labeled *FILESTREAM*. You can see the details of the *FILESTREAM* tab of the SQL Server Properties dialog box in Figure 7-3.

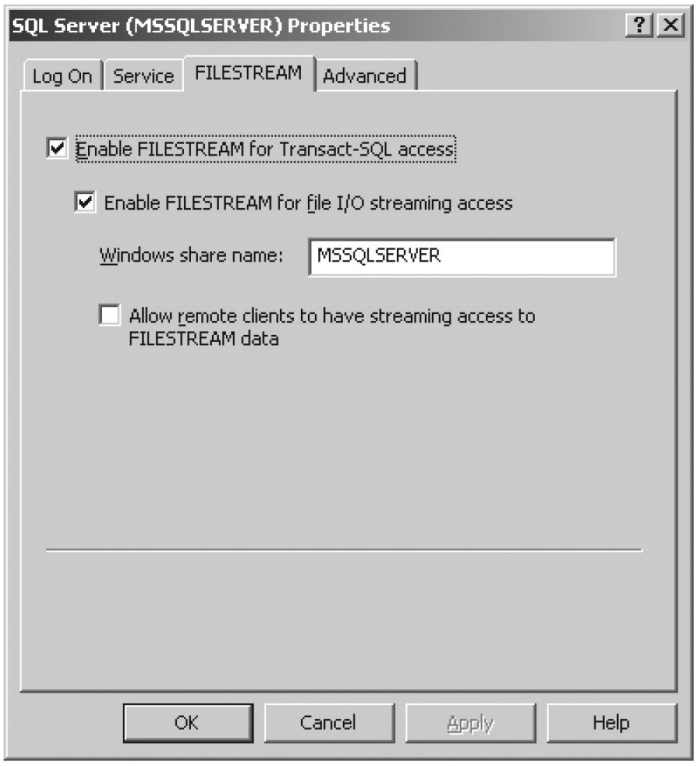

Figure 7-3: Configuring a SQL Server instance to allow FILESTREAM access

After the server instance has been configured, you need to use *sp_configure* to set your SQL Server instance to the level of filestream access that you require. Three values are possible. A value of 0 means that no filestream access is allowed, a value of 1 means that you can use T-SQL to access filestream data, and a value of 2 means that you can use both T-SQL and the Win32 API for filestream access. As with all configuration options, don't forget to run the *RECONFIGURE* command after changing a setting, as shown here:

```
EXEC sp_configure 'filestream access level', 1;
RECONFIGURE;
```

Creating a Filestream-Enabled Database

To store filestream data, a database must have at least one filegroup that has been created to allow filestream data. When creating a database, a filegroup that allows filestream data is specified differently than a filegroup containing row data in several different ways:

- There can be only one file in the filestream filegroup.
- The path specified for the filestream filegroup must exist only up to the last folder name. The last folder name must not exist but will be created when SQL Server creates the database.
- The *size, maxsize,* and *filegrowth* properties do not apply to filestream filegroups.
- If there is no filestream-containing filegroup specified as DEFAULT, the first filestream-containing filegroup listed is the default. (Therefore, there is one default filegroup for row data and one default filegroup for filestream data.)

Look at the following code, which creates a database with two filestream-containing filegroups. The path C:\Data2 must exist, but it must not contain either the *filestream1* or the *filestream2* folders:

```
CREATE DATABASE MyFilestreamDB
ON
PRIMARY ( NAME = Rowdata1,
    FILENAME = 'c:\Data2\Rowdata1.mdf'),
FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM DEFAULT( NAME = FSData1,
    FILENAME = 'c:\Data2\filestream1'),
FILEGROUP FileStreamGroup2 CONTAINS FILESTREAM ( NAME = FSData2,
    FILENAME = 'c:\Data2\filestream2')
LOG ON  ( NAME = FSDBLOG,
    FILENAME = 'c:\Data2\FSDB_log.ldf');
```

When the above *MyFilestreamDB* database is created, SQL Server creates the two folders, *filestream1* and *filestream2*, in the C:\Data2 directory. These folders are referred to as the filestream *containers.* Initially, each container contains an empty folder called *$FSLOG* and a header file called *filestream.hdr*. As tables are created to use filestream space in a container, a folder for each partition or each table containing filestream data is created in the container.

An existing database can be altered to have a filestream filegroup added, and then a subsequent *ALTER DATABASE* command can add a file to the filestream filegroup. Note that you cannot add filestream filegroups to the *master*, *model,* and *tempdb* databases.

Creating a Table to Hold Filestream Data

To specify that a column is to contain filestream data, it must be defined as type *varbinary(MAX)* with a *FILESTREAM* property. The database containing the table must have at least one filegroup defined for *FILESTREAM.* Your table creation statement can specify which filegroup its filestream data is stored in, and if none is specified, the default filestream filegroup is used. Finally, any table that has filestream columns must have a column of the *uniqueidentifier* data type with the *ROWGUIDCOL* attribute specified. This column must not allow NULL values and must be guaranteed to be unique by specifying either the UNIQUE or PRIMARY KEY single-column constraint. The *ROWGUIDCOL* column acts as a key that the *FILESTREAM* agent can use to locate the actual row in the table to check permissions, obtain the physical path to the file, and possibly lock the row if required.

Now let's look at the files that are created within the container. When created in the *MyFilestreamDB* database, the table here adds several folders to the container for the *FileStreamGroup1* container:

```
CREATE TABLE MyFilestreamDB.dbo.Records
(
        [Id] [uniqueidentifier] ROWGUIDCOL NOT NULL UNIQUE,
        [SerialNumber] INTEGER UNIQUE,
        [Chart_Primary] VARBINARY(MAX) FILESTREAM NULL,
        [Chart_Secondary] VARBINARY(MAX) FILESTREAM NULL)
FILESTREAM_ON FileStreamGroup1;
```

Because this table is created on *FileStreamGroup1*, the *filestream1* container is used. One subfolder is created within *filestream1* for each table or partition created in the *FileStreamGroup1* filegroup, and those file names will be GUIDs. Each of those files has a subfolder for each column within the table or partition, which holds filestream data, and again, the names of those folders will be GUIDs. Figure 7-4 shows the structure of my files on disk right after the *MyFilestreamDB.dbo.Records* table is created. The *filestream2* folder only has the $FSLOG subfolder, and no subfolders for any tables. The *filestream1* folder has a GUID-named subfolder for the *dbo.Records* table, and within that, a GUID-named subfolder for each of the two *FILESTREAM* columns in the table. There are still no files except for the original *filestream.hdr* file.

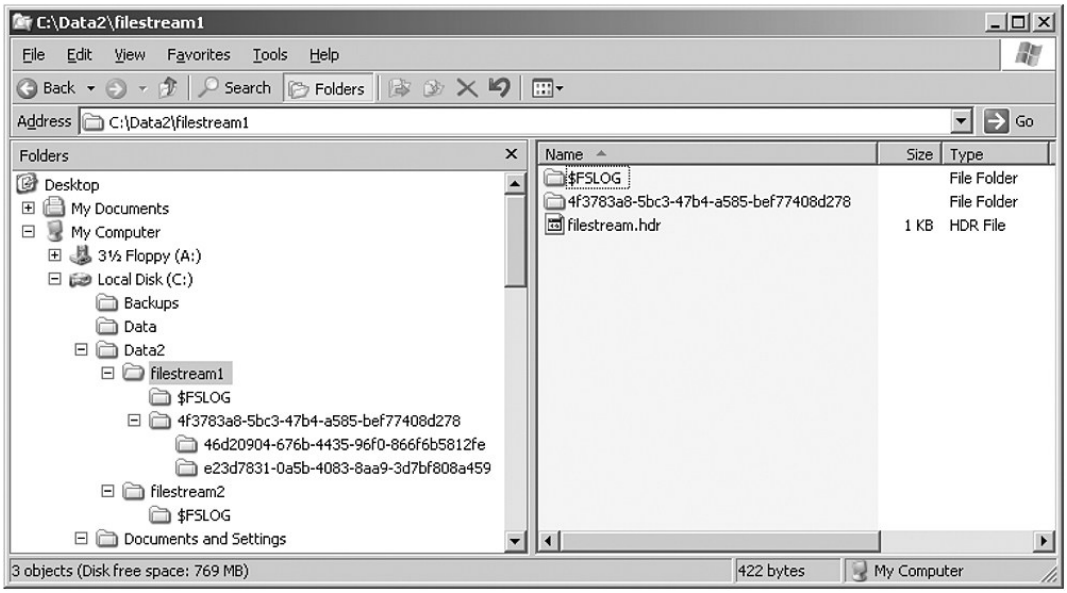

Figure 7-4: The operating system file structure after creating a table with two filestream data columns

Files are not added until we actually insert filestream data into the table.

Warning When the table is dropped, the folders, subfolders, and the files they contain are *not* removed from the file system immediately. They are removed by a Garbage Collection thread, which fires at regular intervals and also when the SQL Server service stops and restarts. You can delete the files manually, but be careful. You can delete folders for a column or table that still exists in the database even while the database is online. Subsequent access to that table generates an error message containing the text "Path not found." You might think that SQL Server should prevent any file that is part of the database from being deleted; but to absolutely prevent the file deletions, SQL Server has to hold open file handles for every single file in all the filestream containers for the entire database, and for large tables, that would not be practical.

Manipulating Filestream Data

Filestream data can be manipulated either using T-SQL or the Win32 API. When using T-SQL, the data can be processed exactly as if it were *varbinary(MAX*). Using the Win32 API requires that you first obtain the file path and current transaction context. You can then open a WIN32 handle and use that handle to read and write the large object data. All the examples in this section use T-SQL. You can get the details of Win32 manipulation from *SQL Server Books Online.*

As you add data to the table, files are added to the subfolders for each column. *INSERT* operations that fail with a run-time error (for example, due to a uniqueness violation) still create a file for each of the filestream columns in the row. Although the row is never accessible, it still uses file system space.

Inserting Filestream Data

Data can be inserted using normal T-SQL *INSERT* statements. Filestream data must be inserted using the *varbinary(MAX)* data type, but any string data can be converted in the *INSERT* statement. The following statement adds one row to the *dbo.Records* table created previously, which has two filestream columns. The first filestream column gets a 90,000-byte character string converted to *varbinary(MAX)* and the second filestream column gets an empty binary string. Note that we first convert the nine-character string *Base Data* to *varchar(MAX)* because a normal string value cannot be more than 8,000 bytes. The *REPLICATE* function returns the same data type as its first parameter, so I want that first parameter to be unambiguously a large object. Replicating the 9-byte string 10,000 times results in a 90,000-byte string, which is then converted to *varbinary(MAX):*

```
USE MyFileStreamDB
INSERT INTO dbo.Records
    SELECT newid (), 24,
      CAST (REPLICATE (CONVERT(varchar(MAX), 'Base Data'), 10000)
                       AS varbinary(max)),
      0x;
```

Note that a value of 0x is an empty binary string, which is not the same as a NULL. Every row that has a non-NULL value in a *FILESTREAM* column has a file, even for zero-length values.

Figure 7-5 shows you what your file system would look like after running the previous code to create a database with two filestream containers and create a table with two *FILESTREAM* columns, and then inserting one row into that table. In the left pane, you can see the two filestream containers (*filestream1* and *filestream2*).

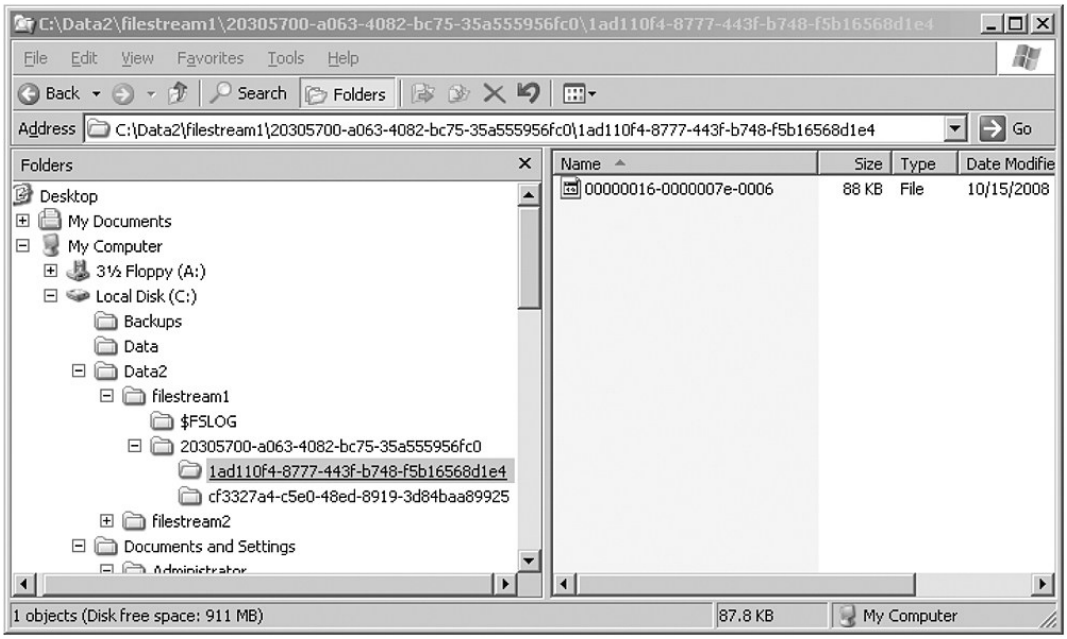

Figure 7-5: The operating system file structure after inserting filestream data

The *filestream1* container has a folder with a GUID name for the *dbo.Records* table that I created, and that folder container has two folders with GUID names, for the two columns in that table. The right pane shows you the file containing the actual data inserted into one of the columns.

Updating Filestream Data

When an *UPDATE* statement is used to modify a filestream column, the file containing the data is modified and the file increases or decreases in size as appropriate. Specifically, setting the column to an empty, zero-length value causes the file to have a size of zero. Also, in this first release, the T-SQL "chunked update," specified with the .WRITE clause, is not supported. It is recommended that you use file system streaming access for manipulation (both inserts and updates) of your filestream data. Updates to *FILESTREAM* data are always performed as a *DELETE* followed by an *INSERT*, so you see a new row in the directory for the column(s) updated.

When a filestream cell is set to NULL, the filestream file associated with that cell is deleted when the Garbage Collection thread runs. (I'll tell you about Garbage Collection later in this chapter.)

Deleting Filestream Data

When a row is deleted through the use of a *DELETE* or a *TRUNCATE TABLE* statement, any *FILESTREAM* file associated with the row is deleted. However, the delete of the file is not synchronous with the deletion of the row. The file is deleted by the *FILESTREAM* Garbage Collection thread. This is also true for *DELETE*s that are generated as part of an *UPDATE.* A new row is added, but the old one is not physically removed until Garbage Collection runs.

> **Note** The OUTPUT clause for data manipulation operations (*INSERT, UPDATE, DELETE,* and *MERGE*) is supported the same way as it is for column modifications. However, you need to be careful if you are using the OUTPUT clause to insert into a table with a *varbinary(MAX)* column instead without the filestream specifier. If the filestream data is larger than 2 GB, the insert of filestream data into the table may result in a run-time error.

Filestream Data and Transactions

Filestream data manipulation is fully transactional. But you need to be aware that when you are manipulating *FILESTREAM* data, not all isolation levels are supported. In addition, some isolation levels are supported for T-SQL access but not for filesystem access. Table 7-2 indicates which isolation levels are available in which access mode.

Table 7-2: Isolation Levels Supported with Filestream Data Manipulation

Isolation Level	T-SQL Access	Filesystem Access
Read uncommitted	Supported	Not supported

Read committed	Supported	Supported
Repeatable read	Supported	Not supported
Serializable	Supported	Not supported
Read committed snapshot	Supported	Not supported
Snapshot	Supported	Not supported

If two processes trying to access the same *FILESTREAM* datafile are in incompatible modes, the filesystem APIs fail with an ERROR_SHARING_VIOLATION message instead of just blocking, as would happen when using T-SQL. As with all data access, readers and writers within the same transaction can never get a conflict on the same file but unlike non-*FILESTREAM* access, two write operations within the same transaction can end up conflicting with each other when accessing the same file, unless the file handle has been previously closed. You can read much more about transactions, isolation levels, and conflicts in Chapter 10, “Transactions and Concurrency.”

Logging Filestream Changes

As mentioned previously, each *FILESTREAM* filegroup has a $FSLOG folder that keeps track of all filestream activity that touches that filegroup. The data in this folder is used when you perform transaction log backup and restore operations in the database (which include the *FILESTREAM* filegroup) and also during the recovery process.

The $FSLOG folder primarily keeps track of new information added to the filestream filegroup. A file gets added to the log folder to reflect each of the following:

- A new table containing filestream data is created.
- A *FILESTREAM* column is defined.
- A new row is inserted containing non-NULL data in the *FILESTREAM* column.
- A *FILESTREAM* value is updated.
- A *COMMIT* occurs.

Here are some examples:

- If you create a table containing two filestream columns, four files are added to the $FSLOG folder—one for the table, two for the columns, and one for the implied *COMMIT.*
- If you insert a single row containing filestream data in an autocommit transaction, two files will be added to the $FSLOG folder—one for the *INSERT* and one for the *COMMIT.*
- If you insert five rows in an explicit transaction, six files are added to the $FSLOG folder.

Files are not added to the $FSLOG folder when data is deleted or when a table is truncated or dropped. However, the SQL Server transaction log keeps track of these operations, and a new metadata table contains information about the data that has been removed.

Garbage Collection for Filestream Data

The filestream data can be viewed as serving as the live user data, as well as the log of changes to that data, and as row versions for snapshot operations (discussed in Chapter 10). SQL Server needs to make sure that the filestream data files are not removed if there is any possibility they might be needed for any backup or recovery needs. In particular, for log backups, all new filestream content must be backed up since the transaction log does not contain the actual filestream data, and only the filestream data has the redo information for the actual *FILESTREAM* contents. In general, if your database is not in the SIMPLE recovery mode, you need to back up the log twice before the Garbage Collector can remove unneeded data files from your *FILESTREAM* folders. Let’s look at an example. We’ll start with a clean slate, by dropping and re-creating the *MyFilestreamDB* database. A *DROP DATABASE* statement immediately removes all the folders and files because now there is no chance we’re going to do any subsequent log backups. The script given here re-creates the database and creates a table with just a single *FILESTREAM* column. Finally, the script inserts three rows into the table and backs up the database. If you inspect the *filestream1* container, you see that the folder for the columns contains three files for the three rows:

```
USE master;
```

```
GO
DROP DATABASE MyFilestreamDB;
GO
CREATE DATABASE  MyFilestreamDB  ON  PRIMARY
     (NAME = N'Rowdata1', FILENAME = N'c:\data\Rowdata1.mdf' , SIZE = 2304KB ,
      MAXSIZE = UNLIMITED, FILEGROWTH = 1024KB ),
 FILEGROUP  FileStreamGroup1  CONTAINS FILESTREAM  DEFAULT
    (NAME = N'FSData1', FILENAME = N'c:\data\filestream1' ),
 FILEGROUP  FileStreamGroup2  CONTAINS FILESTREAM
    (NAME = N'FSData2', FILENAME = N'c:\data\filestream2' )
 LOG ON
    (NAME = N'FSDBLOG', FILENAME = N'c:\data\FSDB_log.ldf' , SIZE = 1024KB ,
     MAXSIZE = 2048GB , FILEGROWTH = 10%);
GO
USE MyFilestreamDB;
GO
CREATE TABLE dbo.Records
(
        Id [uniqueidentifier] ROWGUIDCOL NOT NULL UNIQUE,
        SerialNumber INTEGER UNIQUE,
        Chart_Primary VARBINARY(MAX) FILESTREAM NULL
)
FILESTREAM_ON FileStreamGroup1;
GO
INSERT INTO dbo.Records
    VALUES (newid(), 1,
               CAST (REPLICATE (CONVERT(varchar(MAX), 'Base Data'),
                     10000) as varbinary(max))),
           (newid(), 2,
               CAST (REPLICATE (CONVERT(varchar(MAX), 'New Data'),
                     10000) as   varbinary(max))),
           (newid(), 3, 0x);
GO
BACKUP DATABASE MyFileStreamDB to disk = 'C:\backups\FBDB.bak';
GO
```

Now delete one of the rows, as follows:

```
DELETE dbo.Records
WHERE SerialNumber = 2;
GO
```

Now inspect the files on disk, and you still see three files.

Back up the log and run a checkpoint. Note that in a real system, enough changes would probably be made to your data that your database's log would get full enough to trigger an automatic *CHECKPOINT*. However, during testing, I'm not putting much into the log at all, so I have to force the *CHECKPOINT*:

```
BACKUP LOG  MyFileStreamDB to disk = 'C:\backups\FBDB_log.bak';
CHECKPOINT;
```

Now if you check the *FILESTREAM* data files, you still see three rows. Wait five seconds for Garbage Collection, and you'll still see three rows. We need to back up the log and then force another *CHECKPOINT*:

```
BACKUP LOG  MyFileStreamDB to disk = 'C:\backups\FBDB_log.bak';
CHECKPOINT;
```

Now within five seconds, you should see one of the files disappear. The reason that we need to back up the log twice before the physical file is available for garbage collection is to make sure that the file space is not reused by other filestream operations while it still might be needed for restore purposes.

You can run some additional tests of your own. For example, if you try dropping the *dbo.Records* table, notice that you again have to perform two log backups and *CHECKPOINT*s before SQL Server removes the folders for the table and the column.

Metadata for Filestream Data

Within your SQL Server tables, the storage required for filestream is not particularly complex. In the row itself, each

filestream column contains a file pointer that is 48 bytes in size. Even if you look at a data page with the *DBCC PAGE* command, there is not much more information about the file that is available. However, SQL Server does provide a new function to translate the file pointer to a path name. The function is actually a method applied to the column name in the table. So the following code returns a UNC name for the file containing the actual column's data in the row I inserted previously:

```
SELECT Chart_Primary, Chart_Primary.PathName()
FROM dbo.Records
WHERE SerialNumber = 24;
GO
```

The UNC value returned looks like this:

```
\\<server_name>\<share_name>\v1\<db_name>\<object_schema>\<table_name>\<column_name>\<GUID>
```

Keep in mind the following points about using the *PathName* function:

- The function name is case-sensitive, even on a server that is not case-sensitive, so it always must be entered as *PathName.*
- The default *share_name* is the service name for your SQL Server instance. (So for the default instance, it is MSSQLSERVER.) Using the SQL Server Configuration Manager, you can right-click your SQL Server instance and choose Properties. The *FILESTREAM* tab of the SQL Server Properties dialog box allows you to change the *share_name* to another value of your choosing.
- The *PathName* function can take an optional parameter of 0, 1, or 2, with 0 being the default. The parameter controls only how the *server_name* value is returned; all other values in the UNC string are unaffected. Table 7-3 shows the meanings of the different values.

Table 7-3: Parameter Values for the PathName Function

Value	Description
0	Returns the server name converted to BIOS format; for example: \\SERVERNAME\MSSQLSERVER\v1 \MyFilestream\dbo\Records\Chart_Primary\ A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9
1	Returns the server name without conversion; for example: \\ServerName\MSSQLSERVER\v1\MyFilestream\Dbo\Records\ Chart_Primary\A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9
2	Returns the complete server path; for example: \\ServerName.MyDomain.com\MSSQLSERVER\v1 \MyFilestream\Dbo\Records\ Chart_Primary\A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9

Some of the metadata added in SQL Server 2005 has been enhanced to give you information about your filestream data:

- **sys.database_files** returns a row for each of your filestream files. These files have a *type* value of 2 and a *type_desc* value of *FILESTREAM*.
- **sys.filegroups** returns a row for each of your filestream filegroups. These files have a *type* value of FD and a *type_desc* value of *FILESTREAM_DATA_FILEGROUP*.
- **sys.data_spaces** returns one row for each data space, which is either a filegroup or a partition scheme. Filegroups holding filestream data are indicated by the type FD.
- **sys.tables** has a value in the column for *filestream_data_space_id*, which is the data space ID for either the filestream filegroup or the partition scheme that the filestream data uses. Tables with no filestream data have NULL in this column.
- **sys.columns** has a value of 1 in the *is_filestream* column for columns with the filestream attribute.

The older metadata, such as the system procedure *sp_helpdb* <database_name> or *sp_help* <object_name>, does not show any information about filestream data.

I mentioned previously that rows or objects that are deleted do not generate files in the $FSLOG folder, but data about the removed data is stored in a system table. No metadata view allows you to see this table; you can observe it only by using the dedicated administrator connection (DAC). You can look in a view called *sys.internal_tables* for an object with *TOMBSTONE* in its name. Then using the DAC, you can look at the data inside the *TOMBSTONE* table. If you rerun the

above script but don't back up the log, you can use the following script:

```
USE MyFilestreamDB;
GO
SELECT name FROM sys.internal_tables
WHERE name like '%tombstone%';

-- I see the table named: filestream_tombstone_2073058421
-- Reconnect using DAC, which puts us in the master database
USE MyFileStreamDB;
GO
SELECT * FROM sys.filestream_tombstone_2073058421;
GO
```

If this table is empty, then the log in SQL Server and the $FSLOG are in sync, and all unneeded files have been removed from the *FILESTREAM* containers on disk.

Performance Considerations for Filestream Data

Although a thorough discussion of performance tuning and troubleshooting is beyond the scope of this book, I want to provide you with some basic information about setting up your system to get high performance from filestream data. Paul Randal, one of the co-authors of this book, has written a white paper on *FILESTREAM* that you can access on the MSDN site at *http://msdn.microsoft.com/en-us/library/cc949109.aspx.* (This white paper is also available on this book's companion Web site, *http://www.SQLServerInternals.com/companion*.) In this section, I'll just briefly mention some of the main points Paul makes regarding what you can do to get good performance. All these suggestions are explained in much more detail in the white paper.

- Make sure you're storing the right-sized data in the right way. Jim Gray (et al.) published a research paper a couple of years ago called "To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?" To summarize the findings, large object data smaller than 256 KB should be stored in a database, and data that is 1 MB or larger should be stored in the file system. For data between these two values, the answer depends on other factors and you should test your application thoroughly. The key point here is that you won't get good performance if you store lots of relatively small large objects using *FILESTREAM*.
- Use an appropriate RAID level for the NTFS volume that hosts the *FILESTREAM* data container. For example, don't use RAID-5 for a write-intensive workload.
- Use an appropriate disk technology. SCSI is usually faster than SATA/IDE because SCSI drives usually have higher rotational speeds, so they have lower latency and seek times. However, SCSI drives are also more expensive.
- Whichever disk technology you choose, if it is SATA, ensure that it supports NCQ, and if SCSI, ensure that it supports CTQ. Both of these allow the drives to process multiple, interleaved I/Os concurrently.
- Separate the data containers from each other, and separate the containers from other database data and log files. This avoids contention for the disk heads.
- Defragment the NTFS volume if needed before setting up *FILESTREAM*, and defragment periodically to maintain good scan performance.
- Turn off 8.3 name generation on the NTFS volume using the command-line *fsutil* utility. This is an order-N algorithm that has to check that the new name generated doesn't collide with any existing names in the directory. Note, however, that this slows insert and update performance down a lot.
- Turn off tracking of last access time using *fsutil.*
- Set the NTFS cluster size appropriately. For larger objects greater than 1 MB in size, use a cluster size of 64 KB to help reduce fragmentation.
- A partial update of *FILESTREAM* data creates a new file. Batch lots of small updates into one large update to reduce churn.
- When streaming the data back to the client, use an SMB buffer size of approximately 60 KB or multiples thereof. This helps keep the buffers from getting overly fragmented, because Transmission Control Protocol/Internet Protocol (TCP/IP) buffers are 64 KB.

Taking these suggestions into consideration and performing thorough testing of your application can give you great performance when working with very large data objects.

Sparse Columns

In this section, we'll look at another special storage format, added in SQL Server 2008. Sparse columns are ordinary columns that have an optimized storage format for NULL values. Sparse columns reduce the space requirements for NULL values, allowing you to have many more columns in your table definition, so long as most of them are NULL. The cost of using sparse columns is that there will be more overhead to store and retrieve non-NULL values.

Sparse columns are intended to be used for tables storing data describing entities with many possible attributes, where most of the attributes will be NULL for most rows. For example, a content management system like Microsoft Windows SharePoint Services may need to keep track of many different types of data in a single table. Different properties apply to different subsets of rows in the table. So for each row, only a small subset of the columns is populated with values. Another way of looking at it is that for any particular property, only a subset of rows has a value for that property. The sparse columns in SQL Server 2008 allow us to store a very large number of possible columns for a single row. For this reason, the SPARSE column feature is sometimes also referred to as the *wide-table* feature.

Management of Sparse Columns

It is recommended that you don't consider defining a column as SPARSE unless at least 90 percent of the rows in the table are expected to have NULL values for that column. This is not an enforced limit, however, and you can define almost any column as SPARSE. Sparse columns save space on NULL values.

This new SQL Server 2008 feature allows you to have far more columns that you ever could before. The limit is now 30,000 columns in a table, with no more than 1,024 of them being non-sparse. (Computed columns are considered non-sparse.) Obviously, not all 30,000 columns could have values in them. The number of populated columns you can have depends on the bytes of data in the row. Sparse columns optimize the storage size for NULL values, which take no space at all for sparse columns, unlike non-sparse columns, which do need space even for NULLs. (As we saw in Chapter 5, a fixed-length NULL column always uses the whole column width, and a variable-length NULL column uses at least two bytes in the column offset array.). Although the sparse columns themselves take no space, some fixed overhead is needed to allow for sparse columns in a row. As soon as you define even one column with the SPARSE attribute, SQL Server adds a sparse vector to the end of the row. We'll see the actual structure of this sparse vector in the section entitled "Physical Storage," later in this chapter, but to start with, you should be aware even with sparse columns, the maximum size of a data row (excluding LOB and row-overflow) remains at 8,060, including overhead bytes. Because the sparse vector includes additional overhead, the maximum number of bytes for the rest of the rows decreases. In addition, the size of all fixed-length non-NULL sparse columns in a row is limited to 8,019 bytes.

Table Creation

Creating a table with sparse columns is very straightforward, as you can just add the attribute SPARSE to any column of any data type except *text*, *ntext*, *image*, *geography*, *geometry*, *timestamp,* or any user-defined data type. In addition, sparse columns cannot include the *IDENTITY, ROWGUIDCOL,* or *FILESTREAM* attributes. A sparse column cannot be part of a clustered index or part of the primary key. Tables containing sparse columns cannot be compressed, either at the row level or the page level. (I'll discuss compression in detail in the next section.) There are also a few other restrictions, particularly if you are partitioning a table with sparse columns, so you should check the documentation for full details. The examples in this section are necessarily very simple because it would be impractical to print code examples with enough columns to make sparse columns really useful. The following example shows the creation of two very similar tables, one that doesn't allow sparse columns and another that does. I attempt to insert the same rows into each table. Because a row allowing sparse columns has a smaller maximum length, it fails when trying to insert a row that the table with no sparse columns has no problem with:

```
USE test;
GO
CREATE TABLE test_nosparse
(
  col1 int,
  col2 char(8000),
  col3 varchar(8000)
);
GO
INSERT INTO test_nosparse
```

```
        SELECT null, null, null;
INSERT INTO test_nosparse
        SELECT 1, 'a', 'b';
GO
```

These two rows can be inserted with no error. Now, build the second table:

```
CREATE TABLE test_sparse
(
 col1 int SPARSE,
 col2 char(8000) SPARSE,
 col3 varchar(8000) SPARSE
);
GO
INSERT INTO test_sparse
        SELECT NULL, NULL, NULL;
INSERT INTO test_sparse
        SELECT 1, 'a', 'b';
GO
```

The second *INSERT* statement generates the following error:

```
Msg 576, Level 16, State 5, Line 2
Cannot create a row that has sparse data of size 8042 which is greater than the allowable
maximum sparse data size of 8019.
```

Although the second row inserted into the *test_sparse* table looks just like a row that was inserted successfully into the *test_nosparse* table, internally it is not. The total of the sparse columns is 4 bytes for the *int,* plus 8,000 bytes for the *char,* and 24 bytes for the row-overflow pointer, which is greater than the 8,019-byte limit.

Altering a Table

Tables can be altered to convert a non-sparse column into a sparse column, or vice versa. Be careful, however, because if you are altering a very large row in a table with no sparse columns, changing one column to be sparse reduces the number of bytes of data that are allowed on a page. This can result in an error being thrown in cases where an existing column is converted into a sparse column. For example, the following code creates a table with large rows, but my *INSERT* statements, with or without NULLs, are accepted. However, when we try to make one of the columns SPARSE, even a relatively small column like the 8-byte *datetime* column, the extra overhead makes the existing rows too large and the *ALTER* fails:

```
IF EXISTS (SELECT * FROM sys.tables WHERE name = 'test_nosparse_alter')
                DROP TABLE test_nosparse_alter;
GO
CREATE TABLE test_nosparse_alter
(
c1 int,
c2 char(4020) ,
c3 char(4020) ,
c4 datetime
);
GO
INSERT INTO test_nosparse_alter SELECT NULL, NULL, NULL, NULL;
INSERT INTO test_nosparse_alter SELECT 1, 1, 'b', GETDATE();
GO
ALTER TABLE test_nosparse_alter
  ALTER COLUMN c4 datetime SPARSE;
```

We receive this error:

```
Msg 1701, Level 16, State 1, Line 2
Creating or altering table 'test_nosparse_alter' failed because the minimum row size would
be 8075, including 23 bytes of internal overhead. This exceeds the maximum allowable table
row size of 8060 bytes.
```

In general, sparse columns can be treated just like any other column, with only a few restrictions. In addition to the restrictions mentioned previously on the data types that cannot be defined as SPARSE, there are also the following limitations to keep in mind:

- A sparse column cannot have a default value.

- A sparse column cannot be bound to a rule.
- Although a computed column can refer to a sparse column, a computed column cannot be marked as SPARSE.
- A sparse column cannot be part of a clustered index or a unique primary key index. However, both persisted and nonpersisted computed columns that refer to sparse columns can be part of a clustered key.
- A sparse column cannot be used as a partition key of a clustered index or heap. However, a sparse column can be used as the partition key of a nonclustered index.

Note that except for the requirement that sparse columns cannot be part of the clustered index or primary key, there aren't any other restrictions on building indexes on sparse columns. However, if you are using sparse columns the way they are intended to be used, and the vast majority of your rows have NULL for the sparse columns, any regular index on a sparse column is very inefficient and may have limited usefulness. Sparse columns are really intended to be used with filtered indexes, which are discussed in Chapter 6.

Column Sets and Sparse Column Manipulation

If sparse columns are used as intended, only a few columns in each row have values, and your *INSERT* and *UPDATE* statements are relatively straightforward. For *INSERT* statements, you can specify a column list and then specify values only for those few columns in the column list. For *UPDATE* statements, there are only a few columns in each row whose values can be manipulated. The only time you need to be concerned about how to deal with a potentially very large list of columns is if you are selecting data without listing individual columns (that is, using a *SELECT *)*. Good developers know that using *SELECT* * is never a good idea, but SQL Server needs a way of dealing with a result set with potentially thousands (or tens of thousands) of columns. The mechanism provided to help deal with *SELECT* * is a construct called a COLUMN_SET. A COLUMN_SET is an untyped XML representation that combines multiple columns of a table into a structured output. You can think of a COLUMN_SET as a nonpersisted computed column because the COLUMN_SET is not physically stored in the table. In this release of SQL Server, the only possible COLUMN_SET contains all the sparse columns in the table. Future versions may allow us to define other COLUMN_SET variations.

A table can only have one COLUMN_SET defined, and once a table has a COLUMN_SET defined, *SELECT* * no longer returns individual sparse columns. Instead, it returns an XML fragment containing all the non-NULL values for the sparse columns. Let's look at an example.

The following code builds a table containing an identity column, 25 sparse columns, and a column set:

```
USE test;
GO
IF EXISTS (SELECT * FROM sys.tables WHERE name = 'lots_of_sparse_columns')
                DROP TABLE lots_of_sparse_columns;
GO
CREATE TABLE lots_of_sparse_columns
(ID int IDENTITY,
 col1 int SPARSE,
 col2 int SPARSE,
 col3 int SPARSE,
 col4 int SPARSE,
 col5 int SPARSE,
 col6 int SPARSE,
 col7 int SPARSE,
 col8 int SPARSE,
 col9 int SPARSE,
 col10 int SPARSE,
 col11 int SPARSE,
 col12 int SPARSE,
 col13 int SPARSE,
 col14 int SPARSE,
 col15 int SPARSE,
 col16 int SPARSE,
 col17 int SPARSE,
 col18 int SPARSE,
 col19 int SPARSE,
 col20 int SPARSE,
 col21 int SPARSE,
 col22 int SPARSE,
```

```
 col23 int SPARSE,
 col24 int SPARSE,
 col25 int SPARSE,
 sparse_column_set XML COLUMN_SET FOR ALL_SPARSE_COLUMNS);
 GO
```

Next, I insert values into 3 of the 25 columns, specifying individual column names:

```
INSERT INTO lots_of_sparse_columns (col4, col7, col12)  SELECT 4,6,11;
```

You can also insert directly into the COLUMN_SET, specifying values for columns in an XML fragment. Being able to update the COLUMN_SET is another feature that differentiates COLUMN_SETs from computed columns:

```
INSERT INTO lots_of_sparse_columns (sparse_column_set)
               SELECT '<col8>42</col8><col17>0</col17><col22>30000</col22>';
```

Here are the results when I run *SELECT* * from this table:

```
SELECT * FROM lots_of_sparse_columns;
Results:
ID      sparse_column_set
------- -----------------------------------------------------
1       <col4>4</col4><col7>6</col7><col12>11</col12>
2       <col8>42</col8><col17>0</col17><col22>30000</col22>
```

We can still select from individual columns, either instead of or in addition to selecting the entire COLUMN_SET. So the following *SELECT* statements are both valid:

```
SELECT ID, col10, col15, col20
    FROM lots_of_sparse_columns;
SELECT *, col11
    FROM lots_of_sparse_columns;
```

Keep the following points in mind if you decide to use sparse columns in your tables:

- Once defined, the COLUMN_SET cannot be altered. To change a COLUMN_SET, you must drop and re-create the COLUMN_SET column.
- A COLUMN_SET can be added to a table that does not include any sparse columns. If sparse columns are later added to the table, they appear in the column set.
- A COLUMN_SET is optional and is not required to use sparse columns.
- Constraints or default values cannot be defined on a COLUMN_SET.
- Distributed queries are not supported on tables that contain COLUMN_SETs.
- Replication does not support COLUMN_SETs.
- The Change Data Capture feature does not support COLUMN_SETs.
- A COLUMN_SET cannot be part of any kind of index. This includes XML indexes, full-text indexes, and indexed views. A COLUMN_SET cannot be added as an included column in any index.
- A COLUMN_SET cannot be used in the filter expression of a filtered index or filtered statistics.
- When a view includes a COLUMN_SET, the COLUMN_SET appears in the view as an XML column.
- XML data has a size limit of 2 GB. If the combined data of all the non-NULL sparse columns in a row exceeds this limit, the operation produces an error.
- Copying all columns from a table with a COLUMN_SET (using either *SELECT * INTO* or *INSERT INTO SELECT **) does not copy the individual sparse columns. Only the COLUMN_SET, as data type XML, is copied.

Now let's look at how sparse columns are actually stored.

Physical Storage

At a high level, you can think of sparse columns as being stored much as they are displayed using the COLUMN_SET; that

is, as a set of (column-name, value) pairs. So if there is no value for a particular column, it is not listed and no space at all is required. If there is a value for a column, not only does SQL Server need to store the value, but it needs to store information about which column has that value. So non-NULL sparse columns take more space than their NULL counterparts. To see the difference graphically, you can compare Tables 7-4 and 7-5.

Table 7-4: Representation of a Table Defined with Non-SPARSE Columns, with Many NULL Values

ID	sc1	sc2	sc3	sc4	sc5	sc6	sc7	sc8	sc9
1	1								9
2		2		4					
3						6	7		
4	1				5				
5				4				8	
6			3						9
7					5		7		
8		2						8	
9			3			6			

Table 7-5: Representation of a Table Defined with SPARSE Columns, with Many NULL Values

ID	<sparse columns>
1	(sc1,sc9)(1,9)
2	(sc2,sc4)(2,4)
3	(sc6,sc7)(6,7)
4	(sc1,sc5)(1,5)
5	(sc4,sc8)(4,8)
6	(sc3,sc9)(3,9)
7	(sc5,sc7)(5,7)
8	(sc2,sc8)(2,8)
9	(sc3,sc6)(3,6)

Table 7-4 represents a table with non-sparse columns. You can see a lot of wasted space when most of the columns are NULL. Table 7-5 shows what the same table looks like if all the columns except the ID are defined as SPARSE. All that is stored are the names of all the non-NULL columns and their values.

SQL Server keeps track of the physical storage of SPARSE columns with a structure within a row called a *sparse vector.* Sparse vectors are present only in the data records of a base table that has at least one sparse column declared and each data record of these tables contains a sparse vector.

A sparse vector is stored as a special variable-length column at the end of a data record. It is a special system column, and there is no metadata about this column in *sys.columns* or any other view. The sparse vector is stored as the last variable-length column in the row. The only thing after the sparse vector would be versioning information, used primarily with Snapshot isolation, as is discussed in Chapter 10. There is no bit in the NULL bitmap for the sparse vector column (if a sparse vector exists, it is never NULL), but the count in the row of the number of variables columns includes the sparse vector. You may want to revisit Figure 5-10 in Chapter 5 at this time to familiarize yourself with the general structure of data rows.

Table 7-6 lists the meaning of the bytes in the sparse vector.

Table 7-6: Bytes in a Sparse Vector

Name	Number of Bytes	Meaning
Complex Column Header	2	A value of 05 indicates that the complex column is a sparse vector.
Sparse Column Count	2	Number of sparse columns.
Column ID Set	2 * the number of sparse columns	Two bytes for the column ID of each column in the table with a value stored in the sparse vector.
Column Offset Table	2 * the number of sparse columns	Two bytes for the offset of the ending position of each sparse column.
Sparse Data	Depends on actual values	Data

Let's look at the bytes of a row containing SPARSE columns. First, build a table containing two sparse columns, and populate it with three rows:

```
USE test;
GO
IF EXISTS (SELECT * FROM sys.tables WHERE name = 'sparse_bits')
                DROP TABLE sparse_bits;
GO
CREATE TABLE sparse_bits
(
c1 int IDENTITY,
c2 varchar(4),
c3 char(4) SPARSE,
c4 varchar(4) SPARSE
);
GO
INSERT INTO sparse_bits SELECT 'aaaa', 'bbbb', 'cccc';
INSERT INTO sparse_bits SELECT 'dddd', null, 'eeee';
INSERT INTO sparse_bits SELECT 'ffff', null, 'gg';
GO
```

Now we can use *DBCC IND* to find the page number for the data page storing these three rows and then use *DBCC PAGE* to look at the bytes on the page:

```
DBCC IND(test, sparse_bits, -1);
GO
-- The output indicated that the data page for my table was on page 289;
DBCC TRACEON(3604);
DBCC PAGE(test, 1, 289, 1);
```

I won't show you the entire page output, but only the output for the first row (spread over three lines of output):

```
00000000:   30000800 01000000 02000002 00150029 †0..............)
00000010:   80616161 61050002 00030004 00100014 †.aaaa...........
00000020:   00626262 62636363 63††††††††††††††††.bbbbcccc
```

The grayed bytes are the sparse vector. I can find it easily because it starts right after the last non-sparse variable-length column, which contained *aaaa,* or 61616161, and continues to the end of the row. Figure 7-6 translates the sparse vector according to the meanings given in Table 7-6. Don't forget that you need to byte-swap numeric fields before translating. For example, the first two bytes are 05 00, which need to be swapped to get the hex value 0x0005. Then you can convert it to decimal.

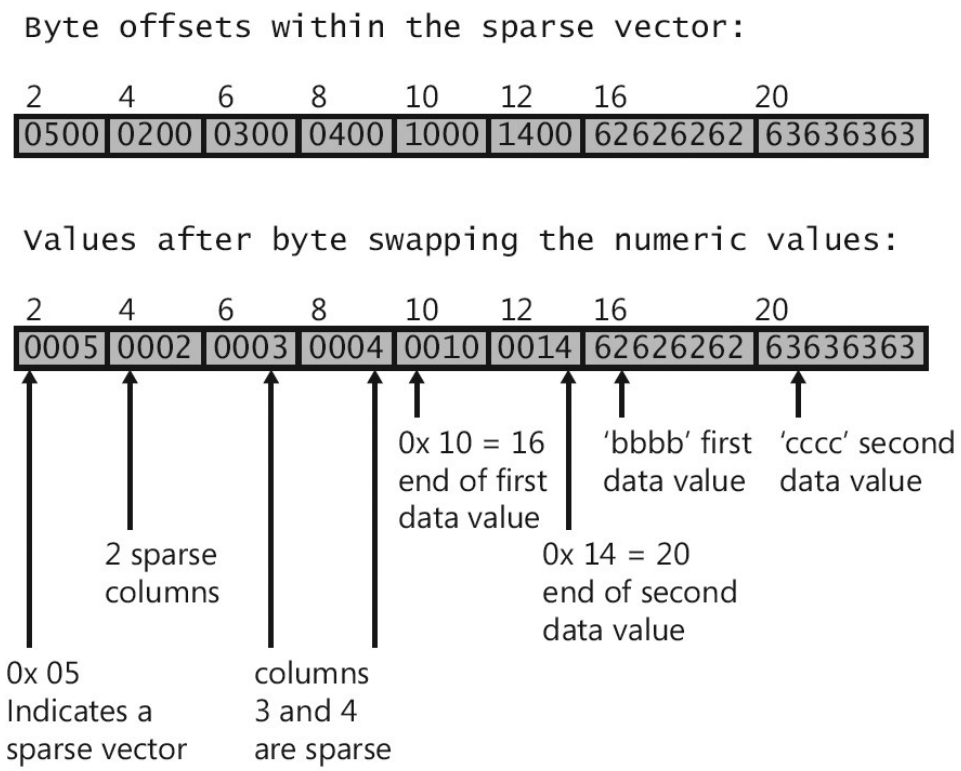

Figure 7-6: Interpretation of the actual bytes in a sparse vector for a row in the sparse_bits table

You can apply the same analysis to the bytes in the other two rows on the page. Here are some things to note:

- No information about columns with NULL values appears in the sparse vector.
- Within the sparse vector, there is no difference in storage between fixed-length and variable-length strings. However, that doesn't mean you should use the two interchangeably. A SPARSE *varchar* column that doesn't fit in the 8,060 bytes can be stored as row-overflow data; a SPARSE *char* column cannot be.
- Because only two bytes are used to store the number of sparse columns, this sets the limit on the maximum number of sparse columns.
- The two bytes for the complex column header indicate that there might be other possibilities for complex columns. At this time, the only other type of complex column that can be stored is one storing a back-pointer, as SQL Server does when it creates a forwarded record. (I discussed forwarded records when discussing updates to heaps in Chapter 5.)

Metadata

Very little extra metadata is needed to support SPARSE columns. The catalog view *sys.columns* contains the following two columns to keep track of SPARSE columns in your tables. Each of these columns has only two possible values, 0 or 1:

- *is_sparse*
- *is_column_set*

Corresponding to these column properties in *sys.columns,* the property function *COLUMNPROPERTY()* also has the following two properties related to SPARSE columns:

- *IsSparse*
- *IsColumnSet*

If I want to inspect all the tables I had created with "sparse" in their name and determine which of their columns were SPARSE, which were column sets, and which were neither, I could run the following query:

```
SELECT OBJECT_NAME(object_id) as 'Table', name as 'Column', is_sparse, is_column_set
FROM sys.columns
WHERE OBJECT_NAME(object_id) like '%sparse%';
```

If I want to see just the table and column names for all COLUMN_SET columns, I could run the following query:

```
SELECT OBJECT_NAME(object_id) as 'Table', name as 'Column'
FROM sys.columns
WHERE COLUMNPROPERTY(object_id, name, 'IsColumnSet') = 1;
```

Storage Savings with Sparse Columns

The SPARSE column feature is designed to save you considerable space when most of your values are NULL. In fact, as mentioned previously, columns that are not NULL but are defined as SPARSE take up more space than if they weren't defined as SPARSE because the sparse vector has to store a couple of extra bytes to keep track of them. To start to see the space differences, you can run the following script, which creates four tables with relatively short, fixed-length columns. Two have sparse columns and two don't. Rows are inserted into each of the tables in a loop, which inserts 100,000 rows. One table with sparse columns is populated with rows with NULL values, and the other is populated with rows that are not NULL. One of the tables with no sparse columns is populated with rows with NULL values, and the other is populated with rows that are not NULL:

```
USE test;
GO
SET NOCOUNT ON;
GO
IF EXISTS (SELECT * FROM sys.tables
                        WHERE name = 'sparse_nonulls_size')
                DROP TABLE sparse_nonulls_size;
GO
CREATE TABLE sparse_nonulls_size
(col1 int IDENTITY,
 col2 datetime SPARSE,
 col3 char(10) SPARSE
 );
GO
IF EXISTS (SELECT * FROM sys.tables
                        WHERE name = 'nonsparse_nonulls_size')
                DROP TABLE nonsparse_nonulls_size;
GO
CREATE TABLE nonsparse_nonulls_size
(col1 int IDENTITY,
 col2 datetime,
 col3 char(10)
 );
GO
IF EXISTS (SELECT * FROM sys.tables
                        WHERE name = 'sparse_nulls_size')
                DROP TABLE sparse_nulls_size;
GO
CREATE TABLE sparse_nulls_size
(col1 int IDENTITY,
 col2 datetime SPARSE,
 col3 char(10) SPARSE
 );
GO
IF EXISTS (SELECT * FROM sys.tables
                        WHERE name = 'nonsparse_nulls_size')
                DROP TABLE nonsparse_nulls_size;
GO
CREATE TABLE nonsparse_nulls_size
(col1 int IDENTITY,
 col2 datetime,
 col3 char(10)
 );
GO
DECLARE @num int
SET @num = 1
WHILE @num < 100000
BEGIN
  INSERT INTO sparse_nonulls_size
        SELECT GETDATE(), 'my message';
  INSERT INTO nonsparse_nonulls_size
```

```
        SELECT GETDATE(), 'my message';
  INSERT INTO sparse_nulls_size
        SELECT NULL, NULL;
  INSERT INTO nonsparse_nulls_size
        SELECT NULL, NULL;
  SET @num = @num + 1;
END;
GO
```

Now look at the number of pages in each table. The following metadata query looks at the number of data pages in the *sys.allocation_units* view for each of the four tables:

```
SELECT object_name(object_id) as 'table with 100K rows', data_pages
FROM sys.allocation_units au
    JOIN sys.partitions p
       ON p.partition_id = au.container_id
WHERE object_name(object_id) LIKE '%sparse%size';
```

And here are my results:

```
table with 100K rows     data_pages
------------------------ ----------
sparse_nonulls_size      610
nonsparse_nonulls_size   402
sparse_nulls_size        169
nonsparse_nulls_size     402
```

Note that the smallest number of pages is required when the table has sparse columns that are NULL. If the table has no sparse columns, the space usage is the same whether the columns have NULLs or not because the data was defined as fixed length. This space requirement is more than twice as much as needed for the sparse columns with NULL. The worst case is if the columns have been defined as SPARSE but there are no NULL values.

Of course, the previous examples are edge cases, where *all* the data is either NULL or non-NULL, and it is all of fixed-length data types. So although we can say that SPARSE columns require more storage space for non-NULL values than is required for identical data that is not declared as SPARSE, the actual space savings depends on the data types and the percentage of rows that are NULL. Table 7-7 is reprinted from *SQL Server Books Online* and shows the space usage for each data type. The *NULL Percentage* column indicates what percent of the data must be NULL to achieve a net space savings of 40 percent. Table 7-7 shows the savings for various data types in SQL Server 2008.

Table 7-7: Storage Requirements for SPARSE Columns

Data Type	Storage Bytes When Not SPARSE	Storage Bytes When SPARSE and Not NULL	NULL Percentage
Fixed Length Data Types			
bit	0.125	4.125	98 percent
tinyint	1	5	86 percent
smallint	2	6	76 percent
int	4	8	64 percent
bigint	8	12	52 percent
real	4	8	64 percent
float	8	12	52 percent
smallmoney	4	8	64 percent
money	8	12	52 percent
smalldatetime	4	8	64 percent
datetime	8	12	52 percent
uniqueidentifier	16	20	43 percent
date	3	7	69 percent
Precision-Dependent-Length Data Types			
datetime2(0)	6	10	57 percent

datetime2(7)	8	12	52 percent
time(0)	3	7	69 percent
time(7)	5	9	60 percent
datetimetoffset(0)	8	12	52 percent
datetimetoffset (7)	10	14	49 percent
decimal/numeric(1,s)	5	9	60 percent
decimal/numeric (38,s)	17	21	42 percent
Data-Dependent-Length Data Types			
sql_variant	Varies		
varchar or *char*	4+avg. data	2+avg. data	60 percent
nvarchar or *nchar*	4+avg. data	2+avg. data	60 percent
varbinary or *binary*	4+avg. data	2+avg. data	60 percent
xml	4+avg. data	2+avg. data	60 percent
hierarchyId	4+avg. data	2+avg. data	60 percent

The general recommendation is that you should consider using SPARSE columns when you anticipate that it provides a space savings of at least 20 to 40 percent.

Data Compression

SQL Server 2008 provides the capability of data compression, a new feature that is available in Enterprise edition only. Compression can reduce the size of your tables by exploiting inefficiencies that exist in the actual data. These inefficiencies can be grouped into two general categories. The first category relates to storage of individual data values when they are stored in columns defined using the maximum possible size. For example, a table may need to define a *quantity* column as *int*, because occasionally you may be storing values larger than 32,767, which is the maximum *smallint* value. However, *int* columns always need four bytes, and if most of your *quantity* values are less than 100, they could be stored in *tinyint* columns, which need only 1 byte of storage. The Row Compression feature of SQL Server can compress individual columns of data to use only the minimum amount of space required.

The second type of inefficiency in the data storage occurs when the data on a page contains duplicate values or common prefixes across columns and rows. This inefficiency can be minimized by storing the repeating values only once and then referencing those values from other columns. The Page Compression feature of SQL Server can compress the data on a page by maintaining entries containing common prefixes or repeating values. Note that when you choose to apply page compression to a table or index, SQL Server always also applies row compression.

Vardecimal

SQL Server 2005 SP2 introduced a simple from of compression, which could be applied only to columns defined using the *decimal* data type. (Keep in mind that the data type numeric is completely equivalent to *decimal*, and anytime I mention *decimal*, it also means *numeric*.) In SQL Server 2005, the option has to be enabled at both the database level (using the procedure *sp_db_vardecimal_storage_format*) and at the table level (using the procedure *sp_tableoption*). In SQL Server 2008, all user databases are enabled automatically for the vardecimal storage format, so vardecimal must only be enabled for individual tables. Like data compression in SQL Server 2008, which we'll look at in detail in this section, the vardecimal storage format is available only in SQL Server Enterprise edition.

In SQL Server 2005, once both of these stored procedures have been run, *decimal* data in the tables enabled for vardecimal will be stored differently. Instead of being treated as fixed-length data, *decimal* columns are stored in the variable section of the row and use only the number of bytes required. (We looked at the difference between fixed-length data and variable-length data storage in Chapter 5.) In addition to all the partitions of the table using the vardecimal format for all *decimal* data, all indexes on the table use the vardecimal format automatically.

Decimal data values are defined with a precision of between 1 and 38, and depending on the defined precision, they use between 5 and 17 bytes. Fixed-length *decimal* data uses the same number of bytes for every row, even if the actual data could fit into far fewer bytes. When a table is not using the vardecimal storage format, every entry in the table consumes

the same number of bytes for each defined decimal column, even if the value of a row is 0, NULL, or some value that could be expressed in a smaller number of bytes, such as the number 3. When vardecimal storage format is enabled for a table, the *decimal* columns in each row use the minimum amount of space required to store the specified value. Of course, as we saw in Chapter 5, every variable-length column has 2 bytes of additional overhead associated with it, but when storing very small values in a column defined as *decimal* with a large precision, the space saving can more than make up for those additional 2 bytes. For vardecimal storage, both NULLs and zeros are stored as zero-length data and use only the 2 bytes of overhead.

Although SQL Server 2008 supports the vardecimal format, it is recommended that you use row compression when you want to reduce the storage space required by your data rows. Both the table option and the database option for enabling vardecimal storage have been deprecated.

Row Compression

You can think of row compression as an extension of the vardecimal storage format. There can be many situations in which SQL Server uses more space than is necessary to store data values, and without SQL Server 2008 Enterprise Edition, the only control you have is to use a variable-length data type. Any fixed-length data types always uses the same amount of space in every row of a table, even if space is wasted. For example, you may declare a column as type *int* because occasionally you may need to store values greater than 32,000. An *int* needs 4 bytes of space, no matter what number is stored, even if the column is NULL. Only character and binary data can be stored in variable-length columns (and, of course, decimal, once that option is enabled). Row compression allows integer values to use only the amount of storage space required, with the minimum being 1 byte. A value of 100 needs only a single byte for storage, and a value of 1,000 needs 2 bytes. There is an optimization that allows zero and NULL to use no storage space for the data itself. We'll see the details about this later in this section.

Enabling Row Compression

Compression can be enabled when creating a table or index, or using the *ALTER TABLE* or *ALTER INDEX* command. In addition, if the table or index is partitioned, you can choose to just compress a subset of the partitions. (We'll look at partitioning later in this chapter.)

The following script creates two copies of the *dbo.Employees* table in the *AdventureWorks2008* database. When storing row-compressed data, SQL Server treats values that can be stored in 8 bytes or fewer (that is, short columns) differently than it stores data that needs more than 8 bytes (long columns). For this reason, my script updates one of the rows in the new tables so that none of the columns contains more than 8 bytes. The *Employees_rowcompressed* table is then enabled for row compression, and the *Employees_uncompressed* table is left uncompressed. A metadata query examining pages allocated to each table is executed against each of the tables so that you can compare the sizes before and after row compression:

```
USE AdventureWorks2008;
GO
IF EXISTS (SELECT * FROM sys.tables
        WHERE name = 'Employees_uncompressed')
           DROP TABLE Employees_uncompressed;
GO
SELECT e.BusinessEntityID, NationalIDNumber, JobTitle,
        BirthDate, MaritalStatus, VacationHours,
        FirstName, LastName
  INTO Employees_uncompressed

  FROM HumanResources.Employee e
   JOIN Person.Person p
        ON e.BusinessEntityID = p.BusinessEntityID;
GO
UPDATE Employees_uncompressed
SET NationalIDNumber = '1111',
        JobTitle = 'Boss',
        LastName = 'Gato'
WHERE FirstName = 'Ken'
AND LastName = 'Sánchez';
GO
ALTER TABLE dbo.Employees_uncompressed
   ADD CONSTRAINT EmployeeUn_ID
       PRIMARY KEY (BusinessEntityID);
```

```
GO
SELECT OBJECT_NAME(object_id) as name,
        rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
        ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_uncompressed');
IF EXISTS (SELECT * FROM sys.tables
        WHERE name = 'Employees_rowcompressed')
            DROP TABLE Employees_rowcompressed;
GO
SELECT BusinessEntityID, NationalIDNumber, JobTitle,
        BirthDate, MaritalStatus, VacationHours,
        FirstName, LastName
  INTO Employees_rowcompressed
  FROM dbo.Employees_uncompressed
GO
ALTER TABLE dbo.Employees_rowcompressed
   ADD CONSTRAINT EmployeeR_ID
       PRIMARY KEY (BusinessEntityID);
GO
ALTER TABLE dbo.Employees_rowcompressed
REBUILD WITH (DATA_COMPRESSION = ROW);
GO
SELECT OBJECT_NAME(object_id) as name,
        rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
        ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_rowcompressed');
GO
```

I'll refer to the *dbo.Employees_rowcompressed* table again later in this section, or you can examine it on your own as I discuss the details of compressed row storage.

Now we'll start looking at the details of row compression, but keep these points in mind:

- Row compression is available only in SQL Server 2008 Enterprise and Developer editions.
- Row compression does not change the maximum row size of a table or index.
- Row compression cannot be enabled on a table with any columns defined as SPARSE.
- If a table or index has been partitioned, row compression can be enabled on all the partitions or on a subset of the partitions.

New Row Format

In Chapter 5, we looked at the format for storing rows that has been used since SQL Server 7.0 and is still used in SQL Server 2008 if you have not enabled compression. That format is referred to as *FixedVar* format because it has a fixed-length data section separate from a variable-length data section. A completely new row format is introduced in SQL Server 2008 for storing compressed rows, and this format is referred to as *CD* format. CD stands for "column descriptor," and that term refers to the fact that every column has description information contained in the row itself. You might want to re-examine Figure 5-10 in Chapter 5 as a reminder of what the FixedVar format looks like, and compare it to the new CD format. Figure 7-7 shows an abstraction of the CD format. It's difficult to be as specific as Figure 5-10 is because except for the Header, the number of bytes in each region is completely dependent on the data in the row.

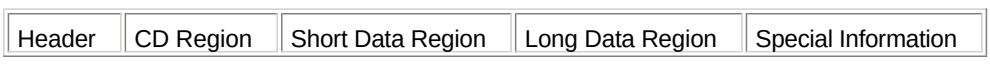

Figure 7-7: General structure of a CD record

I'll describe each of these sections in detail.

Header The row header is always a single byte and roughly corresponds to what I called Status Bits A in Chapter 5. The bits have the following meanings:

Bit 0 Indicates the type of record; it's 1 for the new CD record format.

Bit 1 Indicates that the row contains versioning information.

Bits 2 through 4 Taken as a three-bit value, these bits indicate what kind of information is stored in the row. The possible values are the following:

- 000—primary record
- 001—ghost empty record
- 010—forwarding record
- 011—ghost data record
- 100—forwarded record
- 101—ghost forwarded record
- 110—index record
- 111—ghost index record

Bit 5 Indicates that the row contains a long data region (with values greater than 8 bytes in length).

Bit 6 - 7 Not used in SQL Server 2008.

The CD Region The CD region is composed of two parts. The first part is either 1 or 2 bytes, indicating the number of short columns. If the most significant bit of the first byte is set to 0, then it is a 1-byte field with a maximum value of 127. If there are more than 127 columns, then the most significant bit is 1, and SQL Server uses 2 bytes to represent the number of columns, which can be up to 32,767.

Following the 1 or 2 bytes for the number of columns is the CD array. The CD array uses four bits for each column in the table, to represent information about the length of the column. Four bits can have 16 different possible values, but in SQL Server 2008, only 13 of them are used:

- 0 (0x0) indicates that the corresponding column is NULL
- 1 (0x1) indicates that the corresponding column is a 0-byte short value.
- 2 (0x2) indicates that the corresponding column is a 1-byte short value.
- 3 (0x3) indicates that the corresponding column is a 2-byte short value.
- 4 (0x4) indicates that the corresponding column is a 3-byte short value.
- 5 (0x5) indicates that the corresponding column is a 4-byte short value.
- 6 (0x6) indicates that the corresponding column is a 5-byte short value.
- 7 (0x7) indicates that the corresponding column is a 6-byte short value.
- 8 (0x8) indicates that the corresponding column is a 7-byte short value.
- 9 (0x9) indicates that the corresponding column is an 8-byte short value.
- 10 (0xa) indicates that the corresponding column is long data value and uses no space in the short data region.
- 11 (0xb) is used for columns of type bit with the value of 1. The corresponding column takes no space in the short data region.
- 12 (0xc) indicates that the corresponding column is a 1-byte symbol, representing a value in the page dictionary. (I'll talk about the dictionary in the section entitled "Page Compression," later in this chapter).

The Short Data Region The short data region doesn't need to store the length of each of the short data values because that information is available in the CD region. However, if there are hundreds of columns in the table, it can be expensive to access the last columns. To minimize this cost, columns are grouped into clusters of 30 columns each and at the beginning of the short data region, there is an area called the short data cluster array. Each array entry is a single-byte integer and

indicates the sum of the sizes of all the data in the previous cluster in the short data region, so that the value is basically a pointer to the first column of the cluster. The first cluster of short data starts right after the cluster array, so no cluster offset is needed for it. There may not be 30 data columns in a cluster, however, because only columns with a length less than or equal to 8 bytes are stored in the short data region.

As an example, consider a row with 64 columns, and columns 5, 10, 15, 20, 25, 30, 40, 50, and 60 are long data, and the others are short. The CD region contains the following:

- A single byte containing 64, the number of columns, in the CD region.
- A CD array of 4 * 64 bits, or 32 bytes, containing information about the length of each column. There are 55 entries with values indicating an actual data length for the short data, and 8 entries of 0xa, indicating long data.
- The short data region contains the following.
  - A short data cluster offset array containing the two values, each containing the length of a short data cluster. In this example, the first cluster, which is all the short data in the first 30 columns, has a length of 92, so the 92 in the offset array indicates that the second cluster starts 92 bytes after the first. The number of clusters can be calculated as (Number of columns – 1) /30. The maximum value for any entry in the cluster array is 240, if all 30 columns were short data of 8 bytes in length.
  - All the short data values.

Figure 7-8 illustrates the CD region and the short data region with sample data for the row described previously. The CD array is shown in its entirety, with a symbol indicating the length of each of the 64 values. So the array can fit on a page of this book, the actual data values are not shown. The first cluster has 24 values in the short data region (6 are long values), the second cluster has 27 (3 are long) and the third cluster has the remaining 4 columns (all short). I'll discuss the storage of the long values next.

CD Region		**Short Data Region**				
Number of columns	CD array --64 4-bit values ('a' indicates long column)	Length of short data in each 30-column cluster (N–1)/30 values		Three clusters of actual data		
N = 64	3285a4358a6543a3456a6666a5463a254372644a745269277a463495736a5433	92	106	24 values	27 values	4 values

Figure 7-8: The CD region and short data region in a CD record

To locate the entry for a short column value in the short data region, the short data cluster array is first examined to determine the start address of the containing cluster for the column in the short data region.

The Long Data Region Any data in the row longer than 8 bytes is stored in the long data region. This includes complex columns, which do not contain actual data but rather contain information necessary to locate data stored off the row. This can include large object data and row overflow data pointers. Unlike short data, where the length can be stored simply in the CD array, long data needs an actual offset value to allow SQL Server to determine the location of each value. This offset array looks very similar to the offset array I talked about in Chapter 5 for the FixedVar records.

The long data region is composed of three parts: an offset array, a long data cluster array, and the long data.

The offset array is composed of the following:

- A 1-byte header. In SQL Server 2008, only the first two bits are used. Bit 0 indicates if the long data region contains any 2-byte offset values, and in SQL Server 2008, this value is always 1, as all offsets are always 2 bytes. Bit 1 indicates if the long data region contains any complex columns.
- A 2-byte value indicating the number of offsets to follow. The most significant bit in the first byte of the offset value is used to indicate whether the corresponding entry in the long data region is a complex column or not. The rest of the bits/bytes in the array entry store the ending offset value for the corresponding entry in the long data region.

The long data cluster array is similar to the cluster array for the short data and is used to limit the cost of finding columns near the end of a long list of columns. It has one entry for each 30-column cluster (except the last one). Because we already have the offset of each long data column stored in the offset array, the cluster array just needs to keep track of

how many of the long data values are in each cluster. Each value is a one-byte integer representing the number of long data columns in that cluster. Just as for the short data cluster, the number of entries in the cluster array can be computed as (Number of columns in the table – 1)/ 30.

Figure 7-9 illustrates the long data region for the row described previously, with 64 columns, 9 of which are long. I have not actually included values for the offsets due to space considerations. The long data cluster array has two entries indicating that 6 of the values are in the first cluster and 2 are in the second. The remaining values are in the last cluster.

Offset Array			**Long Data Cluster Array**		**Long Data**								
Header	# of entries	Offset entries	Number of entries in each 30-column cluster (N? 1)/30 values		Long data 1	Long data 2	Long data 3	Long data 4	Long data 5	Long data 6	Long data 7	Long data 8	Long data 9
01	09		06	02									

Figure 7-9: The long data region of a CD record

Special Information The end of the row contains three optional pieces of information. The existence of any or all of this information is indicated by bits in the 1-byte header at the very beginning of the row. The three special areas are the following:

- **Forwarding Pointer** This value is used when a heap contains a forwarding stub that points to a new location to which the original row has been moved. Forwarding pointers were discussed in Chapter 5. The forwarding pointer contains three header bytes and an 8-byte Row ID.
- **Back Pointer** This value is used in a row that has been forwarded to indicate the original location of the row. It is stored as an 8-byte Row ID.
- **Versioning Info** When a row is modified under Snapshot isolation, SQL Server adds 14 bytes of versioning information to the row. Row versioning and Snapshot isolation is discussed in Chapter 10.

Now let's look at the actual bytes in two of the rows in the *dbo.Employees_rowcompressed* table created previously. The *DBCC PAGE* command has been enhanced to give additional information about compressed rows and pages. In particular, before the bytes for the row are shown, *DBCC PAGE* will display the CD array. For the first row returned on the first page in the *dbo.Employees_rowcompressed* table, all the columns contain short data. The row has the following data values:

BusinessEntityID	NationalIDNumber	JobTitle	BirthDate	MaritalStatus	VacationHours	FirstName	LastName
1	1111	Boss	1959-03-02	S	99	Ken	Gato

For short data, the CD array contains the actual length of each of the columns, and we can see the following information for the first row in the *DBCC PAGE* output:

```
CD array entry = Column 1 (cluster 0, CD array offset 0): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 2 (cluster 0, CD array offset 0): 0x09 (EIGHT_BYTE_SHORT)
CD array entry = Column 3 (cluster 0, CD array offset 1): 0x09 (EIGHT_BYTE_SHORT)
CD array entry = Column 4 (cluster 0, CD array offset 1): 0x04 (THREE_BYTE_SHORT)
CD array entry = Column 5 (cluster 0, CD array offset 2): 0x03 (TWO_BYTE_SHORT)
CD array entry = Column 6 (cluster 0, CD array offset 2): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 7 (cluster 0, CD array offset 3): 0x07 (SIX_BYTE_SHORT)
CD array entry = Column 8 (cluster 0, CD array offset 3): 0x09 (EIGHT_BYTE_SHORT)
```

So the first column has a CD code of 0x02, which indicates a 1-byte value, and, as we see in the data row, is the integer 1. The second column contains an 8-byte value and is the Unicode string 1111. I'll leave it to you to inspect the code for the remaining columns.

Figure 7-10 shows the *DBCC PAGE* output for the row contents, and I have indicated the meaning of the different bytes.

```
Record Memory Dump for first row from DBCC PAGE:

01089249 23978131 00310031 00310042 †..'I#—.1.1.1.1.B
```

```
006f0073 007300c4 e90a5300 e34b0065 †.o.s.s.Äé.S.ãK.e
006e0047 00610074 006f00†††††††††††††.n.G.a.t.o.
```

Row expansion with byte swapping:

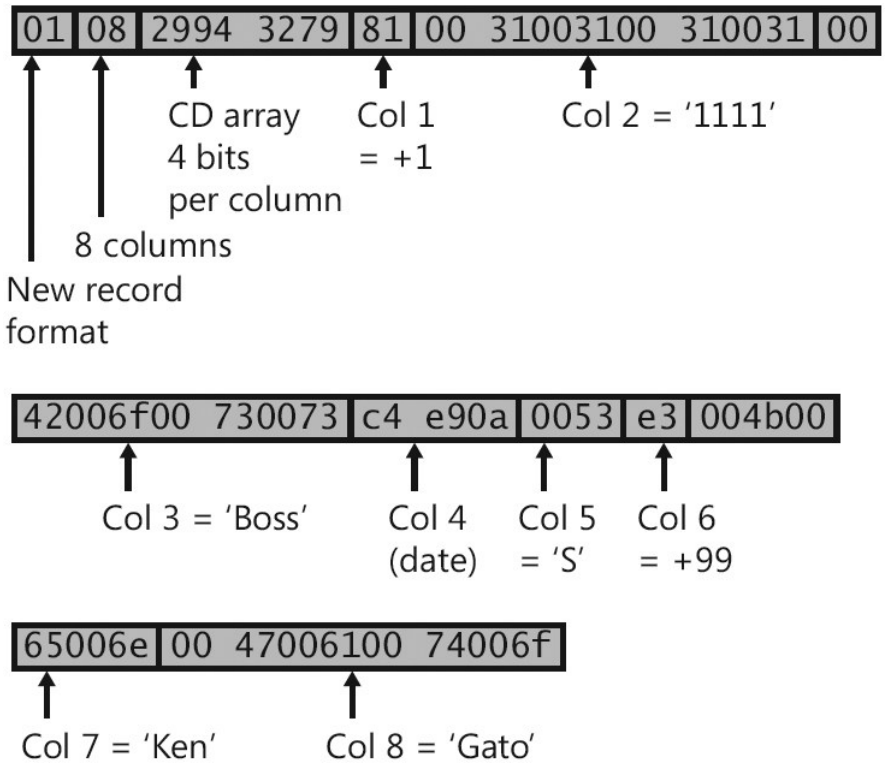

Figure 7-10: A compressed row with eight short data columns

The second row returned on the first page has a few long columns in the following data values:

Bus...	NationalIDNumber	JobTitle	BirthDate	Marital...	Vacation...	FirstName	LastName
2	245797967	Vice President of Engineering	1961-09-01	S	1	Terri	Duffy

The CD array for this row looks like the following:

```
CD array entry = Column 1 (cluster 0, CD array offset 0): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 2 (cluster 0, CD array offset 0): 0x0a (LONG)
CD array entry = Column 3 (cluster 0, CD array offset 1): 0x0a (LONG)
CD array entry = Column 4 (cluster 0, CD array offset 1): 0x04 (THREE_BYTE_SHORT)
CD array entry = Column 5 (cluster 0, CD array offset 2): 0x03 (TWO_BYTE_SHORT)
CD array entry = Column 6 (cluster 0, CD array offset 2): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 7 (cluster 0, CD array offset 3): 0x0a (LONG)
CD array entry = Column 8 (cluster 0, CD array offset 3): 0x0a (LONG)
```

Note that four of the eight columns are long data values.

Figure 7-11 shows the bytes that *DBCC PAGE* returns for this second data row.

```
RecordMemoryDump??
6294C08B:   2108a24a 23aa8256 ed0a5300 81010400 †!.¢J#ª,Ví.S.....???????
6294C09B:   12004c00 56006000 32003400 35003700 †..L.V.`.2.4.5.7.???????
6294C0AB:   39003700 39003600 37005600 69006300 †9.7.9.6.7.V.i.c.???????
6294C0BB:   65002000 50007200 65007300 69006400 †e..P.r.e.s.i.d.????????
6294C0CB:   65006e00 74002000 6f006600 20004500 †e.n.t..o.f..E.?????????
6294C0DB:   6e006700 69006e00 65006500 72006900 †n.g.i.n.e.e.r.i.???????
6294C0EB:   6e006700 54006500 72007200 69004400 †n.g.T.e.r.r.i.D.???????
```

Figure 7-11: A compressed row with four short data columns and four long

I have highlighted the bytes in the long data region. Here are some things to notice in the first part of the row, before the long data region:

- The first byte in the row is 0x21, indicating that not only is this row in the new CD record format, but also that the row contains a long data region.
- The second byte indicates there are eight columns in the table, just as for the first row.
- The following 8 bytes for the CD array has four values of a, which indicate a long value not included in the short data region.
- The short data values are listed in order after the CD array and are as follows:
  - The *BusinessEntityID* is 1 byte, with the value 0x82, or +2
  - The *Birthdate* is 3 bytes
  - The *MaritalStatus* is 1 byte, with the value 0x0053, or 'S'
  - The *VacationHours* is 1 byte, with the value 0x81, or +1

The Long Data Region Offset Array is 10 bytes long, with the following interpretation:

- The first byte is 0x01, which indicates that the row-offset positions are 2 bytes long.
- The second byte is 0x04, which indicates there are four columns in the long data region.
- The next 8 bytes are the 2-byte offsets for each of the four values. Note that the offset refers to position the column ends with the Long Data area itself.
  - The first 2-byte offset is 0x0012, or 18. This indicates that the first long value is 18 bytes long. (It is Unicode string of 9 characters, 245797967, which would need 18 bytes.)
  - The second 2-byte offset is 0x004c, or 76, which indicates that the second long value ends 58 bytes after the first. The second value is *Vice President of Engineering*, which is a 29-byte Unicode string.
  - The third 2-byte offset is 0x0056, or 86, which indicates the third value, *Terri*, is 10 bytes long.
  - The fourth 2-byte offset is 0x0060, or 96, which indicates the fourth value, *Duffy*, is 10 bytes long.

Because there are fewer than 30 columns, there is no Long Data Cluster Array, and the data values are stored immediately after the Long Data Region Offset Array.

Due to space constraints, I won't show you the details of a row with multiple column clusters (that is, more than 30 columns), but hopefully you have enough information to start exploring such rows on your own.

Page Compression

In addition to storing rows in a compressed format to minimize the space required, SQL Server 2008 can compress whole pages by isolating and reusing repeating patterns of bytes on the page.

Unlike row compression, page compression is applied only once a page is full, and if SQL Server determines that compressing the page saves a meaningful amount of space. (I'll elaborate on what that amount is later in this section.) You should keep the following points in mind when planning for page compression:

- Page compression is available only in the SQL Server 2008 Enterprise and Developer editions.
- Page compression always includes row compression. (That is, if you enable page compression for a table, row compression is automatically enabled.)
- When compressing a B-tree, only the leaf level can be page-compressed. For performance reasons, the node levels are left uncompressed.

- If a table or index has been partitioned, page compression can be enabled on all the partitions, or a subset of the partitions.
- Page compression is not maintained as new rows are added. The page compression algorithm must be reapplied to an entire page and is done only when SQL Server determines that doing so brings benefits. Again, we'll see details about this later in this section.

The code here makes another copy of the *dbo.Employees* table and applies page compression to it. It then captures the page location and linkage information from *DBCC IND*, for the three tables: *dbo.Employees_uncompressed, dbo.Employees_rowcompressed,* and *dbo. Employees_pagecompressed*. The code then uses the captured information to report on the number of data pages in each of the three tables:

```
IF EXISTS (SELECT * FROM sys.tables
        WHERE name = 'Employees_pagecompressed')
                DROP TABLE Employees_pagecompressed;
GO
SELECT BusinessEntityID, NationalIDNumber, JobTitle,
        BirthDate, MaritalStatus, VacationHours,
        FirstName, LastName
  INTO Employees_pagecompressed
  FROM dbo.Employees_uncompressed
GO
ALTER TABLE dbo.Employees_pagecompressed
   ADD CONSTRAINT EmployeeP_ID
       PRIMARY KEY (BusinessEntityID);
GO
ALTER TABLE dbo.Employees_pagecompressed
REBUILD WITH (DATA_COMPRESSION = PAGE);
GO
SELECT OBJECT_NAME(object_id) as name,
                rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
                ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_pagecompressed');
GO

TRUNCATE TABLE sp_table_pages;
GO
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_pagecompressed, -1)');
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_rowcompressed, -1)');
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_uncompressed, -1)');
GO
SELECT OBJECT_NAME(ObjectID), count(*) as NumPages
FROM sp_tablepages
WHERE pagetype = 1
GROUP BY OBJECT_NAME(ObjectID);
GO
```

If you run this script, note in the output that row compression did not reduce the size of this small table, but page compression shrank the table from five data pages to three.

SQL Server can perform two different operations to try to compress a page using common values: *column prefix compression* and *dictionary compression*.

Column Prefix Compression

As the name implies, column prefix compression works on columns of data in the table being compressed, but it looks only at the column values on a single page. For each column, SQL Server chooses a common prefix that can be used to reduce the storage space required for values in that column. The longest value in the column that contains that prefix is chosen as the *anchor value*. Each column is then stored not as the actual data value but as a delta from the anchor value. An example is probably needed to clarify this. Suppose that we have the following character values in a column of the table to be page-compressed:

```
DEEM
```

```
DEE
FFF
DEED
DEE
DAN
```

SQL Server might note that DEE is a useful common prefix, so *DEED* is chosen as the anchor value. Each column would be stored as the difference between its value and the anchor value. This difference is stored as a two-part value: the number of characters from the anchor to use, and the additional characters to append. So *DEEM* is stored as <3><M>, meaning the value uses the first three characters from the common prefix and appends a single character, *M,* to it. *DEED* is stored as an empty string (but not null) to indicate it matched the prefix exactly. *DEE* is stored as <3>, with the second part empty, because there are no additional characters to be appended. The list of column values is replaced by the values shown here:

```
DEEM -> <3><M>
DEE -> <3><>
FFF -> <><FFF>
DEED -> <><>
DEE -> <3><>
DAN -> <1><AN>
```

Keep in mind that the compressed row is stored in the CD record format, so the CD array value has a special encoding to indicate the value is actually NULL. If the replacement value is <><>, and the encoding doesn't indicate NULL, then the value matches the prefix exactly.

SQL Server applies the prefix detection and value replacement algorithm to every column and creates a new row called an *anchor record* to store the anchor values for each column. If no useful prefix can be found, the value in the anchor record is NULL, and then all the values in the column are stored just as they are.

Figure 7-12 shows an image of six rows in a table prior to page compression, and then shows the six rows after the anchor record has been created and the substitutions have been made for the actual data values.

Original Data		
ABCD	DEEM	ABC
ABD	DEE	DEE
ABC	FFF	GHI
AAN	DEED	HHH
NULL	DEE	KLM
ADE	DAN	NOP
Data After Column Prefix Compression		
Anchor Record		
ABCD	DEED	NULL
<><>	<3><M>	ABC
<2><D>	<3><>	DEE
<3><>	<><FFF>	GHI
<1><AN>	<><>	HHH
NULL	<3><>	KLM
<1><DE>	<1><AN>	NOP

Figure 7-12: Before and after column prefix compression

Dictionary Compression

After prefix compression has been applied to every column individually, the second phase of page compression looks at all values on the page to find duplicates in any column of any row, even if they have been encoded to reflect prefix usage. You can see in the bottom part of Figure 7-11 that two of the values occur multiple times: <3><> occurs three times and <1><AN> occurs twice. The process of detecting duplicate values is datatype-agnostic, so values in completely different

columns could be the same in their binary representation. For example, a 1-byte character is represented in hex as 0x54, and it would be seen as a duplicate of the 1-byte integer 84, which is also represented in hex as 0x54. The dictionary is stored as a set of symbols, each of which corresponds to a duplicated value on the data page. Once the symbols and data values have been determined, each occurrence of one of the duplicated values is replaced by the symbol. SQL Server recognizes that the value actually stored in the column is a symbol and not a data value by examining the encoding in the CD array. Values which have been replaced by symbols have a CD array value of 0xc. Figure 7-13 shows the data from Figure 7-12 after replacing the five values with symbols.

Dictionary of Symbols: **[S1] = <1><AN> [S2] = <3><>**		
<><>	<3><M>	ABC
<2><D>	[S2]	DEE
[S2]	<><FFF>	GHI
[S1]	<><>	HHH
NULL	[S2]	KLM
<1><DE>	[S1]	NOP

Figure 7-13: A page compressed with dictionary compression

Not every page in a compressed table has both an anchor record for prefixes and a dictionary. If there are no useful prefix values, the page might just have a dictionary. If no values repeat often enough that replacing them with symbols saves space, the page might just have an anchor record. And, of course, there may be pages that have neither an anchor record nor a dictionary, if there are no patterns at all in the data on the page.

Physical Storage

There is only one main structural change to a page when it is page-compressed. SQL Server adds a hidden row right after the page header (at byte offset 96, or 0x60) called the *compression information (CI) record*. The structure of the CI record is shown in Figure 7-14.

Header	PageModCount	Offsets	Anchor Record	Dictionary

Figure 7-14: Structure of a CI record

The CI record does not have an entry in the slot array for the page, but it is always at the same location. In addition, a bit in the page header indicates that the page is page-compressed, so SQL Server looks for the CI record. If you use *DBCC PAGE* to dump a page, the page header information contains a value called *m_typeFlagBits*. If this value is 0x80, the page is compressed.

The following script can be run if you have already created the table called *sp_tablepages*, described earlier in this chapter. This script captures the *DBCC IND* information from the tables created in this section: *Employees_uncompressed*, *Employees_rowcompressed,* and *Employees_pagecompressed.* The script then displays the first data page number for each of the tables. You can use this information to examine the page with *DBCC PAGE.* Note that only the page for *Employees_pagecompressed* has the *m_typeFlagBits* value set to 0x80:

```
USE AdventureWorks2008;
GO
TRUNCATE TABLE sp_tablepages;
GO
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_pagecompressed, -1)');
GO
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_rowcompressed, -1)');
GO
INSERT INTO sp_tablepages
   EXEC ('DBCC IND(AdventureWorks2008, Employees_uncompressed, -1)');
GO
SELECT OBJECT_NAME(ObjectID), PageFID, PagePID
FROM sp_tablepages
```

```
WHERE pagetype = 1
  AND PrevPagePID = 0;
GO
DBCC TRACEON(3604);
GO
```

Using *DBCC PAGE* to look at a page-compressed page does provide information about the contents of the CI record, and we'll look at some of that information after we examine what each of the sections means, which is discussed next.

Header The header is a 1-byte value keeping track of information about the CI. Bit 0 indicates the version, which in SQL Server 2008 is always 0. Bit 1 indicates whether the CI has an anchor record, and bit 2 indicates whether the CI has a dictionary. The rest of the bits are unused.

PageModCount The *PageModCount* value keeps track of the changes to this particular page and is used when determining whether the compression on the page should be reevaluated, and a new CI record built. I'll talk more about how this value is used in the next section, when I discuss page compression analysis.

Offsets The offsets contain values to help SQL Server find the dictionary. It contains a value indicating the page offset for the end of the anchor record and a value indicating the page offset for the end of the CI record itself.

Anchor Record The anchor record looks exactly like a regular CD record on the page, including the record header, the CD array, and both a short data area and a long data area. The values stored in the data area are the common prefix values for each column, some of which might be NULL.

Dictionary The dictionary area is composed of three sections. The first is a 2-byte field containing a numeric value representing the number of entries in the dictionary. The second section is an offset array of 2-byte entries, indicating the end offset of each dictionary entry relative to the start of the dictionary data section. The third section contains the actual dictionary data entries.

Remember that each dictionary entry is a byte string that is replaced in the regular data rows by a symbol. The symbol is simply an integer value from 0 to N. In addition, remember that the byte strings are datatype-independent; that is, they are just bytes. After SQL Server determines what recurring values are stored in the dictionary, it sorts the list first by data length, then by data value, and then assigns the symbols in order. So suppose that the values to be stored in the dictionary are these:

```
0x 53 51 4C
0x FF F8
0x DA 15 43 77 64
0x 34 F3 B6 22 CD
0x 12 34 56
```

Table 7-8 shows the sorted dictionary, along with the length and symbol for each entry.

Table 7-8: Values in a Page Compression Dictionary

Value	Length	Symbol
0x FF F8	2 bytes	0
0x 12 34 56	3 bytes	1
0x 53 51 4C	3 bytes	2
0x 34 F3 B6 22 CD	4 bytes	3
0x DA 15 43 77 64	4 bytes	4

The dictionary area would then look like Figure 7-15.

Header	Offsets	Dictionary
5	02 00	0x FF F8
	05 00	0x 12 34 56
	08 00	0x 53 51 4C
	0D 00	0x 34 F3 B6 22 CD

	12 00	0x DA 15 43 77 64

Figure 7-15: The dictionary area in a Compression Information Record

Note that the dictionary never actually stores the symbol values. They are stored only in the data records that need to use the dictionary. Because they are simply integers, they can be used as an index into the offset list to find the appropriate dictionary replacement value. For example, if a row on the page contains the dictionary symbol [2], SQL Server looks in the offset list for the third entry, which in Figure 7-14 ends at offset 0800 from the start of the dictionary. SQL Server then finds the value that ends at that byte, which is 0x 53 51 4C. If this byte string was stored in a *char* or *varchar* column (that is, a single-byte character string), it would correspond to the character string *SQL*.

I illustrated earlier in this chapter that the *DBCC PAGE* output shows you the CD array for compressed rows. For compressed pages, *DBCC PAGE* shows the CI record and details about the anchor record within it. In addition, with format 3, *DBCC PAGE* shows details about the dictionary entries. When I captured the *DBCC PAGE* in format 3 for the first page of my *Employees_pagecompressed* table and copied it to a Microsoft Office Word document, it needed 261 pages. Needless to say, I will not show you all that output. Even when I just copied the CI record information, it took 7 pages, which is still too much to show in this book. I'll leave it to you to explore the output of *DBCC PAGE* for the tables with compressed pages.

Page Compression Analysis

In this section, I discuss some of the details regarding how SQL Server determines whether to compress a page or not and what values it uses for the anchor record and the dictionary. Row compression is always performed when requested, but page compression depends on the amount of space that can be saved. However, the actual work of compressing the rows has to wait until page compression has been performed. Because both types of page compression, prefix substitution and dictionary symbol substitution, replace the actual data values with encodings, the row cannot be compressed until SQL Server determines what encodings are going to replace the actual data.

When page compression is first enabled for a table or partition, SQL Server goes through every full page to determine the possible space savings. (Any pages that are not full are not considered for compression.) This compression analysis actually creates the anchor record, modifies all the columns to reflect the anchor values, and generates the dictionary. Then it compresses each row. If the new compressed page can hold at least five more rows, or 25 percent more rows than the current page (whichever is larger), then the compressed page replaces the uncompressed page. If compressing the page does not result in this much savings, the compressed page is discarded.

When determining what values to use for the anchor record on a compressed page, SQL Server needs to look at every byte in every row, one column at a time. As it scans the column, it also keeps track of possible dictionary entries (which can be used in multiple columns). The anchor record values can be determined for each column in a single pass; that is, by the time all the bytes in all the rows for the first column are examined once, SQL Server has determined the anchor record value for that column or it has determined that no anchor record value will save sufficient space.

As SQL Server examines each column, it collects a list of possible dictionary entries. As we've discussed, the dictionary contains values that occur enough times on the page so that replacing them with a symbol is cost-effective in terms of space. For each possible dictionary entry, SQL Server keeps track of the value, its size, and the count of occurrences. If (size_of_data_value –1) * (count–1) –2 is greater than zero, it means the dictionary replacement saves space, and the value is considered eligible for the dictionary. In general, SQL Server tries to keep no more than 300 entries in the dictionary, so if more dictionary entries are possible, they are sorted by count during the analysis and only the most frequently occurring values are used in the dictionary.

Rebuilding the CI Record

If a table is enabled for either page or row compression, new rows are always compressed before they are inserted into the table. However, the CI record containing the anchor record and the dictionary is rebuilt on an all-or-nothing basis; that is, SQL Server does not just add some new entry to the dictionary when new rows are inserted. SQL Server evaluates whether to rebuild the CI record when the page has been changed a sufficient number of times. It keeps track of changes to each page in the *PageModCount* field of the CI record, and that value is updated every time a row is inserted, updated, or deleted. If a full page is encountered during a data modification operation, SQL Server examines the *PageModCount* value. If the *PageModCount* value is greater than 25, or if the value *PageModCount/<number of rows on the page>* is greater than 25 percent, SQL Server applies the compression analysis as it does when it first compresses a page. Only if it is determined that recompressing the page makes room for at least five more rows (or 25 percent more rows than the current page) does the new compressed page replace the old page.

There are some important differences between compression of pages in a B-tree and compression of pages in a heap.

Compression of B-tree Pages For B-trees, only the leaf level is page-compressed. When inserting a new row into a B-tree, if the compressed row fits on the page, it is inserted, and nothing more is done. If it doesn't fit, SQL Server tries to recompress the page, according to the conditions described in the preceding section. If the recompression succeeded, it means that the CI record changed, so the new row must be recompressed and then SQL Server tries to insert it into the page. Again, if it fits, it is simply inserted. If the new compressed row doesn't fit on the page, even after possibly recompressing the page, the page needs to be split. When splitting a compressed page, the CI record is copied to a new page, exactly as is, except that the *PageModCount* value is set to 25. This means that the first time the page gets full, it gets a full analysis to determine if it should be recompressed. B-tree pages are also checked for possible recompression during index rebuilds (either online or offline) and during shrink operations.

Compression of Heap Pages Pages in a heap are checked for possible compression only during rebuild and shrink operations. (Note that SQL Server 2008 provides an option to rebuild a table and specify a compression level for just this reason.) Also, if you drop a clustered index on a table so that it becomes a heap, SQL Server runs compression analysis on any full pages. To make sure that the *RowID* values stay the same, heaps are not recompressed during normal data modification operations. Although the *PageModCount* value is maintained, SQL Server never tries to recompress a page based on the *PageModCount* value.

Compression Metadata

There is not an enormous amount of metadata information relating to data compression. The catalog view *sys.partitions* has a *data_compression* column and a *data_compression_desc* column. The *data_compression* column has possible values of 0, 1, and 2, corresponding to *data_compression_desc* values of NONE, ROW, and PAGE. Keep in mind that although row compression is always performed, page compression is not. Even if *sys.partitions* indicates that a table or partition is page-compressed, that just means that page compression is enabled. Each page is analyzed individually, and if a page is not full, or if compression would not save enough space, the page is not compressed.

You can also inspect the dynamic management function *sys.dm_db_index_operational_stats*. This table-valued function returns the following compression-related columns:

- **page_compression_attempt_count** The number of pages that were evaluated for PAGE-level compression for specific partitions of a table, index, or indexed view. Includes pages that were not compressed because significant savings could not be achieved.
- **page_compression_success_count** The number of data pages that were compressed by using PAGE compression for specific partitions of a table, index, or indexed view.

SQL Server 2008 also provides a stored procedure called *sp_estimate_data_compression_savings,* which can give you some idea of whether compression provides a large space savings or not.

This procedure samples up to 5,000 pages of the table and creates an equivalent table with the sampled pages in *tempdb*. Using this temporary table, SQL Server can estimate the new table size for the requested compression state (NONE, ROW, or PAGE). Compression can be evaluated for whole tables or parts of tables. This includes heaps, clustered indexes, nonclustered indexes, indexed views, and table and index partitions.

Keep in mind that the result is only an estimate and your actual savings can vary widely based on the fillfactor and the size of the rows. If the procedure indicates that you can reduce your row size by 40 percent, you might not actually get a 40-percent space savings for the whole table. For example, if you have a row that is 8,000 bytes long and you reduce its size by 40 percent, you still can fit only one row on a data page and your table still needs the same number of pages.

You may get results from running *sp_estimate_data_compression_savings* that indicate that the table will grow. This can happen when many rows in the table use almost the whole maximum size of the data types, and the addition of the overhead needed for the compression information is more than the savings from compression.

If the table is already compressed, you can use this procedure to estimate the size of the table (or index) if it were to be uncompressed.

Performance Issues

The main motivation for compressing your data is to save space with extremely large tables, such as data warehouse fact tables. A second goal is to increase performance when scanning a table for reporting purposes, because far fewer pages

need to be read. You need to keep in mind that compression comes at a cost: there is a tradeoff between the space savings and the extra CPU overhead to compress the data for storage and then uncompress the data when it needs to be used. On a CPU-bound system, you may find that compressing your data can actually slow down your system considerably.

Page compression provides the most benefit for systems that are I/O-bound, with tables for which the data is written once and then read repeatedly, as in the situations I mentioned in the previous paragraph: data warehousing and reporting. For environments with heavy read and write activity, such as OLTP applications, you might want to consider enabling row compression only and avoid the costs of analyzing the pages and rebuilding the CI record. In this case, the CPU overhead is minimal. In fact, row compression is highly optimized so that it is visible only at the storage engine layer. The relational engine (query processor) doesn't need to deal with compressed rows at all. The relational engine sends uncompressed rows to the storage engine, which compresses them if required. When returning rows to the relational engine, the storage engine waits as long as it can before uncompressing them. In the storage engine, comparisons can be done on compressed data, as internal conversions can convert a data type to its compressed form before comparing to data in the table. In addition, only columns requested by the relational engine need to be uncompressed, as opposed to uncompressing an entire row.

Compression and Logging In general, SQL Server logs only uncompressed data because the log needs to be read in an uncompressed format. This means that logging changes to compressed records has a greater performance impact because each row needs to be uncompressed and decoded (from the anchor record and dictionary) prior to writing to the log. This is another reason why compression gives you more benefit on primarily read-only systems, where logging is minimal.

SQL Server writes compressed data to the log in a few situations. The most common situation is when a page is split. SQL Server writes the compressed rows as it logs the data movement during the split operation.

Compression and the Version Store I will be discussing the version store in Chapter 10, when I talk about Snapshot isolation, but I want to mention briefly here how the version store interacts with compression. SQL Server can write compressed rows to the version store and the version store processing can traverse older versions in their compressed form. However, the version store does not support page compression, so the rows in the version store cannot contain encodings of the anchor record prefixes and the page dictionary. So anytime any row from a compressed page needs to be versioned, the page must be uncompressed first.

The version store is used for both varieties of Snapshot isolation (full snapshot and read-committed snapshot) and is also used for storing the before-and-after images of changed data when triggers are fired. (These images are visible in the logical tables *inserted* and *deleted*.) You should keep this in mind when evaluating the costs of compression. Snapshot isolation has lots of overhead already, and adding page compression into the mix affects performance even more.

Backup Compression

I mentioned backup compression briefly in Chapter 1, when discussing configuration options. I believe it bears repeating that the algorithm used for compressing backups is very different than the database compression algorithms discussed in this chapter. Backup compression uses an algorithm very similar to zipping, where it is just looking for patterns in the data. Even after tables and indexes have been compressed using the data compression techniques, they still can be compressed further using the backup compression algorithms.

Page compression looks only for prefix patterns, and it can still leave other patterns that are not compressed, including common suffixes. Page compression eliminates redundant strings, but there still are plenty of strings in most cases that are not redundant, and string data compresses very well using zip-type algorithms.

In addition, there is a fair amount of space in a database that constitutes overhead, such as unallocated slots on pages and unallocated pages in allocated extents. Depending on whether Instant File Initialization was used, and what was on the disk previously if it was, the background data can actually compress very well.

So making a compressed backup of a database that has many compressed tables and indexes can provide additional space savings for the backup set.

Table and Index Partitioning

As we've already seen when looking at the metadata for table and index storage, partitioning is an integral feature of SQL Server space organization. Figure 5-7 in Chapter 5 illustrated the relationship between tables and indexes (hobts), partitions, and allocation units. Tables and indexes that are built without any reference to partitions are considered to be

stored on a single partition. One of the more useful metadata objects for retrieving information about data storage is the dynamic management view called *sys.dm_db_partition_stats*, which combines information found in *sys.partitions*, *sys.allocation_units*, and *sys.indexes*.

A partitioned object is one that is split internally into separate physical units that can be stored in different locations. Partitioning is invisible to the users and programmers, who can use T-SQL code to select from a partitioned table exactly the same way they select from a nonpartitioned table. Creating large objects on multiple partitions improves the manageability and maintainability of your database system and can greatly enhance the performance of activities such as purging historic data and loading large amounts of data. In SQL Server 2000, partitioning was available only by manually creating a view that combines multiple tables. That functionality is referred to as *partitioned* views. The SQL Server 2005 and SQL Server 2008 built-in partitioning of tables and indexes has many advantages over partitioned views, including improved execution plans and fewer prerequisites for implementation.

In this section, we focus primarily on physical storage of partitioned objects and the partitioning metadata. In Chapter 8, “The Query Optimizer,” we’ll examine query plans involving partitioned tables and partitioned indexes.

Partition Functions and Partition Schemes

To understand the partitioning metadata, we need a little background into how partitions are defined. I will use an example based on the SQL Server samples. You can find my script, called *Partition.sql,* on the companion Web site. This script defines two tables: *TransactionHistory* and *TransactionHistoryArchive*, along with a clustered index and two nonclustered indexes on each. Both tables are partitioned on the *TransactionDat*e column, with each month of data in a separate partition. Initially, there are 12 partitions in *TransactionHistory* and 2 in *TransactionHistoryArchive*.

Before you create a partitioned table or index, you must define a partition function. A partition function is used to define the partition boundaries logically. When a partition function is created, you must specify whether or not the partition will use a LEFT- or RIGHT-based boundary point. Simply put, this defines whether the boundary value itself is part of the left-hand or right-hand partition. Another way to consider this is to ask this question: Is it an upper boundary of one partition (in which case it goes to the LEFT), or a lower boundary point of the next partition (in which case it goes to the RIGHT)? The number of partitions created by a partition function with *n* boundaries will be *n*+1. Here is the partition function that we are using for this example:

```
CREATE PARTITION FUNCTION [TransactionRangePF1] (datetime)
AS RANGE RIGHT FOR VALUES ('20081001', '20081101', '20081201',
               '20090101', '20090201', '20090301', '20090401',
               '20090501', '20090601', '20090701', '20090801');
```

Note that the table name is not mentioned in the function definition because the partition function is not tied to any particular table. The function *TransactionRangePF1* divides the data into 12 partitions because there are 11 *datetime* boundaries. The keyword *RIGHT* specifies that any value that equals one of the boundary points goes into the partition to the right of the endpoint. So for this function, all values less than October 1, 2008, go in the first partition, and values greater than or equal to October 1, 2008, and less than November 1, 2008, go in the second partition. I could have also specified *LEFT* (which is the default), in which case the value equal to the endpoint goes in the partition to the left. After you define the partition function, you define a partition scheme, which lists a set of filegroups onto which each range of data is placed. Here is the partition schema for my example:

```
CREATE PARTITION SCHEME [TransactionsPS1]
AS PARTITION [TransactionRangePF1]
TO ([PRIMARY], [PRIMARY], [PRIMARY]
, [PRIMARY], [PRIMARY], [PRIMARY]
, [PRIMARY], [PRIMARY], [PRIMARY]
, [PRIMARY], [PRIMARY], [PRIMARY]);
GO
```

To avoid having to create 12 files and filegroups, I have put all the partitions on the PRIMARY filegroup, but for the full benefit of partitioning, you should probably have each partition on its own filegroup. The *CREATE PARTITION SCHEME* command must list at least as many filegroups as there are partitions, but there can be one more. If one extra filegroup is listed, it is considered the “next used” filegroup. If the partition function splits, the new boundary point is added in the filegroup used next. If you do not specify an extra filegroup at the time you create the partition scheme, you can alter the partition scheme to set the next-used filegroup prior to modifying the function.

As you’ve seen, the listed filegroups do not have to be unique. In fact, if you want to have all the partitions on the same filegroup, as I do here, there is a shortcut syntax:

```
CREATE PARTITION SCHEME [TransactionsPS1]
AS PARTITION [TransactionRangePF1]
ALL TO ([PRIMARY]);
GO
```

Note that putting all the partitions on the same filegroup is usually just done for the purpose of testing your code.

Additional filegroups are used in order as more partitions are added, which can happen when a partition function is altered to split an existing range into two. If you do not specify extra filegroups at the time you create the partition scheme, you can alter the partition scheme to add another filegroup.

The partition function and partition scheme for a second table are shown here:

```
CREATE PARTITION FUNCTION [TransactionArchivePF2] (datetime)
AS RANGE RIGHT FOR VALUES ('20080901');
GO

CREATE PARTITION SCHEME [TransactionArchivePS2]
AS PARTITION [TransactionArchivePF2]
TO ([PRIMARY], [PRIMARY]);
GO
```

My script then creates two tables and loads data into them. I will not include all the details here. To partition a table, you must specify a partition scheme in the *CREATE TABLE* statement. I create a table called *TransactionArchive* that includes this line as the last part of the *CREATE TABLE* statement as follows:

```
ON [TransactionsPS1] (TransactionDate)
```

My second table, *TransactionArchiveHistory*, is created using the *TransactionsPS1* partitioning scheme.

My script then loads data into the two tables, and because the partition scheme has already been defined, each row is placed in the appropriate partition as the data is loaded. After the tables are loaded, we can examine the metadata.

Metadata for Partitioning

Figure 7-16 shows most of the catalog views for retrieving information about partitions. Along the left and bottom edges, you can see the *sys.tables*, *sys.indexes*, *sys.partitions*, and *sys.allocation_units* catalog views that I've discussed previously in this chapter.

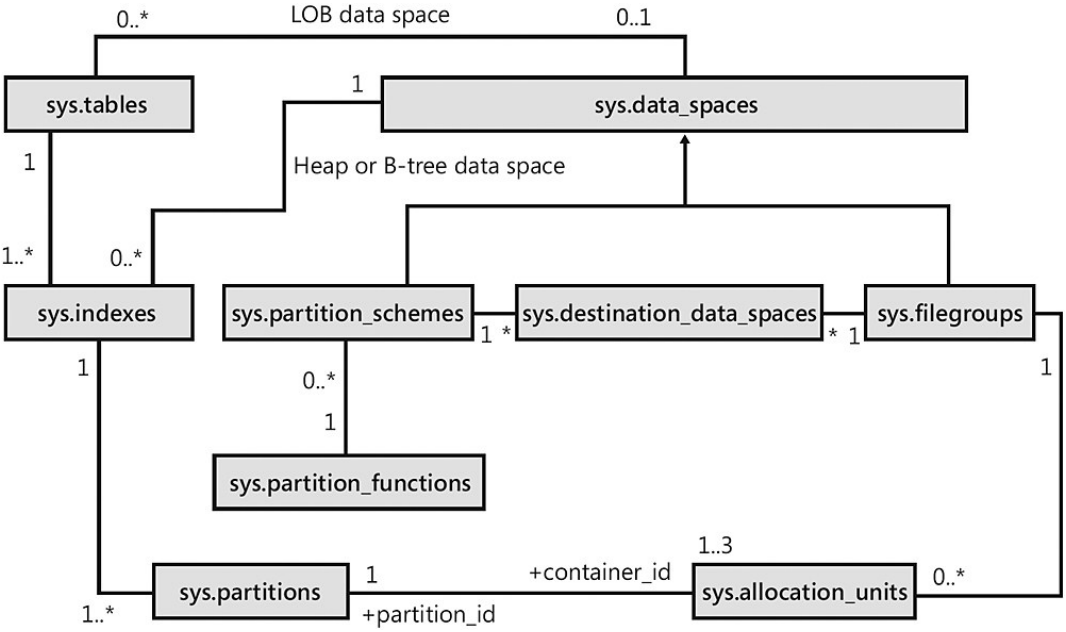

Figure 7-16: Catalog views containing metadata for partitioning and data storage

In some of my queries, I am using the undocumented *sys.system_internals_allocation_units* view instead of *sys.allocation_units* to retrieve page address information. Here, I'll describe the most relevant columns of each of these

views:

- ***sys.data_spaces*** has a primary key called *data_space_id,* which is either a partition ID or a filegroup ID, and there is one row for each filegroup and one row for each partition scheme. One of the columns in *sys.data_spaces* specifies to which type of data space the row refers. If the row refers to a partition scheme, *data_space_id* can be joined with *sys.partition_schemes.data_space_id.* If the row refers to a filegroup, *data_space_id* can be joined with *sys.filegroups.data_space_id.* The *sys.indexes* view also has a *data_space_id* column to indicate how each heap or B-tree stored in *sys.indexes* is stored. So, if we know that a table is partitioned, we can directly join it with *sys.partition_schemes* without going through *sys.data_spaces.* Alternatively, you can use the following query to determine whether a table is partitioned by replacing *Production.TransactionHistoryArchive* with the name of the table in which you're interested:

```
SELECT DISTINCT object_name(object_id) as TableName,
            ISNULL(ps.name, 'Not partitioned') as PartitionScheme
    FROM (sys.indexes i LEFT  JOIN sys.partition_schemes ps
                    ON (i.data_space_id = ps.data_space_id))
    WHERE (i.object_id = object_id('Production.TransactionHistoryArchive'))
              AND   (i.index_id IN (0,1));
```

- ***sys.partition_schemes*** has one row for each partition scheme. In addition to the *data_ space_id* and the name of the partition scheme, it has a *function_id* column to join with *sys.partition_functions.*

- ***sys.destination_data_spaces*** is a linking table because *sys.partition_schemes* and *sys.filegroups* are in a many-to-many relationship with each other. For each partition scheme, there is one row for each partition. The partition number is in the *destination_id* column, and the filegroup ID is stored in the *data_space_id* column.

- ***sys.partition_functions*** contains one row for each partition function, and its primary key *function_id* is a foreign key in *sys.partition_schemes.*

- ***sys.partition_range_values*** (not shown) has one row for each endpoint of each partition function. Its *function_id* column can be joined with *sys.partition_functions,* and its *boundary_id* column can join with either *partition_id* in *sys.partitions* or with *destination_id* in *sys.destination_data_spaces.*

These views have other columns that I haven't mentioned, and there are additional views that provide information, such as the columns and their data types that the partitioning is based on. However, the preceding information should be sufficient to understand Figure 7-15 and the view shown in the next block of code. This view returns information about each partition of each partitioned table. The WHERE clause filters out partitioned indexes (other than the clustered index), but you can change that condition if you desire. When selecting from the view, you can add your own WHERE clause to find information about just the table you're interested in:

```
CREATE VIEW Partition_Info AS
SELECT OBJECT_NAME(i.object_id) as Object_Name,
        p.partition_number, fg.name AS Filegroup_Name, rows, au.total_pages,
        CASE boundary_value_on_right
                  WHEN 1 THEN 'less than'
                  ELSE 'less than or equal to' END as 'comparison', value
FROM sys.partitions p JOIN sys.indexes i
     ON p.object_id = i.object_id and p.index_id = i.index_id
       JOIN sys.partition_schemes ps
               ON ps.data_space_id = i.data_space_id
       JOIN sys.partition_functions f
                  ON f.function_id = ps.function_id
       LEFT JOIN  sys.partition_range_values rv
ON f.function_id = rv.function_id
                   AND p.partition_number = rv.boundary_id
       JOIN sys.destination_data_spaces dds
              ON dds.partition_scheme_id = ps.data_space_id
                   AND dds.destination_id = p.partition_number
       JOIN sys.filegroups fg
                 ON dds.data_space_id = fg.data_space_id
       JOIN (SELECT container_id, sum(total_pages) as total_pages
                     FROM sys.allocation_units
                     GROUP BY container_id) AS au
                 ON au.container_id = p.partition_id
  WHERE i.index_id <2;
```

The LEFT JOIN operator is needed to get all the partitions because the *sys.partition_range_values* view has a row only for each boundary value, not for each partition. LEFT JOIN gives the last partition with a boundary value of NULL, which means that the value of the last partition has no upper limit. A derived table groups together all the rows in *sys.allocation_units* for a partition, so the space used for all the types of storage (in-row, row-overflow, and LOB) is aggregated into a single value. This query uses the preceding view to get information about my *TransactionHistory* table's partitions:

```
SELECT * FROM Partition_Info
WHERE Object_Name = 'TransactionHistory';
```

Here are my results:

Object_Name	Partition_number	Filegroup_Name	Rows	Total_pages	Comparison	Value
TransactionHistory	1	PRIMARY	11155	209	Less than	2008-10-01
TransactionHistory	2	PRIMARY	9339	177	Less than	2008-11-01
TransactionHistory	3	PRIMARY	10169	185	Less than	2008-12-01
TransactionHistory	4	PRIMARY	12181	225	Less than	2009-01-01
TransactionHistory	5	PRIMARY	9558	177	Less than	2009-02-01
TransactionHistory	6	PRIMARY	10217	193	Less than	2009-03-01
TransactionHistory	7	PRIMARY	10703	201	Less than	2009-04-01
TransactionHistory	8	PRIMARY	10640	193	Less than	2009-05-01
TransactionHistory	9	PRIMARY	12508	225	Less than	2009-06-01
TransactionHistory	10	PRIMARY	12585	233	Less than	2009-07-01
TransactionHistory	11	PRIMARY	3380	73	Less than	2009-08-01
TransactionHistory	12	PRIMARY	1008	33	Less than	NULL

This view contains details about the boundary point of each partition, as well as the filegroup that each partition is stored on, the number of rows in each partition, and the amount of space used. Note that although the comparison indicates that the values in the partitioning column for the rows in a particular partition are less than the specified value, you should assume that it also means that the values are greater than or equal to the specified value in the preceding partition. However, this view doesn't provide information about where in the particular filegroup the data is located. We'll look at a metadata query that gives us location information in the next section.

Note If a partitioned table contains filestream data, it is recommended that the filestream data be partitioned using the same partition function as the non-filestream data. Because the regular data and the filestream data are on separate filegroups, the filestream data needs its own partition scheme. However, the partition scheme for the filestream data can use the same partition function to make sure the same partitioning is used for both filestream and non-filestream data.

The Sliding Window Benefits of Partitioning

One of the main benefits of partitioning your data is that you can move data from one partition to another as a metadata-only operation. The data itself doesn't have to move. As I mentioned, this is not intended to be a complete how-to guide to SQL Server 2008 partitioning; rather, it is a description of the internal storage of partitioning information. However, to show the internals of rearranging partitions, we need to look at some additional partitioning operations.

The main operation you use when working with partitions is the SWITCH option to the *ALTER TABLE* command. This option allows you to

- Assign a table as a partition to an already-existing partitioned table
- Switch a partition from one partitioned table to another
- Reassign a partition to form a single table

In all these operations, no data is moved. Rather, the metadata is updated in the *sys.partitions* and

sys.system_internals_allocation_units views to indicate that a given allocation unit now is part of a different partition. Let's look at an example. The following query returns information about each allocation unit in the first two partitions of my *TransactionHistory* and *TransactionHistoryArchive* tables, including the number of rows, the number of pages, the type of data in the allocation unit, and the page where the allocation unit starts:

```
SELECT convert(char(25),object_name(object_id)) AS name,
    rows, convert(char(15),type_desc) as page_type_desc,
    total_pages AS pages, first_page, index_id, partition_number
FROM sys.partitions p JOIN sys.system_internals_allocation_units a
     ON p.partition_id = a.container_id
WHERE (object_id=object_id('[Production].[TransactionHistory]')
   OR object_id=object_id('[Production].[TransactionHistoryArchive]'))
  AND index_id = 1 AND partition_number <= 2;
```

Here is the data I get back. (I left out the *page_type_desc* because all the rows are of type IN_ROW_DATA.)

```
name                        rows    pages      first_page      index_id    partition_number
--------------------------  ------- ---------- --------------  ----------- ----------------
TransactionHistory          11155   209        0xD81B00000100  1           1
TransactionHistory          9339    177        0xA82200000100  1           2
TransactionHistoryArchive   89253   1553       0x981B00000100  1           1
TransactionHistoryArchive   0       0          0x000000000000  1           2
```

Now let's move one of my partitions. My ultimate goal is to add a new partition to *TransactionHistory* to store a new month's worth of data and to move the oldest month's data into *TransactionHistoryArchive*. The partition function used by my *TransactionHistory* table divides the data into 12 partitions, and the last one contains all dates greater than or equal to August 1, 2009. I'm going to alter the partition function to put a new boundary point in for September 1, 2009, so the last partition is split. Before doing that, I must ensure that the partition scheme using this function knows what filegroup to use for the newly created partition. With this command, some data movement occurs and all data from the last partition of any tables using this partition scheme is moved to a new allocation unit. Please refer to *SQL Server Books Online* for complete details about each of the following commands:

```
ALTER PARTITION SCHEME TransactionsPS1
NEXT USED [PRIMARY];
GO

ALTER PARTITION FUNCTION TransactionRangePF1()
SPLIT RANGE ('20090901');
GO
```

Next, I'll do something similar for the function and partition scheme used by *TransactionHistoryArchive*. In this case, I'll add a new boundary point for October 1, 2008:

```
ALTER PARTITION SCHEME TransactionArchivePS2
NEXT USED [PRIMARY];
GO

ALTER PARTITION FUNCTION TransactionArchivePF2()
SPLIT RANGE ('20081001');
GO
```

I want to move all data from *TransactionHistory* with dates earlier than October 1, 2008, to the second partition of *TransactionHistoryArchive*. However, the first partition of *TransactionHistory* technically has no lower limit; it includes everything earlier than October 1, 2008. The second partition of *TransactionHistoryArchive* does have a lower limit, which is the first boundary point, or September 1, 2008. To SWITCH a partition from one table to another, I have to guarantee that all the data to be moved meets the requirements for the new location. So I add a CHECK constraint that guarantees that no data in *TransactionHistory* is earlier than September 1, 2008. After adding the CHECK constraint, I run the *ALTER TABLE* command with the SWITCH option to move the data in partition 1 of *TransactionHistory* to partition 2 of *TransactionHistoryArchive*. (For testing purposes, you could try leaving out the next step that adds the constraint and try just executing the *ALTER TABLE/SWITCH* command. You get an error message. After that, you can add the constraint and run the *ALTER TABLE/SWITCH* command again.)

```
ALTER TABLE [Production].[TransactionHistory]
ADD CONSTRAINT [CK_TransactionHistory_DateRange]
CHECK ([TransactionDate] >= '20080901');
GO
ALTER TABLE [Production].[TransactionHistory]
SWITCH PARTITION 1
```

```
TO [Production].[TransactionHistoryArchive] PARTITION 2;
GO
```

Now we run the metadata query that examines the size and location of the first two partitions of each table as follows:

```
SELECT convert(char(25),object_name(object_id)) AS name,
    rows, convert(char(15),type_desc) as page_type_desc,
    total_pages AS pages, first_page, index_id, partition_number
FROM sys.partitions p JOIN sys.system_internals_allocation_units a
     ON p.partition_id = a.container_id
WHERE (object_id=object_id('[Production].[TransactionHistory]')
   OR object_id=object_id('[Production].[TransactionHistoryArchive]'))
  AND index_id = 1 AND partition_number <= 2;
RESULTS:
name                 rows    pages      first_page      index_id    partition_number
-------------------  ------- ---------- --------------  ----------- ----------------
TransactionHistory    0       0          0x000000000000  1           1
TransactionHistory    9339    177        0xA82200000100  1           2
TransactionHistoryAr  89253   1553       0x981B00000100  1           1
TransactionHistoryAr  11155   209        0xD81B00000100  1           2
```

You'll notice that the second partition of *TransactionHistoryArchive* now has exactly the same information that the first partition of *TransactionHistory* had in the first result set. It has the same number of rows (11,155), the same number of pages (209), and the same starting page (0xD81B00000100, or file 1, page 7,128). No data was moved; the only change was that the allocation unit starting at file 1, page 7,128 is not recorded as belonging to the second partition of the *TransactionHistoryArchive* table.

Although my partitioning script created the indexes for my partitioned tables using the same partition scheme used for the tables themselves, this is not always necessary. An index for a partitioned table can be partitioned using the same partition scheme or a different one. If you do not specify a partition scheme or filegroup when you build an index on a partitioned table, the index is placed in the same partition scheme as the underlying table, using the same partitioning column. Indexes built on the same partition scheme as the base table are called *aligned indexes*.

However, an internal storage component is associated with automatically aligned indexes. As previously mentioned, if you build an index on a partitioned table and do not specify a filegroup or partitioning scheme on which to place the index, SQL Server creates the index using the same partitioning scheme that the table uses. However, if the partitioning column is not part of the index definition, SQL Server adds the partitioning column as an extra included column in the index. If the index is clustered, adding an included column is not necessary because the clustered index already contains all the columns. Another case in which SQL Server does not add an included column automatically is when you create a unique index, either clustered or nonclustered. Because unique partitioned indexes require that the partitioning column is contained in the unique key, a unique index for which you have not explicitly included the partitioning key is not partitioned automatically.

Summary

In this chapter, we looked at how SQL Server 2008 stores data that doesn't use the normal FixedVar record format and data that doesn't fit into the normal 8-KB data page.

I discussed row-overflow and large object data, which is stored on its own separate pages, and filestream data, which is stored outside SQL Server, in files in the filesystem.

Some of the new storage capabilities in SQL Server 2008 require that we look at row storage in a completely different way. Sparse columns allow us to have very wide tables of up to 30,000 columns, so long as most of those columns are NULL in most rows. Each row in a table containing sparse columns has a special descriptor field that provides information about which columns are non-NULL for that particular row.

I also described a completely new row storage format used with compressed data. Data can be compressed at either the row level or the page level, and the rows and pages themselves describe the data that is contained therein. This type of row format is referred to as the CD format.

Finally we looked at partitioning of tables and indexes. Although partitioning doesn't really require a special format for your rows and pages, it does require accessing the metadata in a special way.