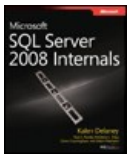

Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.

Chapter 6: Indexes—Internals and Management

Kalen Delaney, with Kimberly L. Tripp and Paul S. Randal

Microsoft SQL Server doesn't have a configuration option or a knob that allows you to make it run faster; there's no magic bullet. However, indexes—when created and designed appropriately—are probably the closest thing to a magic bullet. The right index, created for the right query, can take query execution time from hours down to seconds. There's absolutely no other way to see these kinds of gains—adding hardware, or tweaking configuration options often only give marginal gains. What is it about indexes that can make a query request drop from millions of I/Os to only a few? And does just any index improve performance? Unfortunately, great performance doesn't just happen; all indexes are not equal, nor is just any index going to improve performance. In fact, over-indexing is often worse than under-indexing. You can't just "index every column" and expect SQL Server to improve.

So how do you know how to create the best indexes? Honestly, it takes multiple pieces—knowing your data, knowing your workload, and knowing how SQL Server works. In terms of how SQL Server works, there are multiple components: index internals, statistics, query optimization, and maintenance. In this chapter, we focus on index internals and maintenance—expanding these topics to give you creation best practices and optimal base indexing strategies. By knowing how SQL Server physically stores indexes as well as how the storage engine accesses and manipulates these physical structures, you are better equipped to create the *right* indexes for your workload. In addition, this information helps to prepare you for Chapter 8, "The Query Optimizer," as you can visualize the choices (in terms of physical structures) from which SQL Server can choose and why some structures are more effective than others for certain requests.

This chapter is split into multiple sections. The first section explains index usage and concepts and internals. In this section, you learn how indexes are stored and how they work for data retrieval. The second section dives into what happens when data is modified—both how it happens and how SQL Server guarantees consistency. In this section, you also learn the potential effects of data modifications on indexes, such as fragmentation. Finally, the third section discusses index management and maintenance.

Overview

Think of the indexes you might see in your everyday life—those in books and other documents. Suppose that you're trying to create an index in SQL Server using the *CREATE INDEX* statement and you're using two SQL Server references to find out how to write the statement. One reference is the (hypothetical) *Microsoft SQL Server Transact-SQL Language Reference Manual,* which we'll refer to as the "T-SQL Reference." Assume that this book is just an alphabetical list of all the SQL Server keywords, commands, procedures, and functions. The other reference is this book: *Microsoft SQL Server 2008 Internals*. You can find information quickly in either book about indexes, even though the two books are organized differently.

In the T-SQL Reference, all the commands and keywords are organized alphabetically. You know that *CREATE INDEX* is near the front with all the other *CREATE* statements, so you can just ignore most of the rest of the book. Keywords and phrases are shown at the top of each page to tell you what commands are on that page. Thus, you can flip through just a few pages quickly and end up at a page that has *CREATE DATABASE* on it, and you know that *CREATE INDEX* appears shortly thereafter. Now, if you flipped forward and came to *CREATE VIEW* without passing *CREATE INDEX*, you'd know that *CREATE INDEX* was missing from the book, as the commands and keywords are organized alphabetically. (Of course, this is just an example—*CREATE INDEX* would certainly be in the T-SQL Reference.)

Next, you try to find *CREATE INDEX* in *Microsoft SQL Server 2008 Internals*. This book is *not* ordered alphabetically by commands and keywords, but there's an index at the back of the book, and all the index entries are organized alphabetically. So, again, you can use the fact that *CREATE INDEX* is near the front of the alphabet and find it quickly. However, unlike in the T-SQL Reference, once you find the words *CREATE INDEX*, you won't see nice, neat examples right in front of you. The index only gives you pointers—it tells you what pages to look at. In fact, it might list many pages in the book. And, if you look up *CREATE TABLE* in the book's index, you might find dozens of pages listed. Finally, if you look up the stored procedure, *sp_addumpdevice* (a completely deprecated command), you won't find it in the index at all because it's not described in this book.

The point is that these two searches are analogous to using a clustered index (in the case of the book's contents actually being ordered) and a nonclustered index (in the case of the lookup from the index into the book). If a table is clustered, the table data is logically stored in the clustering key order, just as the T-SQL Reference has all the main topics in order. Once you find the data you're looking for, your search is complete. In a nonclustered index, the index is a completely separate

structure from the data itself. Once you find what you're looking for in the index, you have to follow some sort of reference pointer to get to the actual data. Although a nonclustered index in SQL Server is very much like the index in the back of a book, it is not exactly the same.

SQL Server Index B-trees

In SQL Server, indexes are organized using a B-tree structure, as shown in Figure 6-1. *B-tree* stands for "balanced tree," and SQL Server uses a special kind called *B+ trees* (pronounced "b-plus trees") *that are not kept strictly balanced in all ways at all times.* Unlike a normal tree, B-trees are always inverted, with their root (a single page) at the top and their leaf level at the bottom. The existence of intermediate levels depends on multiple factors. *B-tree* is an overloaded term used in different ways by different people—either to mean the entire index structure or just the non-leaf levels. In this book, the term *B-tree* means the entire index structure.

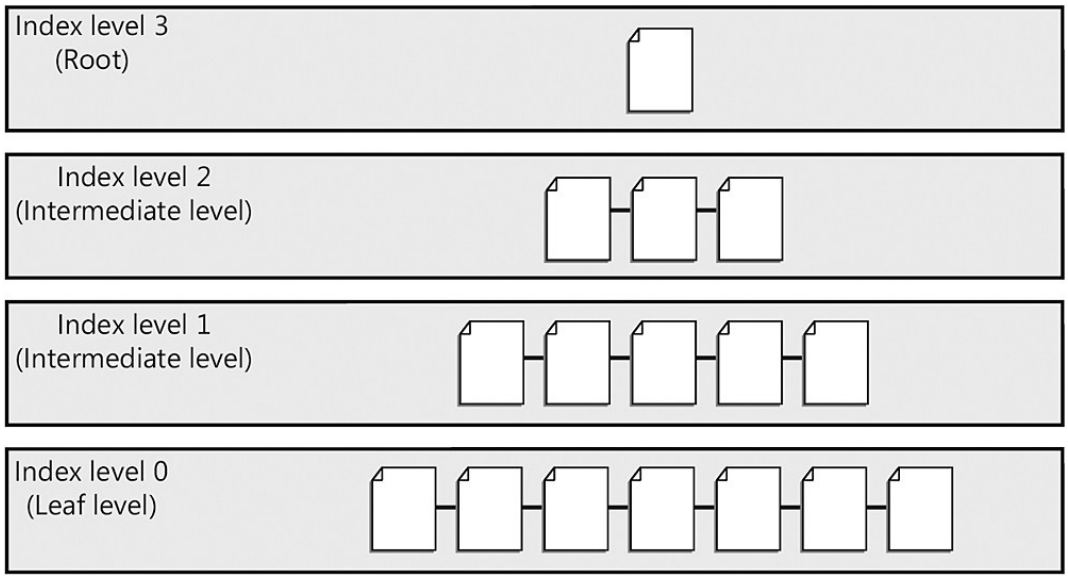

Figure 6-1: A B-tree for a SQL Server index

What's interesting about these B-trees in SQL Server is how they are constructed and what is contained in each level. Structurally, indexes might change a small amount based on whether or not multiple CPUs are used to create or rebuild them (which is explained in more detail in the section "MAXDOP," later in this chapter), but for the most part, the size and width of the tree are based on the definition of the index and the number and size of rows in the table. To show this, we give a few examples starting with the general terms and definitions. First, indexes have two primary components: a leaf level and one or more non-leaf levels. The non-leaf levels are interesting to understand and discuss, but simply put, they're used for navigation (mostly for navigating to the leaf level). However, the first intermediate level is also used in fragmentation analysis and to drive read-ahead during large range scans of the index.

To understand these structures, we start with defining the leaf level in generic terms (meaning that these basic concepts apply to both clustered and nonclustered indexes). The leaf level of an index contains *something* (we discuss the specifics when we get into the topic of physical index structures later in the chapter) for every row of the table in indexed order. In this discussion, we are focusing on traditional indexes and those created without filters, which refers to a new SQL Server 2008 feature called *filtered indexes.*

Non-leaf levels exist to *help navigate to a row* at the leaf level but the architecture is rather straightforward. Each non-leaf level stores *something* for every page of the level below—and levels are added until the index builds up to a root of one page. Each higher non-leaf level in the index (that is, farther away from the leaf level) is smaller than the one below it because each row at a level contains only the minimum key value that can be on each page of the level below, plus a pointer to that page. Although it sounds like this could result in a lot of levels (that is, a tall tree), the limitation on the size of the key (which has a maximum of 900 bytes or 16 columns—whichever comes first) in SQL Server helps to keep index trees relatively small.

In fact, in the example that we show coming up—which has an index with fairly wide rows and a key definition that is at the maximum size—the tree size of this example index (at the time the index is created) is only eight levels deep.

To see this tree (and the computations used to determine its size), we use an example where the leaf level of the index contains 1,000,000 “rows.” We put quotes around rows because these are not necessarily data rows—these are just leaf-level rows of *any* index. Later in the chapter—when we discuss the physical structures of each specific index—you will see exactly what leaf-level rows are and how they are structured. However, for this example, we’re focused on an abstract “index” where we’re concerned only about the leaf and non-leaf levels—as well as how they’re structured within the confines of SQL Server pages (8-KB pages). In this example, our leaf-level rows are 4,000 bytes, which means we can store only two rows per page. For a table with 1,000,000 rows, the leaf level of our index would have 500,000 pages. Relatively speaking, this is a fairly wide row structure; however, we are not wasting a lot of space on the page. If our leaf-level page had two 3,000-byte rows we’d still only fit two rows per page, but then we’d have 2,000 bytes of wasted space. (This would be an example of internal fragmentation, which is discussed in the section entitled “Fragmentation,” later in this chapter.)

Now, why are these just “rows” and not specifically data rows? The reason is that this leaf level *could* be the leaf level for a clustered index (therefore data rows) *or* these leaf-level rows could be rows in a nonclustered index that uses *INCLUDE* (which was added in SQL Server 2005) to add non-key columns to the leaf level of the index. When *INCLUDE* is used, leaf-level pages can contain wider rows (wider than the 900-byte or 16-column key maximum). Again, although this doesn’t currently sound interesting, we explain later in this chapter why this can be beneficial. In this example, the leaf level of this index would be 4 GB in size (500,000 8-KB pages) at the time it’s created. This structure, depending on its definition could become larger—and possibly very fragmented—if a lot of new data is added. However (and again depending on its definition), there are ways to control how fragmented this index becomes when data is volatile (we look at this topic further in multiple sections later in this chapter). In this case, the leaf level of the index is large because of “row” width. And, using the maximum of 900 bytes means that you can fit only eight (8,096 bytes per page/900 bytes per row) rows per non-leaf level page. However, using this maximum, the resulting tree (up to a root of one page) would be *relatively* small and result only in eight levels—as shown here. In fact, improving scalability is the primary reason for the limit to an index key of 900 bytes or 16 columns—whichever comes first:

- Root page of non-leaf level (Level 7) = 2 rows = 1 page (8 rows per page)
- Intermediate non-leaf level (Level 6) = 16 rows = 2 pages (8 rows per page)
- Intermediate non-leaf level (Level 5) = 123 rows = 16 pages (8 rows per page)
- Intermediate non-leaf level (Level 4) = 977 rows = 123 pages (8 rows per page)
- Intermediate non-leaf level (Level 3) = 7,813 rows = 977 pages (8 rows per page)
- Intermediate non-leaf level (Level 2) = 62,500 rows = 7,813 pages (8 rows per page)
- Intermediate non-leaf level (Level 1) = 500,000 rows = 62,500 pages (8 rows per page)
- Leaf level (Level 0) = 1,000,000 rows = 500,000 pages (2 rows per page)

An index with a smaller key size would scale even faster. Imagine the same leaf-level pages as shown previously (1,000,000 rows at 2 rows per page) but with a smaller index key and therefore a smaller row size in the non-leaf levels (including some space for overhead) of only 20 bytes, you can fit 404 rows per non-leaf-level page:

- Root page of non-leaf level (Level 3) = 4 rows = 1 page (404 rows per page)
- Intermediate non-leaf level (Level 2) = 1,238 rows = 4 pages (404 rows per page)
- Intermediate non-leaf level (Level 1) = 500,000 rows = 1,238 pages (404 rows per page)
- Leaf level (Level 0) = 1,000,000 rows = 500,000 pages (2 rows per page)

In this second example, not only is the initial index only four levels, but it can have an additional 130,878,528 rows added (the maximum possible number of rows is 404*404*404*2—or 131,878,528—minus the number of rows that already exist—1,000,000) before it would require another level. Think of it like this—the root page currently allows 404 entries; however, we’re only storing 4 (and the existing non-leaf levels are not entirely 100 percent full). This is only a theoretical maximum, but without any other factors—such as fragmentation—a four-level tree would be able to seek into a table with over 131 million rows (again, with this small index key size). This means that a lookup into this index which uses the tree to navigate down to the corresponding row requires only four I/Os. And because the trees are balanced, finding any record requires the same amount of resources. Retrieval speed is consistent because the index has the same depth throughout. An index

can become fragmented—and pages can become less dense—but these trees do not become unbalanced. This is something we look at later in this chapter when we cover index maintenance.

It's not critical to memorize all the math that was used to show these examples, but understanding the true scalability of indexes—especially with reasonably created keys—means you are likely to create more effective indexes (that is, more efficient, with narrower keys). In addition, there are tools inside SQL Server to help you see the actual structures (no math required). Most importantly, the size of an index (and the number of levels) depends on three things—the index definition, whether or not the base table has a clustered index, and the number of pages in the leaf level of the indexes. The number of leaf-level pages is directly tied to both row size and the number of rows in the table. This does not mean that the goal when defining indexes is to have only very narrow indexes—in fact, extremely narrow indexes usually have fewer uses than slightly wider indexes. It just means that you should understand the implications of different indexing choices and decisions. In addition, features such as *INCLUDE* and filtered indexes can profoundly affect the index in both size and usefulness. However, knowing how SQL Server works and the internal structures of indexes are a large part of finding the right balance between having too many and too few indexes, but most importantly, of having the right indexes.

Tools for Analyzing Indexes

To expose and understand index structures fully, there are a few tools that we're going to use. To make the scenarios easier to understand, we need to get a feel for which tool is the most appropriate to use and when. In addition, this section focuses on an overview of the options for execution, as well as some tips and tricks. However, details on analyzing various aspects of the output can be found throughout this chapter.

Using the dm_db_index_physical_stats DMV

The *sys.dm_db_index_physical_stats* function is one of the most useful functions to determine table structures. DMV can give you insight into whether or not your table has a clustered index, how many nonclustered indexes exist, and whether or not your table (and each index) has row-overflow or Large Object (LOB) data. Most importantly, it can expose to you the entire structure and its state of health. This particular DMV is a function that requires five parameters, all with defaults. If you set all the parameters to their defaults and do not filter the rows or the columns, the function returns 21 columns of data for (almost) every level of every index on every table on every partition in every database of the current SQL Server instance. You would request that information as follows:

```
SELECT * FROM sys.dm_db_index_physical_stats (NULL, NULL, NULL, NULL, NULL);
```

When executed on a very small SQL Server instance, with only the *AdventureWorks2008, pubs,* and *Northwind* databases in addition to the system databases, more than 390 rows are returned. Obviously, 21 columns and 390 rows is too much output to illustrate here, so this is a command that you should play with to get some experience. However, it's unlikely that you actually want to see every index on every table in every database (although that can have some benefits on smaller instances such as a development instance). To distill this to a more targeted execution, let's look at the parameters now:

- **database_id** The first parameter must be specified as a number, but you can embed the *DB_ID* function as a parameter if you want to specify the database by name. If you specify NULL, which is the default, the function returns information about all databases. If the database ID is NULL, the next three parameters must also be NULL (which is their default value). In addition, this function must be executed in a database that has a compatibility mode of at least 90 (indicating SQL Server 2005). If, for some reason, your database is not running in at least compatibility mode 90, then executing this query from *master* and specifying a database name (`DB_ID('databasename')`) or the specific ID means that you can execute this without changing the target database's compatibility mode.

- **object_id** The second parameter is the object ID, which must also be a number, not a name. Again, the NULL default means you want information about all objects, and in that case, the next two parameters, *index _id* and *partition_id,* must be NULL. Just as for the database ID, you can use an embedded function (*OBJECT_ID*) to get the object ID if you know the object name. As a word of caution, if you're executing this from a different database than your current database, you should use a three-part object name with the *OBJECT_ID* function, including the database name and the schema name.

- **index_id** The third parameter allows you to specify the index ID from a particular table, and again, the default of NULL indicates that you want all the indexes. A handy fact to remember here is that the clustered index on a table always has an *index_id* of 1.

- **partition_number** The fourth parameter indicates the partition number, and NULL means you want information for all the partitions. Remember that if you haven't explicitly created a table or index on a partition scheme, SQL Server

internally considers it to be built on a single partition.

- **mode** The fifth and last parameter is the only one for which the default NULL does not result in returning the most information. The last parameter indicates the level of information that you want returned (and therefore directly affects the speed of execution) when querying this function. When the function is called, SQL Server traverses the page chains for the allocated pages for the specified partitions of the table or index. Unlike *DBCC SHOWCONTIG* in SQL Server 2000, which usually requires a shared (S) table lock, *sys.dm_db_index_physical_stats* (and *DBCC SHOWCONTIG* in SQL Server 2005) requires only an Intent-Shared (IS) table lock, which is compatible with most other kinds of locks, as discussed in Chapter 10, “Transactions and Concurrency.” Valid inputs are DEFAULT, NULL, LIMITED, SAMPLED, and DETAILED. The default is NULL, which corresponds to LIMITED. Here is what the latter three values mean:
  - **LIMITED** The LIMITED mode is the fastest and scans the smallest number of pages. For an index, it scans only the first non-leaf (or intermediate) level of the index. For a heap, a scan is avoided by using the table’s IAMs and then the associated Page Free Space (PFS) pages to define the allocation of the table. This allows SQL Server to obtain details about fragmentation in terms of page order (more on this later in the chapter) but not page density (or other details that can only be calculated from actually reading the leaf-level pages). In other words, it’s fast but not quite as detailed. More specifically, this corresponds to the WITH FAST option of the now-deprecated *DBCC SHOWCONTIG* command.
  - **SAMPLED** The SAMPLED mode returns physical characteristics based on a 1-percent sample of all the pages in the index or heap, plus the page order from reading the pages at the first intermediate level. However, if the index has less than 10,000 pages total, SQL Server converts SAMPLED to DETAILED.
  - **DETAILED** The DETAILED mode scans all pages and returns all physical characteristics (both page order and page density) for all levels of the index. This is incredibly helpful when analyzing a small table but can take quite a bit of time for larger tables. It could also essentially “flush” your buffer pool if the index being processed is larger than the buffer pool.

You must be careful when using the built-in *DB_ID* or *OBJECT_ID* functions. If you specify an invalid name or simply misspell the name, you do not receive an error message and the value returned is NULL. However, because NULL is a valid parameter, SQL Server just assumes that this is what you meant to use. For example, to see all the previously described information, but only for the *AdventureWorks2008* database, you might mistype the name as follows:

```
SELECT * FROM sys.dm_db_index_physical_stats
         (DB_ID ('AdventureWorks208'), NULL, NULL, NULL, NULL);
```

There is no such database as *AdventureWorks208*, so the *DB_ID* function returns NULL, and it is as if you had called the function with all NULL parameters. No error or warning is given.

You might be able to guess from the number of rows returned that you made an error, but of course, if you have no idea how much output you are expecting, it might not be immediately obvious. *SQL Server Books Online* suggests that you can avoid this issue by capturing the IDs into variables and error-checking the values in the variables before calling the *sys.dm_db_index_physical_stats* function, as shown in this code:

```
DECLARE @db_id SMALLINT;
DECLARE @object_id INT;

SET @db_id = DB_ID (N'AdventureWorks2008');
SET @object_id = OBJECT_ID (N'AdventureWorks2008.Person.Address');


IF (@db_id IS NULL OR @object_id IS NULL)
BEGIN
    IF @db_id IS NULL
    BEGIN
        PRINT N'Invalid database';
    END;
    ELSE IF @object_id IS NULL
    BEGIN
        PRINT N'Invalid object';
    END
END
ELSE
SELECT *
```

```
FROM sys.dm_db_index_physical_stats
    (@db_id, @object_id, NULL, NULL, NULL);
```

Another more insidious problem is that the *OBJECT_ID* function is called based on your current database, before any call to the *sys.dm_db_index_physical_stats* function is made. So if you are in the *AdventureWorks2008* database but want information from a table in the *pubs* database, you could try running the following code:

```
SELECT *
FROM sys.dm_db_index_physical_stats
      (DB_ID (N'pubs'), OBJECT_ID (N'dbo.authors'), NULL, NULL, NULL);
```

However, because there is no *dbo.authors* table in the current database (*AdventureWorks2008*), *@object_id* is passed as NULL, and you get all the information from all the objects in *pubs*.

If an object with the same name exists in two databases, the problem may be even harder to detect. If there were a *dbo.authors* table in *AdventureWorks2008*, the ID for that table would be used to try to retrieve data from the *pubs* database—and it's unlikely that the *authors* table has the same ID even if it exists in both databases. SQL Server returns an error if the ID returned by *object_id()* does not match any object in the specified database, but if does match the object ID for another table, the details for *that* table are produced, potentially causing even more confusion. The following script shows the error:

```
USE AdventureWorks2008;
GO

CREATE TABLE dbo.authors
   (ID CHAR(11), name varchar(60));
GO

SELECT *
FROM sys.dm_db_index_physical_stats
      (DB_ID (N'pubs'), OBJECT_ID (N'dbo.authors'), NULL, NULL, NULL);
```

When you run the preceding *SELECT*, the *dbo.authors* ID is determined based on the current environment, which is still *AdventureWorks2008*. But when SQL Server tries to use that ID (which does not exist) in *pubs*, the following error is generated:

```
Msg 2573, Level 16, State 40, Line 1
Could not find table or object ID 295672101. Check system catalog.
```

The best solution is to fully qualify the table name, either in the call to the *sys.dm_db_index_ physical_stats* function itself or, as in the code sample shown earlier, to use variables to get the ID of the fully qualified table name. If you write wrapper procedures to call the *sys.dm_db_index_physical_stats* function, you can concatenate the database name onto the object name before retrieving the object ID, thereby avoiding the problem. Because the output of this function is a bit cryptic, you might find it beneficial to write your own procedure to access this function and return the information in a slightly friendlier fashion.

In summary, this DMV is incredibly useful for determining the size and health of your indexes; however, you need to know how to work with it to get only the specific information in which you're interested. But even for a subset of tables or indexes, and with careful use of the available parameters, you still might get more data back than you want. Because *sys.dm_db_index_ physical_stats* is a table-valued function, you can add your own filters to the results being returned. For example, you can choose to look at the results for just the nonclustered indexes. Using the available parameters, your only choices are to see all the indexes or only one particular index. If we make the third parameter NULL to specify all indexes, we can then add a filter in a WHERE clause to indicate that we want only nonclustered index rows (WHERE *index_id* > 1). Note that while a WHERE clause may limit the number of rows returned it does not necessarily limit the tables and indexes analyzed.

Using DBCC IND

The *DBCC IND* command (introduced in Chapter 5, "Tables") is undocumented but widely known and used. It is safe to use on production systems. The command has four parameters, but only the first three are required. The following code shows the command syntax:

```
DBCC IND ( { 'dbname' | dbid }, { 'objname' | objid },
      { nonclustered indid | 1 | 0 | -1 | -2 } [, partition_number]  )
```

The first parameter is the database name or the database ID. The second parameter is an object name or object ID within

the database; the object can be either a table or an indexed view. The third parameter is a specific nonclustered index ID (2-250 or 256-1005) or the values 1, 0, –1, or –2. The values for this parameter have the following meanings:

- **0** Displays information for in-row data pages and in-row IAM pages of the specified object.
- **1** Displays information for all pages, including IAM pages, data pages, and any existing LOB pages or row-overflow pages of the requested object. If the requested object has a clustered index, the index pages are included.
- **–1** Displays information for all IAMs, data pages, and index pages for all indexes on the specified object. This includes LOB and row-overflow data.
- **–2** Displays information for all IAMs for the specified object.
- **Nonclustered index ID** Displays information for all IAMs, data pages, and index pages for one index. This includes LOB and row-overflow data that might be part of the index's included columns.

The final parameter was new for SQL Server 2005 and is optional (to maintain backward compatibility with scripts that might use *DBCC IND* from SQL Server 2000). It specifies a particular partition number, and if no value is specified or a 0 is given, information for all partitions is displayed.

Unlike *DBCC PAGE* (discussed in Chapter 5), SQL Server does not require that you enable trace flag 3604 before running *DBCC IND*. However, because it's likely that you will want to investigate pages using *DBCC PAGE*, after determining the pages owned by an index, it's a good idea to turn the trace flag on at the beginning of your script.

The columns in the result set are described in Table 6-1. Note that all page references have the file and page component conveniently split between two columns, so you don't have to do any conversion.

Table 6-1: Column Descriptions for DBCC IND Output

Column	Meaning
PageFID	File ID containing the page
PagePID	Page number within that file
IAMFID	File ID containing the IAM managing this page
IAMPID	Page number within that file of the IAM managing this page
ObjectID	Object ID
IndexID	Index ID—valid values are 0–250 and 256–1005 (described later)
PartitionNumber	Partition number within the table or index for this page
PartitionID	ID for the partition containing this page (unique in the database)
iam_chain_type	Type of allocation unit this page belongs to: in-row data, row-overflow data, or LOB data
PageType	Page type: 1 = data page, 2 = index page, 3 = TEXT_MIXED_PAGE, 4 = TEXT_TREE_PAGE, 10 = IAM page
IndexLevel	Level of index; 0 is the leaf level and levels are counted up from the leaf to the root page (of an index structure with *IndexID* of 1–1005)
NextPageFID	File ID containing the next page at this level
NextPagePID	Page number within that file for next page at this level
PrevPageFID	File ID containing the previous page at this level
PrevPagePID	Page number within that file for previous page at this level

Some of the return values were described in Chapter 5 because they are equally relevant to heaps. When dealing with indexes, we also can look at the *IndexID* column, which is 0 for a heap, 1 for pages of a clustered index, and a number between 2 and 1,005 for the pages of a nonclustered index pages. In SQL Server 2008, a table can have up to 1,000 total indexes (1 clustered and 999 nonclustered). Although 1,005 is higher than would be expected (2–1,000 would be sufficient for 999 nonclustered indexes), the range of nonclustered index IDs skips 251–255 because 255 had special meaning in earlier releases (it was used for the LOB values in a table) and 251–254 were unused. To simplify any backward-compatibility issues, this range (251–255) has been skipped in SQL Server 2008.

The *IndexLevel* value allows us to see at what level of the index tree a page is located, with a value of 0 meaning the leaf

level. The highest value for any particular index is, therefore, the root page of that index, and you should be able to verify that the root page is the same value you get from the *sys.system_internals_allocation_units* view in the *root_page* column. The remaining four columns indicate the page linkage at each level of each index. For each page, there is a file ID and page ID for the next page and a file ID and page ID for the previous page. Of course, for the root pages, all these values are 0. You can also determine the first page by finding one with zeros for the previous page, and you can find the last page because it has zeros for the next page. Because the output of this DBCC command is too wide to display in a page of a book, and because it's likely that you want to reorder the result set, we are not going to reproduce it here. If you wish to view it, you can use a script that stores the output of this command into a table. Once we have this information in a table, we can query it and retrieve just the columns in which we are interested. Here is a script that creates a table called *sp_tablepages* with columns to hold all the returned information from *DBCC IND*. Note that any object created in the *master* database with a name that starts with *sp_* can be accessed from any database, without having to qualify it with the database name:

```
USE master;
GO
CREATE TABLE sp_tablepages
(PageFID  tinyint,
  PagePID int,
  IAMFID   tinyint,
  IAMPID  int,
  ObjectID  int,
  IndexID  tinyint,
  PartitionNumber tinyint,
  PartitionID bigint,
  iam_chain_type  varchar(30),
  PageType  tinyint,
  IndexLevel  tinyint,
  NextPageFID  tinyint,
  NextPagePID  int,
  PrevPageFID  tinyint,
  PrevPagePID int,
  Primary Key (PageFID, PagePID));
```

The following code truncates the *sp_tablepages* table and then fills it with *DBCC IND* results from the *Sales.SalesOrderDetail* table in the *AdventureWorks2008* database:

```
TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (AdventureWorks2008, [Sales.SalesOrderDetail], -1)');
```

Once you have the results of *DBCC IND* in a table, you can select any subset of rows or columns that you are interested in. We use *sp_tablepages* to report on *DBCC IND* information for many examples in this chapter. You can then use *DBCC PAGE* to examine index pages, just as you do for data pages. However, if you use *DBCC PAGE* with style 3 to print out the details of each column on each row on an index page, the output looks quite different. We see some examples as we analyze the physical structures of indexes next.

Understanding Index Structures

As we discussed earlier in this chapter, index structures are divided into two basic components of the index: the leaf level and the non-leaf level(s). The details in this section help you to better understand what's specifically stored within these portions of your indexes and how they differ based on index type.

The Dependency on the Clustering Key

The leaf level of a clustered index contains the data, not just the index keys. So the answer to the question "What else is in the leaf level of a clustered index besides the key value?" is "Everything else"—that is, all the columns of every row in the table are in the leaf level of a clustered index. Another way to say this is that when a clustered index is created, the data becomes the leaf level of the clustered index. At the time a clustered index is created, data in the table is copied and ordered by the clustering key. Once created, a clustered index is maintained logically rather than physically. This order is maintained through a doubly linked list called a *page chain*. (Note that pages in a heap are not linked in any way to each other.) The order of pages in the page chain, and the order of rows on the data pages, is based on the definition of the clustered index. Deciding on which column(s) to cluster is an important performance consideration.

Because the actual page chain for the data pages can be ordered in only one way, a table can have only one clustered

index. And, in general, most tables perform better when the table is clustered. However, the clustering key needs to be chosen wisely. And, to appropriately choose a clustering key, you must understand how the clustered index works, as well as the internal dependencies on the clustering key (especially as far as the nonclustered indexes are concerned).

The dependencies of the nonclustered indexes on the clustering key have been in SQL Server since the storage engine was rearchitected in SQL Server 7.0. It all starts with how rows are identified (and looked up) when using a nonclustered index to reference a corresponding row within the table. If a table has a clustered index, then rows are identified (and looked up by) their clustering key. If the table does not have a clustered index, then rows are identified (and looked up by) their physical row identifier (RID), described in more detail later in this chapter. This process of looking up corresponding data rows in the base table is known as a *[bookmark] lookup,* which is named after the analogy that nonclustered indexes reference a place within a book, as a bookmark does.

Nonclustered indexes contain only the data as defined by the index. When looking up a row within a nonclustered index, you often need to go to the actual data row for additional data that's not part of the nonclustered index. To retrieve this additional data, you must look into the table for that data. For the purpose of this section, we focus only on how the bookmark lookup is performed when a table is clustered.

First, and foremost, all clustered indexes must be unique. The primary reason why a clustered index must be unique is so that nonclustered index entries can point to exactly one specific row. Consider the problem that would occur if a table were clustered by a nonunique value of last name. If a nonclustered index existed on a unique value, such as social security number, and a query looked into the index for a specific social security number of 123-45-6789 and found that its clustering key was 'Smith,' then if multiple rows with a last name of Smith existed, the question would be—which one? How would the specific row with a social security number of 123-45-6789 be located efficiently?

For a clustering key to be used effectively, all nonclustered index entries must refer to exactly one row. Because that pointer is the clustering key in SQL Server, then the clustering key must be unique. If you build a clustered index without specifying the *UNIQUE* keyword, SQL Server guarantees uniqueness internally by adding a hidden uniquifier column to the rows when necessary.

> **Note** In *SQL Server Books Online*, the word *uniquifier* is written as uniqueifier; however, the internal tools—such as *DBCC PAGE*—spell it as we've spelled it here.

This uniquifier is a 4-byte integer value added to the data row when the row's clustering key is not unique. Once added, it becomes part of the clustering key, meaning that it is duplicated in every nonclustered index. You can see whether or not a specific row has this extra value when you review the actual structure of index rows, as we will see later in this chapter.

Second, if a clustering key is used to look up the corresponding data rows from a nonclustered index into the clustered index (the data) then the clustering key is the most overly duplicated data in a table; all the columns that make up the clustering key are included in every nonclustered index in addition to being in the actual data row. As a result, the width of the clustering key is important. Consider a clustered index with a 64-byte clustering key on a table with 12 nonclustered indexes and 1 million rows. Without counting internal and structural overhead, the overhead required just to store the clustering key (to support the lookup) in every nonclustered index is 732 MB compared to only 92 MB if the clustering key were only 8 bytes and only 46 MB if the clustering key were only 4 bytes. Although this is just a rough estimate, it shows that you waste a lot of space (and potentially buffer pool memory) if you have an overly wide clustering key. However, it's not just about space alone; this also translates into performance and efficiency of your nonclustered indexes. And, in general, you don't want your nonclustered indexes to be unnecessarily wide.

Third, and because the clustering key is the most redundant data within your entire table, you should be sure to choose a clustering key that is *not* volatile. If a clustering key changes, then it can have multiple negative effects. First, it can cause record relocation within the clustered index (which can cause page splits and fragmentation, which we discuss in more detail later in this chapter). Second, it causes every nonclustered index to be modified (so that the value of the clustering key is correct for the relevant nonclustered index rows). This wastes time and space, causes fragmentation which then requires maintenance, and adds unnecessary overhead to every modification of the column(s) that make up the clustering key.

These three attributes—unique, narrow, and static—also (but not always) apply to a well-chosen primary key, and because you can have only one primary key (and only one clustering key), SQL Server uses a unique clustered index to enforce a primary key constraint (when no index type is defined in the primary key definition). However, this is not always known by the table's creator. And, if the primary key doesn't adhere to these criteria (for example, when the primary key has been chosen from the data's natural key, which, for example, is a wide, 100-byte combination of seven columns that is unique only when combined), then using a clustered index to enforce uniqueness and duplicating the entire 100-byte combination

of columns in every nonclustered index can have very negative side effects. So, for some unsuspecting database developers, a very wide clustering key may have been created for them because of these defaults. The good news is that you can define the primary key to be nonclustered and easily create a clustered index on a different column (or set of columns); however, you have to know when—and how—to do this.

Finally, a table's clustering key should also be chosen so as to minimize fragmentation for inserts (fragmentation is discussed in more detail later in this chapter). Although only the logical order of a clustered index is maintained after it is created, the maintenance of this structure does have overhead. If rows consistently need to enter the table at random entry points (for example, inserts into a table ordered by last name), then that table's logical order is slightly more expensive to maintain than a table that's always adding rows to the end of the table (for example, inserts into a table ordered by order number, which is—or should be—an ever-increasing identity column).

More details will be available as we review the internals of indexes later in the chapter, but to summarize our discussion thus far, the clustering key should be chosen not only based on table usage (and, it's really hard to say "always" or "never" with regard to the clustering key) but also based on the internal dependencies that SQL Server has on the clustering key. For the latter, the clustering key should be unique, narrow, and static—and preferably, ever-increasing.

Examples of good clustering keys are the following:

- A single column key defined with an ever-increasing identity column (for example, a 4-byte *int* or an 8-byte *bigint*).
- A composite key defined with an ever-increasing date column (first), followed by a second column that uniquely identifies the rows—like an identity column. This can be very useful for date-based partitioned tables and tables where the data is inserted in increasing date-based order as it offers an additional benefit for range queries on date. Examples of this include a 12-byte composite key comprised of *SalesDate* (8 bytes) and *SalesNumber* (4-byte *int*) or, in SQL Server 2008, a date column that does not include time. However, date alone is not a good clustering key because it is not unique (and requires a uniquifier).
- A GUID column can be used successfully as a clustering key because it's clearly unique, relatively narrow (16 bytes wide), and likely to be static. However, as a clustering key, a GUID is appropriate only when it follows an ever-increasing pattern. Some GUIDs—depending on how there are generated—may cause a tremendous amount of fragmentation. If the GUID is generated outside SQL Server (like in a client application) or generated inside SQL Server using the *NEWID()* function, then fragmentation reduces the effectiveness of this column as a clustering key. If possible, consider using the *NEWSEQUENTIALID()* function instead (for ever-increasing GUIDs) or choosing a different clustering key. If you still want to use a GUID as a primary key and it's not ever-increasing, you can make it a nonclustered index instead of a clustered index.

In summary, there are no absolutes to choosing a clustering key; there are only general best practices which work well for most tables. However, if a table has only one index—and no nonclustered indexes—then the nonclustered index dependencies on the clustering key are no longer relevant and a clustered index can take any form. However, most tables are likely to have at least a few nonclustered indexes, and most tables perform better with a clustered index. Because this is the case, a clustered index with a well-chosen clustering key is always the first step to better performance. The second step is "finding the right balance" in your nonclustered indexes by choosing appropriate—and usually a relatively minimal number of—nonclustered indexes.

Nonclustered Indexes

As shown earlier, there are two primary components of all indexes—the leaf level and the non-leaf level(s). For a clustered index, the leaf level *is* the data. For a nonclustered index, the leaf level is a separate and additional structure that has a copy of some of the data. Specifically, a nonclustered index depends on its definition to form the leaf level. The leaf level of a nonclustered index consists of the index key (as per the definition of the index), any included columns (using the *INCLUDE* feature added in SQL Server 2005), and the data row's bookmark value (either the clustering key if the table is clustered or the row's physical RID if the table is a heap). A nonclustered index has exactly the same number of rows as there are rows in the table, unless a filter predicate is used when the index is defined. Filtered indexes are new in SQL Server 2008 and are discussed in more detail later in this chapter.

In terms of *how* the nonclustered index is used, there are really two ways—either to help point to the data (similar to an index in the back of a book, using bookmark lookups, as discussed earlier) or to answer a query directly. When a nonclustered index has all the data as requested by the query, this is known as *query covering,* and the index is called a *covering index*. When a nonclustered index covers a query, the nonclustered index can be used to answer a query directly and a bookmark lookup (which can be expensive for a nonselective query) can be avoided. This can be one of the most

effective ways to improve range query performance.

The bookmark lookup of a row occurs when a nonclustered index does not have all the data required by the query but the query is driven by a predicate that the index can help to find. If a table has a clustered index, the nonclustered index is used to drive the query to find the corresponding data row by using the clustering key. If the table is a heap (in other words, it has no clustered index), the lookup value is an 8-btye RID, which is an actual row locator in the form *FileID*:*PageID*:*SlotNumber*. This 8-byte row identifier breaks down into 2 bytes for the *FileID,* 4 bytes for the *PageID,* and 2 bytes for the *SlotNumber.* We will see exactly how these lookup values are used when we review data access later in this chapter.

The presence or absence of a nonclustered index doesn't affect how the data pages are organized, so you're not restricted to having only one nonclustered index per table, as is the case with clustered indexes. In SQL Server 2008, each table can include as many as 999 nonclustered indexes (up from 249 in SQL Server 2005), but you'll usually want to have far fewer than that (with a few exceptions, such as filtered indexes).

In summary, nonclustered indexes do not affect the base table; however, the base table's structure—either a heap or a table with a clustered index—affects the structure of your nonclustered indexes. This is something to consider and understand if you want to minimize wasted overhead and achieve the best performance.

Constraints and Indexes

As mentioned earlier, an unsuspecting database developer might have a clustered index unintentionally due to having created a PRIMARY KEY constraint on their table. The idea for using constraints comes from relational theory, where a table has entity identifiers defined (to understand table relationships and help to join tables in a normalized schema). When constraints are defined on a table in SQL Server, both PRIMARY KEY and UNIQUE KEY constraints can enforce certain aspects of entity integrity within the database.

For a PRIMARY KEY constraint, SQL Server enforces two things: first, that all the columns involved in the PRIMARY KEY do not allow NULL values, and second, that the PRIMARY KEY value is unique within the table. If any of the columns allow NULL values, the PRIMARY KEY constraint cannot be created. To enforce uniqueness, SQL Server creates a UNIQUE index on the columns that make up the PRIMARY KEY constraint. The default index type, if not specified, is a unique clustered index.

For a UNIQUE constraint, SQL Server allows the columns that make up the UNIQUE constraint to allow NULLs, but it does not allow all columns to be NULL for more than one row. To enforce uniqueness for the UNIQUE constraint, SQL Server creates a unique index on the columns that make up the constraint. The default index type, if not specified, is a unique nonclustered index.

When you declare a PRIMARY KEY or UNIQUE constraint, the underlying index structure that is created is the same as if you had used the *CREATE INDEX* command directly. However, there are some differences in terms of usage and features. For example, a constraint-based index cannot have other features added (such as included columns or filters, features that are discussed later in this chapter), but a UNIQUE index can have these features while still enforcing uniqueness over the key definition of the index. And when referencing a UNIQUE index—which does not support a constraint—you cannot reference indexes with filters. However, an index that doesn't use filters or an index that uses included columns can be referenced. These can be powerful options to use to minimize the total number of indexes and yet still create a reference with a FOREIGN KEY constraint.

The names of the indexes that are built to support these constraints are the same as the constraint names. In terms of internal storage and how these indexes work, there is no difference between unique indexes created using the *CREATE INDEX* command and indexes created to support constraints. The Query Optimizer makes decisions based on the presence of the unique index rather than on whether the column was declared as a constraint or not. In other words, *how* the index was created is irrelevant to the Query Optimizer.

Index Creation Options

In terms of creating indexes, the *CREATE INDEX* command is relatively straightforward:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
   ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
```

The required parts of an index are the index name, the key definition, and the table on which this index is defined. An index can have non-key columns included in the leaf level of the index, using *INCLUDE.* An index can be defined over the entire rowset of the table—which is the default—or, new in SQL Server 2008, can be limited to only the rows as defined by a filter, using `WHERE <filter_predicate>.` We discuss both of these as we analyze the physical structures of nonclustered indexes.

However, *CREATE INDEX* has some additional options available for specialized purposes. You can add a WITH clause to the *CREATE INDEX* command:

```
[WITH
([FILLFACTOR = fillfactor]
[[,] [PAD_INDEX] = { ON | OFF }]
[[,] DROP_EXISTING  = { ON | OFF }]
[[,] IGNORE_DUP_KEY = { ON | OFF }]
[[,] SORT_IN_TEMPDB = { ON | OFF }]
[[,] STATISTICS_NORECOMPUTE  = { ON | OFF }]
[[,] ALLOW_ROW_LOCKS = { ON | OFF }]
[[,] ALLOW_PAGE_LOCKS = { ON | OFF }]
[[,] MAXDOP = max_degree_of_parallelism]
[[,] ONLINE = { ON | OFF }] )]
```

The FILLFACTOR, PAD_INDEX, DROP_EXISTING, SORT_IN_TEMPDB, and ONLINE index creation options are predominantly defined and used for index maintenance. To use them appropriately, you must better understand the physical structures of indexes as well as how data modifications work. We cover these options in detail in the section entitled “Managing Index Structures,” later in this chapter. The remaining options are described here.

IGNORE_DUP_KEY

You can ensure the uniqueness of an index key by defining it as UNIQUE or by defining a PRIMARY KEY or UNIQUE constraint. If an *UPDATE* or *INSERT* statement would affect multiple rows, or if even one row is found that would cause duplicates of keys defined as unique, the entire statement is aborted and no rows are affected. Alternatively, when you create a UNIQUE index, you can use the IGNORE_DUP_KEY option so that a duplicate key error on a multiple-row *INSERT* won’t cause the entire statement to be rolled back. The nonunique row is discarded, and all other rows are inserted. IGNORE_DUP_KEY doesn’t allow the uniqueness of the index to be violated; instead, it makes a violation in a multiple-row data modification nonfatal to all the nonviolating rows.

STATISTICS_NORECOMPUTE

The STATISTICS_NORECOMPUTE option determines whether the statistics on the index should be updated automatically. Every index maintains a histogram representing the distribution of values for the leading column of the index. Among other things, the Query Optimizer uses these statistics to determine the usefulness of a particular index when choosing a query plan. As data is modified, the statistics get increasingly out of date, and this can lead to less-than-optimal query plans if the statistics are not updated. In Chapter 3, “Databases and Database Files,” you learned about the database option AUTO_UPDATE_STATISTICS, which enables all statistics in a database to be updated automatically when needed. In general, the database option should be enabled. However, if desired, a specific statistic or index can be set to not update automatically, using the STATISTICS_NORECOMPUTE option. Adding this clause overrides an *ON* value for the AUTO_UPDATE_STATISTICS database option. If the database option is set to *OFF,* you cannot override that behavior for a particular index, and in that case, all statistics in the database must be updated manually using *UPDATE STATISTICS* or *sp_updatestats*. To see if the statistics for a given table are set to auto-update, as well as the last time they were updated, use *sp_autostats <table_name>*.

MAXDOP

The option MAXDOP controls the maximum number of processors that can be used for index creation. It can override the server configuration option *max degree of parallelism* for index building. Allowing multiple processors to be used for index creation can greatly enhance the performance of index build operations. As with other parallel operations, the Query Optimizer determines at run time the actual number of processors to use, based on the current load on the system. The MAXDOP value just sets a maximum. Multiple processors can be used for index creation only when you run SQL Server Enterprise or SQL Server Developer editions. And, when used, each processor builds an equal-sized chunk of the index in parallel. When this occurs, the tree might not be perfectly balanced, and the math that’s used to determine the theoretical minimum number of required pages differs from the actual number, as each parallel thread builds a separate tree. Once each of the threads have completed, the trees are essentially concatenated together. SQL Server can use any extra page

space that's reserved during this parallel process for later modifications.

Index Placement

A final clause in the *CREATE INDEX* command allows you to specify the placement of the index:

```
[ ON { partition_scheme_name ( column_name )
       | filegroup_name } ]
```

You can specify that an index should either be placed on a particular filegroup or partitioned according to a predefined partition scheme. By default, if no filegroup or partition scheme is specified, the index is placed on the same filegroup as the base table. We discussed filegroups in Chapter 3 and you will learn about table and index partitioning in Chapter 7, "Special Storage."

Constraints and Indexes

The issue of whether a unique index should be defined using a UNIQUE or PRIMARY KEY constraint or through the *CREATE INDEX* command is a common concern and a frequent source of confusion. As mentioned earlier, there is no internal difference in structure, or in the Query Optimizer's choices, for a unique clustered index built using the *CREATE INDEX* command or one that was built to support a PRIMARY KEY constraint. The difference is really a design issue, so it is beyond the scope of this book, which deals with SQL Server internals. However, one simple distinction can be made; a constraint is a logical construct and an index is a physical one. When you build an index, you are asking SQL Server to create a physical structure that takes up storage space and must be maintained during data modifications. When you define a constraint, you are defining a property of your data and expecting SQL Server to enforce that property, but you are not telling SQL Server *how* to enforce it. In the current version, SQL Server enforces PRIMARY KEY and UNIQUE constraints by creating unique indexes, but there is no requirement that it do so. In a future release, SQL Server could enforce this through another mechanism, although it is unlikely to do so because of backward compatibility issues.

Physical Index Structures

Index pages have almost the same structure as data pages except that they store index records instead of data records. As with all other types of pages in SQL Server, index pages use a fixed size of 8 KB, or 8,192 bytes. Index pages also have a 96-byte header, and there is an offset array at the end of the page with 2 bytes for each row to indicate the offset of that row on the page. A nonclustered index can have all three allocation units associated with it: IN_ROW_DATA, ROW_OVERFLOW_DATA, and LOB_DATA. Each index has a row in the *sys.indexes* catalog view, with an *index_id* value of either 1 (for a clustered index) or a number between 2 and 250 or between 256 and 1005 (indicating a nonclustered index). Remember that SQL Server has reserved values between 251 and 255.

Index Row Formats

Index rows are structured just like data rows, with two main exceptions. First, an index row cannot have *SPARSE* columns. If a *SPARSE* column is used in an index definition (and there are some limitations as to where a *SPARSE* column can be used in indexes, such as that it cannot be used in a PRIMARY KEY), then the column is created in the index row as if it had not been defined as *SPARSE*. Second, if a clustered index is created and the index is not defined as unique, then the duplicate key values include a uniquifier.

There are a couple of other differences in structure between index and data rows. An index row does not use the *TagB* or *Fsize* row header values. In place of the *Fsize* field, which indicates where the fixed-length portion of a row ends, the page header *pminlen* value is used to decode an index row. The *pminlen* value indicates the offset at which the fixed-length data portion of the row ends. If the index row has no variable-length or nullable columns, that is the end of the row. Only if the index row has nullable columns are the field called *NcoI* and the null bitmap both present. The *NcoI* field contains a value indicating how many columns are in the index row; this value is needed to determine how many bits are in the null bitmap. Data rows have an *NcoI* field and null bitmap whether or not any columns allow NULL, but index rows have only a null bitmap and an *NcoI* field if NULLs are allowed in any of the columns of the index. Table 6-2 shows the general format of an index row.

Table 6-2: Information Stored in an Index Row

Information	Mnemonic	Size
Status Bits A	*TagA* Some of the relevant bits are:	1 byte

	▪ Bits 1 through 3: Taken as a 3-bit value. 0 indicates a primary record. 3 indicates an index record. 5 indicates a ghost index record. (Ghost records are discussed later in this chapter.) ▪ Bit 4: Indicates that a NULL bitmap exists. ▪ Bit 5: Indicates that variable-length columns exist in the row.	
Fixed-length data	*Fdata*	*pminlen*—1
Number of columns	*Ncol*	2 bytes
NULL bitmap (1 bit for each column in the table; a 1 indicates that the corresponding column is NULL)	*Nullbits*	Ceiling (*Ncol* / 8)
Number of variable-length columns; only present if > 0	*VarCount*	2 bytes
Variable column offset array; only present if *VarCount* > 0	*VarOffset*	2 * *VarCount*
Variable-length data, if any	*VarData*	

The specific column data stored in an index row depends on the type of index and the level in which that index row is located.

Clustered Index Structures

The leaf level of a clustered index is the data itself. When a clustered index is created, the data is physically copied and ordered based on the clustering key (as discussed earlier in this chapter). There is no difference between the row structure of a clustered index and the row structure of a heap, except in one case: when the clustering key has not been defined with the UNIQUE attribute. In this case, SQL Server must guarantee uniqueness internally, and to do this, each duplicate row requires an additional uniquifier value.

Clustered Index Rows with a Uniquifier

As mentioned earlier, if your clustered index was not created with the *UNIQUE* property, SQL Server adds a 4-byte integer to make each nonunique key value unique. Because the clustering key is used to identify the base rows being referenced by nonclustered indexes (the bookmark lookup), there needs to be a unique way to refer to each row in a clustered index.

SQL Server adds the uniquifier only when necessary—that is, when duplicate keys are added to the table. As an example, we create a small table with all fixed-length columns and then add a clustered, nonunique index to the table:

```
USE AdventureWorks2008;
GO

CREATE TABLE Clustered_Dupes
  (Col1 CHAR(5)   NOT NULL,
   Col2 INT       NOT NULL,
   Col3 CHAR(3)   NULL,
   Col4 CHAR(6)   NOT NULL);
GO

CREATE CLUSTERED INDEX Cl_dupes_col1 ON Clustered_Dupes(col1);
```

If you look at the row in the *sysindexes* compatibility view for this table, you notice something unexpected:

```
SELECT indid, keycnt, name FROM sysindexes
WHERE id = OBJECT_ID ('Clustered_Dupes');

RESULT:
indid  keycnt name
------ ------ --------------
```

```
1       2       Cl_dupes_col1
```

The column called *keycnt*, which indicates the number of keys an index has, has a value of 2. (Note that this column is available only in the compatibility view *sysindexes*, not in the catalog view *sys.indexes*.) If this index had the *UNIQUE* property, the *keycnt* value would be 1. Because creating a clustered index on a nonunique key is not recommended—it wastes time and space with the process of making rows unique—we'll skip a full analysis of this structure. However, there is a script named ExaminingtheClusteredIndexUniquifier. sql included with this chapter's resource materials in the companion content (which can be found at *http://www.SQLServerInternals.com/companion*). The script creates and analyzes the clustered index row structure when the clustering key is not defined as UNIQUE.

The Non-Leaf Level(s) of a Clustered Index

To navigate to the leaf level of an index, a B-tree is created, which includes the data rows in the leaf level. Each row in the non-leaf levels has one entry for every page of the level below (later in this chapter, we look more into what this specifically looks like with each index type) and this entry includes an index key value and a 6-byte pointer to reference the page. In this case, the page pointer is in the format of 2 bytes for the *FileID* and 4 bytes for the *PageNumberInTheFile.* SQL Server does not need an 8-byte RID because the slot number does not need to be stored. The index key part of the entry always indicates the minimum value that could be on the pointed-to page. Note that they do not necessarily indicate the *actual* lowest value, just the lowest *possible* value for the page (as when the row with the lowest key value on a page is deleted, the index row in the level above is not updated).

Analyzing a Clustered Index Structure

To better illustrate how the clustered index is stored as well as traversed, we review specific structures created in a sample database called *IndexInternals.* For this example, we review an *Employee* table created with a clustered index on the PRIMARY KEY.

> **Note** The *IndexInternals* sample database is available for download. A few tables exist in this database already. Review the *EmployeeCaseStudy-TableDefinition.sql* script to see the table definitions, and then move to the *EmployeeCaseStudy-AnalyzeStructures.sql* script to analyze the structures. A backup of this database and a zip file containing the solution can be found in the companion content.

Here is the table definition for the *Employee* table, as it already exists within the *IndexInternals* database:

```
CREATE TABLE Employee
  (EmployeeID      INT         NOT NULL     IDENTITY,
   LastName        NCHAR(30)   NOT NULL,
   FirstName       NCHAR(29)   NOT NULL,
   MiddleInitial   NCHAR(1)    NULL,
   SSN             CHAR(11)    NOT NULL,
   OtherColumns    CHAR(258)   NOT NULL     DEFAULT 'Junk');
GO
```

The *Employee* table was created using a few deviations from normal best practices to make the structures somewhat predictable (for example, easier math and easier visualization). First, all columns have fixed widths even if when data values vary. Not all columns should be variable just because the data values vary, but when your column is over 20 characters and your data varies (and is not overly volatile), then it's best to consider variable-width character columns rather than fixed-width columns, to save space and for better *INSERT* performance. (*UPDATE* performance may be compromised, especially when updates make the variable-width column larger.) We discuss this in more detail in the section on fragmentation later in this chapter. In these specific tables, fixed-width columns are used to ensure a predictable row size and to help in better visualizing the data structures.

In this case, and including overhead, the data rows of the *Employee* table are exactly 400 bytes per row (using a filler column called *OtherColumns,* which adds 258 bytes of junk at the end of the data row). A row size of 400 bytes means that we can fit 20 rows per data page (8,096 bytes per page/400 bytes per row = 20.24, which translates into 20 rows per page because the IN_ROW portion of the data row cannot span pages). To calculate how large our tables are, we need to know how many rows these tables contain. And, in the *IndexInternals* database, this table has already been set up with exactly 80,000 rows. At 20 rows per page, this table requires 4,000 data pages to store its 80,000 rows.

In the current table definition, this table is a heap. For the *Employee* table, we define the clustered index by using a PRIMARY KEY constraint:

```
-- Add the CLUSTERED PRIMARY KEY for Employee
```

```
ALTER TABLE Employee
  ADD CONSTRAINT EmployeePK
    PRIMARY KEY CLUSTERED (EmployeeID);
GO
```

To investigate our *Employee* table further, we use *sys.dm_db_index_physical_stats* to determine the number of pages within the table, as well as the number of levels within our indexes. We can confirm the index structures using the DMV to see the number of levels as well as the number of pages within each level:

```
SELECT index_depth AS D
    , index_level AS L
    , record_count AS 'Count'
    , page_count AS PgCnt
    , avg_page_space_used_in_percent AS 'PgPercentFull'
    , min_record_size_in_bytes AS 'MinLen'
    , max_record_size_in_bytes AS 'MaxLen'
    , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
    (DB_ID ('IndexInternals')
    , OBJECT_ID ('IndexInternals.dbo.Employee')
    , 1, NULL, 'DETAILED');
GO

RESULT:
D   L   Count  PgCnt   PgPercentFull      MinLen  MaxLen  AvgLen
--- --- ------ ------- ------------------ ------- ------- ------
3   0   80000  4000    99.3081294786261   400     400     400
3   1   4000   7       91.7540400296516   11      11      11
3   2   7      1       1.09957993575488   11      11      11
```

The clustered index for this table has a leaf level of 4,000 pages, which is as expected, given that we have 80,000 rows at 20 rows per page. From the *MinLen* (*min_record_size_in_bytes*) column, we can see our row length in the leaf level is 400 bytes; however, the row length of the non-leaf levels is only 11 bytes. This structure is easily broken down as 4 bytes for the integer column (*EmployeeID*) on which the clustered index is defined, 6 bytes for our page pointer, and 1 byte for row overhead. Only 1 byte is needed for overhead because our index row contains only fixed-width columns and none of those columns allow NULLs (therefore, we do not need a NULL bitmap in the index pages). In addition, you can see that there are 4,000 rows in the first level above the leaf level because level 1 has a *Count* (*record_count*) of 4,000. In fact, in level 1 there are only seven pages [shown as *PgCnt* (*page_count*)], and in level 2, you can see that *Count* shows as 7. This refers back to earlier in this chapter, when we explained that each level up the tree contains a pointer for every page of the level below it. If a level has 4,000 pages, then the next level up has 4,000 rows. You can see a more detailed version of this structure in Figure 6-2.

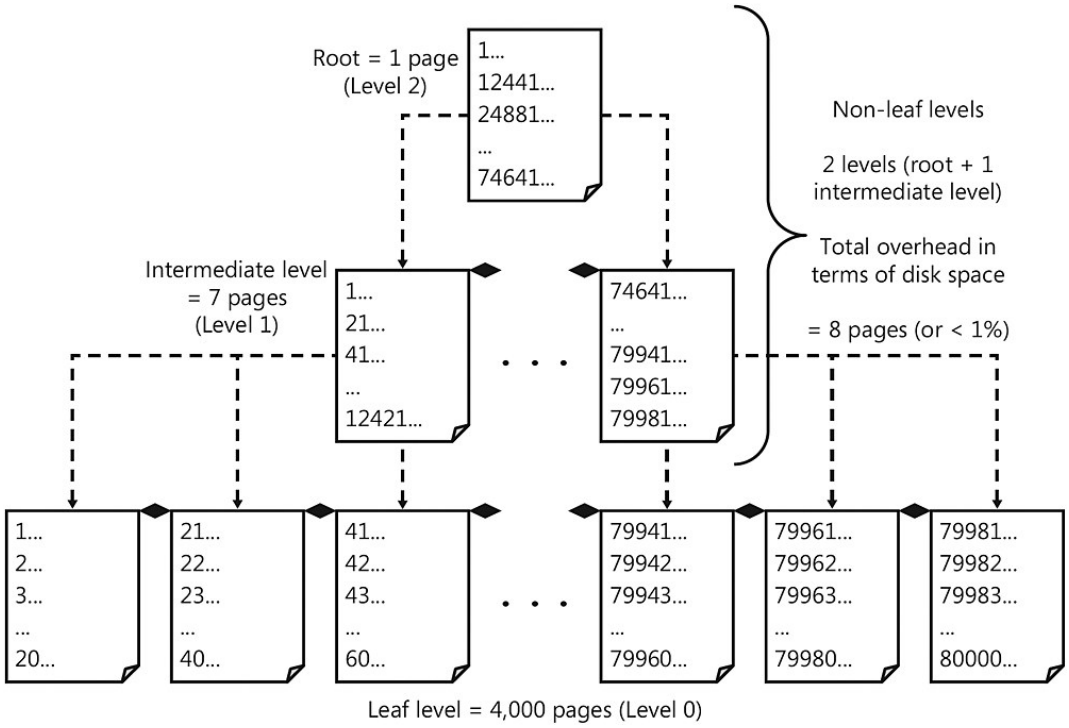

Figure 6-2: Page details for multiple index levels

To understand both traversal as well as linkage further, you can use the *DBCC IND* command to see which pages have which data, as well as which pages precede and follow various pages in all levels of the index. In this case, we insert the results of *DBCC IND* into our *sp_ tablepages* table in the *master* database so that we can access (and order) only specific columns of information:

```
TRUNCATE TABLE sp_tablepages;
INSERT sp_tablepages
        EXEC ('DBCC IND (IndexInternals, Employee, 1)');
GO

SELECT IndexLevel
    , PageFID
    , PagePID
    , PrevPageFID
    , PrevPagePID
    , NextPageFID
    , NextPagePID
FROM sp_tablepages
ORDER BY IndexLevel DESC, PrevPagePID;
GO

RESULT (abbreviated):
IndexLevel PageFID PagePID     PrevPageFID PrevPagePID NextPageFID NextPagePID
---------- ------- ----------- ----------- ----------- ----------- -----------
2          1       234         0           0           0           0

1          1       232         0           0           1           233
1          1       233         1           232         1           235
1          1       235         1           233         1           236
1          1       236         1           235         1           237
1          1       237         1           236         1           238
1          1       238         1           237         1           239
1          1       239         1           238         0           0

0          1       168         0           0           1           169
0          1       169         1           168         1           170
<snip>
0          1       4230        1           4229        1           4231
0          1       4231        1           4230        0           0
```

```
NULL       1       157         0           0           0           0
```

Because this table was created when the database was empty and because the clustered index was built after loading the data into a staging area (this is solely a separate location used for temporarily storing data—in this case, a different filegroup), this table's clustered index was able to use a completely contiguous range of pages within file 1. However, they are not completely contiguous from the root down because indexes are built from the leaf level up to the root as the rows are ordered for each of the levels. The most important thing to understand, however, is navigation. Imagine the following query:

```
SELECT e.*
FROM dbo.Employee AS e
WHERE e.EmployeeID = 27682;
```

To find all the data for a row with an *EmployeeID* of 27682 (remember, this is the clustering key value), SQL Server starts at the root page and navigates down to the leaf level. Based on the output shown previously, the root page is page 234 in *FileID* 1 (you can see this because the root page is the only page at the highest index level (*IndexLevel* = 2). To analyze the root page, we use *DBCC PAGE* with output style 3—and we make sure that the query window in SQL Server Management Studio is set to return grid results. The reason for this is that when using output style 3, the tabular set of a non-leaf page is returned to the grid results, separating the rows from the page header, which is returned to the messages window:

```
DBCC PAGE (IndexInternals, 1, 234, 3);
GO

RESULT:
FileId PageId  Row     Level ChildFileId ChildPageId EmployeeID (key) KeyHashValue
------ ------- ------- ----- ----------- ----------- ---------------- ------------
1      234     0       2     1           232         NULL             NULL
1      234     1       2     1           233         12441            NULL
1      234     2       2     1           235         24881            NULL
1      234     3       2     1           236         37321            NULL
1      234     4       2     1           237         49761            NULL
1      234     5       2     1           238         62201            NULL
1      234     6       2     1           239         74641            NULL
```

Reviewing the output from *DBCC PAGE* for the root page, we can see the *EmployeeID* values at the start of each "child page" in the *EmployeeID* (key) column. And because these are based on ordered rows in the level below, we solely need to find the appropriate range. For the third page, you can see a low value of 24881, and for the fourth page, a low value of 37321. So if the value 27682 exists, it would have to be in the index area defined by this particular range. For navigational purposes, we must navigate down the tree using page (*ChildPageId*) 235 in *FileID* (*ChildFileId*) 1. To see this page's contents, we can again use *DBCC PAGE* with output style 3:

```
DBCC PAGE (IndexInternals, 1, 235, 3);
GO

RESULT (abbreviated):
FileId  PageId Row     Level ChildFileId ChildPageId EmployeeID (key) KeyHashValue
------- ------ ------- ----- ----------- ----------- ---------------- ------------
1       235    0       1     1           1476        24881            NULL
...
1       235    139     1     1           1615        27661            NULL
1       235    140     1     1           1616        27681            NULL
1       235    141     1     1           1617        27701            NULL
...
1       235    621     1     1           2097        3730             NULL
```

Finally, if a row with an *EmployeeID* of 27682 exists, it must be on page 1,616 of *FileID* 1. Let's see if it is:

```
DBCC TRACEON(3604);
GO
DBCC PAGE (IndexInternals, 1, 1616, 3);
GO

...

Slot 1 Column 1 Offset 0x4 Length 4 Length (physical) 4
EmployeeID = 27682
```

```
Slot 1 Column 2 Offset 0x8 Length 60 Length (physical) 60
LastName = Arbariol

Slot 1 Column 3 Offset 0x44 Length 58 Length (physical) 58
FirstName = Burt

Slot 1 Column 4 Offset 0x7e Length 2 Length (physical) 2
MiddleInitial = R

Slot 1 Column 5 Offset 0x80 Length 11 Length (physical) 11
SSN = 373-00-8368

Slot 1 Column 6 Offset 0x8b Length 258 Length (physical) 258
OtherColumns = Junk
...
```

Note *DBCC PAGE* returns all the details for the page; that is, the header and all data rows. In this condensed output, we see only the converted row values from output style 3 for our *EmployeeID* value of interest (27682). The header and all other rows have been removed.

By having traversed the structure for a row, we have reviewed two things—the index internals and the process by which a single data row can be found using a clustering key value. This method is used when performing a bookmark lookup from a nonclustered index to retrieve the data when the table is clustered. To understand fully how nonclustered indexes are used, we also need to know how a nonclustered index is stored and how it is traversed to get to the data.

Nonclustered Index Structures

The contents of the leaf level of a nonclustered index depend on many factors: the definition of the nonclustered index key, the base table's structure (either a heap or a clustered index), the existence of any nonclustered index features such as included columns or filtered indexes, and finally, whether or not the nonclustered index is defined as unique.

To best understand nonclustered indexes, we continue using our *IndexInternals* database. This time, however, we review nonclustered indexes on two tables: the *Employee* table, which is clustered by the PRIMARY KEY constraint on the *EmployeeID* column, and the *EmployeeHeap* table, which does not have a clustered index. The *EmployeeHeap* table is an exact copy of the *Employee* table; however, it uses a nonclustered PRIMARY KEY constraint on the *EmployeeID* column instead of a clustered one. This is the first structure we review.

Nonclustered Index Rows on a Heap

The *EmployeeHeap* table has exactly the same definition and data as the *Employee* table used in the prior example. Here is the *EmployeeHeap* table definition:

```
CREATE TABLE EmployeeHeap
  (EmployeeID      INT         NOT NULL     IDENTITY,
   LastName        NCHAR(30)   NOT NULL,
   FirstName       NCHAR(29)   NOT NULL,
   MiddleInitial   NCHAR(1)    NULL,
   SSN             CHAR(11)    NOT NULL,
   OtherColumns    CHAR(258)   NOT NULL     DEFAULT 'Junk');
GO
```

As with the *Employee* table, the data rows of the *EmployeeHeap* table are exactly 400 bytes per row and with 80,000 rows, this table also requires 4,000 data pages. To see the physical size of the data, you can use the *sys.dm_db_index_physical_stats* DMV discussed at the beginning of this chapter. We can confirm that this table is exactly the same (in terms of data) as the leaf level of the clustered index by using the DMV to see the number of pages, as well as the row length for the index with an *index_id* of 0 (the third parameter to the DMV):

```
SELECT index_depth AS D
    , index_level AS L
    , record_count AS 'Count'
    , page_count AS PgCnt
    , avg_page_space_used_in_percent AS 'PgPercentFull'
    , min_record_size_in_bytes AS 'MinLen'
    , max_record_size_in_bytes AS 'MaxLen'
    , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
```

```
    (DB_ID ('IndexInternals')
    , OBJECT_ID ('IndexInternals.dbo.EmployeeHeap')
    , 0, NULL, 'DETAILED');
GO

RESULT:
D   L   Count  PgCnt   PgPercentFull      MinLen  MaxLen  AvgLen
--- --- ------ ------- ------------------ ------- ------- ------
1   0   80000  4000    99.3081294786261   400     400     400
```

For the *EmployeeHeap* table, all the constraints are going to be created using nonclustered indexes. The following commands create the PRIMARY KEY as a nonclustered index on the *EmployeeID* column and a UNIQUE KEY as a nonclustered index on the *SSN* column:

```
-- Add a NONCLUSTERED PRIMARY KEY for EmployeeHeap
ALTER TABLE EmployeeHeap
  ADD CONSTRAINT EmployeeHeapPK
    PRIMARY KEY NONCLUSTERED (EmployeeID);
GO

-- Add the NONCLUSTERED UNIQUE KEY on SSN for EmployeeHeap
ALTER TABLE EmployeeHeap
  ADD CONSTRAINT SSNHeapUK
    UNIQUE NONCLUSTERED (SSN);
GO
```

To determine what's in the leaf level of a nonclustered index built on a heap, we first review the structure of the nonclustered index as shown by the DMV. For nonclustered indexes, we supply the specific index ID for parameter 3. To see the index ID assigned, we can use a query against *sys.indexes:*

```
SELECT index_depth AS D
    , index_level AS L
    , record_count AS 'Count'
    , page_count AS PgCnt
    , avg_page_space_used_in_percent AS 'PgPercentFull'
    , min_record_size_in_bytes AS 'MinLen'
    , max_record_size_in_bytes AS 'MaxLen'
    , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
    (DB_ID ('IndexInternals')
    , OBJECT_ID ('IndexInternals.dbo.EmployeeHeap')
    , 2, NULL, 'DETAILED');
GO

RESULT:
D   L   Count  PgCnt   PgPercentFull      MinLen  MaxLen  AvgLen
--- --- ------ ------- ------------------ ------- ------- ------
2   0   80000  149     99.477291821102    13      13      13
2   1   149    1       23.9065974796145   11      11      11
```

In this case, the leaf level of the nonclustered index (level 0) shows a record count of 80,000 (based on the fact that there are 80,000 rows in the table) and a minimum, maximum, and average length of 13 (these are fixed-width index rows). This breaks down very clearly and easily—the nonclustered index is defined on the *EmployeeID* column (an integer of 4 bytes); the table is a heap so the data row's bookmark (the physical RID) is 8 bytes; and because this is a fixed-width row with no columns that allow NULL values, the row overhead is 1 byte (4 + 8 + 1 = 13 bytes). To see the data stored more specifically, we can use *DBCC IND* to review the leaf-level pages of this index:

```
TRUNCATE TABLE sp_tablepages;
INSERT sp_tablepages
        EXEC ('DBCC IND (IndexInternals, EmployeeHeap, 2)');
GO

SELECT IndexLevel
    , PageFID
    , PagePID
    , PrevPageFID
    , PrevPagePID
    , NextPageFID
    , NextPagePID
```

```
FROM sp_tablepages
ORDER BY IndexLevel DESC, PrevPagePID;
GO

RESULT (abbreviated):
IndexLevel PageFID PagePID     PrevPageFID PrevPagePID NextPageFID NextPagePID
---------- ------- ----------- ----------- ----------- ----------- -----------
1          1       8608        0           0           0           0

0          1       8544        0           0           1           8545
0          1       8545        1           8544        1           8546
...
0          1       8755        1           8754        1           8756
0          1       8756        1           8755        0           0

NULL       1       254         0           0           0           0
```

The root page is on page 8608 of *FileID* 1. Leaf-level pages are labeled with an *IndexLevel* of 0, so the first page of the leaf level is on page 8544 of *FileID* 1. To review the data on this page, we can use *DBCC PAGE* with output style 3. (The output for this leaf-level index page shows only the first 8 rows and the last 3 rows, out of a total of 539 rows.)

```
DBCC PAGE (IndexInternals, 1, 8544, 3);
GO

RESULT (abbreviated):
FileId PageId   Row    Level  EmployeeID (key) HEAP RID            KeyHashValue
------ -------- ------ ------ ---------------- -------------------- ----------------
1      8544     0      0      1                0xF500000001000000  (010086470766)
1      8544     1      0      2                0xF500000001000100  (020068e8b274)
1      8544     2      0      3                0xF500000001000200  (03000d8f0ecc)
1      8544     3      0      4                0xF500000001000300  (0400b4b7d951)
1      8544     4      0      5                0xF500000001000400  (0500d1d065e9)
1      8544     5      0      6                0xF500000001000500  (06003f7fd0fb)
1      8544     6      0      7                0xF500000001000600  (07005a186c43)
1      8544     7      0      8                0xF500000001000700  (08000c080f1b)
...
1      8544     536    0      537              0xD211000001001000  (190098ec2ef0)
1      8544     537    0      538              0xD211000001001100  (1a0076439be2)
1      8544     538    0      539              0xD211000001001200  (1b001324275a)
```

From the output of *DBCC PAGE*, you can see that the leaf-level page of a nonclustered index on a heap has the index key column value (in this case, the *EmployeeID*), plus the actual data row's RID. The final value displayed is called *KeyHashValue*, which is not actually stored in the index row. It is a fixed-length string derived using a hash formula on all the key columns. This value is used to represent the row in certain other tools. One such tool that is discussed in Chapter 10 is the *sys.dm_tran_locks* DMV that shows the locks that are being held. When a lock is held on an index row, the list of locks displays *KeyHashValue* to indicate which key (or index row) is locked.

The RID can be converted to the *FileID:PageID:SlotNumber* format by using the following function:

```
CREATE FUNCTION convert_RIDs (@rid BINARY(8))
    RETURNS VARCHAR(30)
AS
    BEGIN
    RETURN (
        CONVERT (VARCHAR(5),
            CONVERT(INT, SUBSTRING(@rid, 6, 1)
            + SUBSTRING(@rid, 5, 1)) )
        + ':' +
        CONVERT(VARCHAR(10),
            CONVERT(INT, SUBSTRING(@rid, 4, 1)
            + SUBSTRING(@rid, 3, 1)
            + SUBSTRING(@rid, 2, 1)
            + SUBSTRING(@rid, 1, 1)) )
        + ':' +
        CONVERT(VARCHAR(5),
            CONVERT(INT, SUBSTRING(@rid, 8, 1)
            + SUBSTRING(@rid, 7, 1)) ) )
    END;
GO
```

With this function, you can find out the specific page number on which a row resides. For example, a row with an *EmployeeID* of 6 has a hexadecimal RID of 0xF500000001000500:

```
SELECT dbo.convert_RIDs (0xF500000001000500);
GO

RESULT:
1:245:5
```

Using the function, this converts to 1:245:5, which is comprised of *FileID* 1, *PageID* 245, and *SlotNumber* 5. To view this specific page, we can use *DBCC PAGE* and then review the data on slot 5 (to see if this is in fact the row with *EmployeeID* of 6):

```
DBCC PAGE (IndexInternals, 1, 245, 3);
GO

Slot 5 Column 1 Offset 0x4 Length 4 Length (physical) 4
EmployeeID = 6

Slot 5 Column 2 Offset 0x8 Length 60 Length (physical) 60
LastName = Anderson

Slot 5 Column 3 Offset 0x44 Length 58 Length (physical) 58
FirstName = Dreaxjktgvnhye

Slot 5 Column 4 Offset 0x7e Length 2 Length (physical) 2
MiddleInitial =

Slot 5 Column 5 Offset 0x80 Length 11 Length (physical) 11
SSN = 250-07-9751

Slot 5 Column 6 Offset 0x8b Length 258 Length (physical) 258
OtherColumns = Junk
...
```

In this case, you have seen the structure of a nonclustered index row in the leaf level of the nonclustered index, as well as how a bookmark lookup is performed using the heap's RID from the nonclustered index to the heap.

In terms of navigation, imagine the following query:

```
SELECT e.*
FROM dbo.EmployeeHeap AS e
WHERE e.EmployeeID = 27682;
```

Because this table is a heap, only nonclustered indexes can be used to navigate this data efficiently. And, in this case, we have a nonclustered index on *EmployeeID*. The first step is to go to the root page (as shown in the *DBCC IND* output earlier, the root page is page 8608 of *FileID* 1):

```
DBCC PAGE (IndexInternals, 1, 8608, 3);
GO

RESULT:
FileId  PageId Row     Level ChildFileId ChildPageId EmployeeID (key)  KeyHashValue
------- ------ ------- ----- ----------- ----------- ----------------- ------------
1       8608   0       1     1           8544        NULL              NULL
1       8608   1       1     1           8545        540               NULL
...
1       8608   49      1     1           8593        26412             NULL
1       8608   50      1     1           8594        26951             NULL
1       8608   51      1     1           8595        27490             NULL
1       8608   52      1     1           8596        28029             NULL
1       8608   53      1     1           8597        28568             NULL
1       8608   54      1     1           8598        29107             NULL
...
1       8608   147     1     1           8755        79234             NULL
1       8608   148     1     1           8756        79773             NULL
```

Using the *EmployeeID* column in this output, you can see a low value of 27490 for the child page 8595 in *FileID* 1, and then the next page has a low value of 28029. So if an *EmployeeID* of 27682 exists, it would have to be in the index area

defined by this particular range. Then we must navigate down the tree using page (*ChildPageId*) 8595 in *FileID* (*ChildFileId*) 1. To see this page's contents, we can again use *DBCC PAGE* with output style 3:

```
DBCC PAGE (IndexInternals, 1, 8595, 3);
GO

RESULT:
FileId PageId      Row    Level  EmployeeID (key) HEAP RID           KeyHashValue
------ ----------- ------ ------ ---------------- ------------------ ---------------
1      8595        0      0      27490            0x1617000001000900 (6200aa3b160b)
1      8595        1      0      27491            0x1617000001000A00 (6300cf5caab3)
…
1      8595        191    0      27681            0x2017000001000000 (2100fcdaf887)
1      8595        192    0      27682            0x2017000001000100 (220012754d95)
1      8595        193    0      27683            0x2017000001000200 (23007712f12d)

…
1      8595        538    0      28028            0x3117000001000700   (7c00b4675dbf)
```

Note The output returns 539 rows. In this condensed output, we see the first two rows, the last row, and then three rows surrounding the value of interest (27682).

From this point, you know how the navigation continues. SQL Server translates the data row's RID into the format of *FileID:PageID:SlotNumber* and proceeds to look up the corresponding data row in the heap.

Nonclustered Index Rows on a Clustered Table

For nonclustered indexes on a table that has a clustered index, the leaf-level row structure is similar to that of a nonclustered index on a heap. The leaf level of the nonclustered index contains the index key and the bookmark lookup value (the clustering key). However, if the nonclustered index key has some columns in common with the clustering key, SQL Server stores the common columns only once in the nonclustered index row. For example, if the key of your clustered index is *EmployeeID,* and you have a nonclustered index on (*Lastname*, *EmployeeID, SSN*), then the index rows do not store the value of *EmployeeID* twice. In fact, the number of columns and the column order do not matter. For this example (as it's not generally a good practice to have a wide clustering key), imagine a clustering key that is defined on columns *b, e,* and *h*. The following nonclustered indexes would have these column(s) added to make up the nonclustered index leaf-level rows (the columns—if any—that are added to the leaf level of the nonclustered index, are italicized and bolded):

Nonclustered Index Key	Nonclustered Leaf-Level Row
a	a, ***b, e, h***
c, h, e	c, h, e, ***b***
e	e, ***b, h***
h	h, ***e, b***
b, c, d	b, c, d, ***e, h***

To review the physical structures of a nonclustered index created on a table that is clustered, we review the UNIQUE constraint on the *SSN* column of the *Employee* table:

```
-- Add the NONCLUSTERED UNIQUE KEY on SSN for Employee
ALTER TABLE Employee
    ADD CONSTRAINT EmployeeSSNUK
        UNIQUE NONCLUSTERED (SSN);
GO
```

To gather information on the data size and number of levels, we use the DMV. However, before we can use the DMV, we need the specific index ID for parameter 3. To see the index ID assigned to this nonclustered index, we can use a query against *sys.indexes:*

```
SELECT name AS IndexName, index_id
FROM sys.indexes
WHERE [object_id] = OBJECT_ID ('Employee');
GO

RESULT:
IndexName        index_id
```

```
---------------- --------
EmployeePK       1
EmployeeSSNUK    2

SELECT index_depth AS D
    , index_level AS L
    , record_count AS 'Count'
    , page_count AS PgCnt
    , avg_page_space_used_in_percent AS 'PgPercentFull'
    , min_record_size_in_bytes AS 'MinLen'
    , max_record_size_in_bytes AS 'MaxLen'
    , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
    (DB_ID ('IndexInternals')
    , OBJECT_ID ('IndexInternals.dbo.Employee')
    , 2, NULL, 'DETAILED');
GO

RESULT:
D   L       Count    PgCnt   PgPercentFull    MinLen   MaxLen   AvgLen
--  ------- -------- ------- ---------------- -------- -------  ------
2   0       80000    179     99.3661106992834 16       16       16
2   1       179      1       44.2055843834939 18       18       18
```

In this case, the leaf level of the nonclustered index (level 0) shows a record count of 80,000 (there are 80,000 rows in the table) and a minimum, maximum, and average length of 16 (these are fixed-width index rows). This breaks down very clearly and easily—the nonclustered index is defined on the *SSN* column (a fixed-width character column of 11 bytes), the table has a clustering key of *EmployeeID* so the data row's bookmark (the clustering key) is 4 bytes, and because this row is a fixed-width row with no columns that allow NULL values, the row overhead is 1 byte (11 + 4 + 1 = 16 bytes). To see the data stored more specifically, we can use *DBCC IND* to review the leaf-level pages of this index:

```
TRUNCATE TABLE sp_tablepages;
INSERT sp_tablepages
        EXEC ('DBCC IND (IndexInternals, Employee, 2)');
GO

SELECT IndexLevel
    , PageFID
    , PagePID
    , PrevPageFID
    , PrevPagePID
    , NextPageFID
    , NextPagePID
FROM sp_tablepages
ORDER BY IndexLevel DESC, PrevPagePID;
GO

RESULT (abbreviated):
IndexLevel PageFID PagePID     PrevPageFID PrevPagePID NextPageFID NextPagePID
---------- ------- ----------- ----------- ----------- ----------- -----------
1          1       4328        0           0           0           0

0          1       4264        0           0           1           4265
0          1       4265        1           4264        1           4266
...
0          1       4505        1           4504        1           4506
0          1       4506        1           4505        0           0

NULL       1       158         0           0           0           0
```

The root page is on page 4328 of *FileID* 1. Leaf-level pages are labeled with an *IndexLevel* of 0, so the first page of the leaf level is on page 4264 of *FileID* 1. To review the data on this page, we can use *DBCC PAGE* with format 3:

```
DBCC PAGE (IndexInternals, 1, 4264, 3);
GO

RESULT (abbreviated):
FileId PageId      Row    Level  SSN (key)   EmployeeID  KeyHashValue
------ ----------- ------ ------ ----------- ----------- ----------------
```

```
1       4264        0       0       000-00-0184 31101       (fd00604642ee)
1       4264        1       0       000-00-0236 22669       (fb00de40fee1)
1       4264        2       0       000-00-0395 18705       (0101d993da83)
...
1       4264        446     0       013-00-5906 44969       (ff00355b1727)
1       4264        447     0       013-00-5982 7176        (03012415a3e8)
1       4264        448     0       013-00-6001 11932       (f100f75a17a4)
```

From the output of *DBCC PAGE*, you can see that the leaf-level page of a nonclustered index on a clustered table has actual column values for both the index key (in this case, the *SSN* column) and the data row's bookmark, which in this case is the *EmployeeID*. And this is an actual value, copied into the leaf level of the nonclustered index. Had the clustering key been wider, the leaf level of the nonclustered index would have been wider as well.

In terms of navigation, review the following query:

```
SELECT e.*
FROM dbo.Employee AS e
WHERE e.SSN = '123-45-6789';
```

To find all the data for a row with a *SSN* of 123-45-6789, SQL Server starts at the root page and navigates down to the leaf level. Based on the output shown previously, the root page is in page 4328 of *FileID* 1 (you can see this because the root page is the only page at the highest index level (*IndexLevel* = 1). We could perform the same analysis as before and follow the navigation through the B-tree, but this is left as an exercise for you, if you wish.

Nonunique Nonclustered Index Rows

You now know that the leaf level of a nonclustered index must have a bookmark because from the leaf level, you want to be able to find the actual data row. The non-leaf levels of a nonclustered index need only help us traverse down to pages at the lower levels. In the case of a unique nonclustered index (such as in the previous examples of PRIMARY KEY and UNIQUE constraint indexes), the non-leaf level rows contain only the nonclustered index key values and the child-page pointer. However, if the index is *not* unique, the non-leaf level rows contain the nonclustered index key values, the child-page pointer, and the bookmark value. In other words, the bookmark value is added to the nonclustered index key in a nonunique, nonclustered index to guarantee uniqueness (as the bookmark, by definition, must be unique).

Keep in the mind that for the purposes of creating the index rows, SQL Server doesn't care whether the keys in the nonunique index actually contain duplicates. If the index is not defined to be unique, even if all the values are unique, the non-leaf index rows always contain the bookmark.

You can easily see this by creating the following three indexes to review both their leaf and non-leaf level row sizes:

```
CREATE NONCLUSTERED INDEX TestTreeStructure
ON Employee (SSN);
GO

CREATE UNIQUE NONCLUSTERED INDEX TestTreeStructureUnique1
ON Employee (SSN);
GO

CREATE UNIQUE NONCLUSTERED INDEX TestTreeStructureUnique2
ON Employee (SSN, EmployeeID);
GO

SELECT si.[name] AS iname
    , index_depth AS D
    , index_level AS L
    , record_count AS 'Count'
    , page_count AS PgCnt
    , avg_page_space_used_in_percent AS 'PgPercentFull'
    , min_record_size_in_bytes AS 'MinLen'
    , max_record_size_in_bytes AS 'MaxLen'
    , avg_record_size_in_bytes AS 'AvgLen'
FROM sys.dm_db_index_physical_stats
    (DB_ID ('IndexInternals')
    , OBJECT_ID ('IndexInternals.dbo.Employee')
    , NULL, NULL, 'DETAILED') ps
    INNER JOIN sys.indexes si
        ON ps.[object_id] = si.[object_id]
```

```
            AND ps.[index_id] = si.[index_id]
WHERE ps.[index_id] > 2;
GO

RESULT:
iname                     D L Count PgCnt PgPercentFull    MinLen MaxLen AvgLen
------------------------- - - ----- ----- ---------------- ------ ------ ------
TestTreeStructure         2 0 80000 179   99.3661106992834 16     16     16
TestTreeStructure         2 1 179   1     53.0516431924883 22     22     22
TestTreeStructureUnique1 2 0 80000 179   99.3661106992834 16     16     16
TestTreeStructureUnique1 2 1 179   1     44.2055843834939 18     18     18
TestTreeStructureUnique2 2 0 80000 179   99.3661106992834 16     16     16
TestTreeStructureUnique2 2 1 179   1     53.0516431924883 22     22     22
```

Notice that the leaf level (level 0) of all three indexes is identical in all columns: *Count* (*record_count*), *PgCnt* (*page_count*), *PgPercentFull* (*avg_space_used_in_percent*), and all three length columns. For the non-leaf level of the indexes (which are very small), you can see that the lengths vary—for the first (*TestTreeStructure*) and the third (*TestTreeStructureUnique2*), the non-leaf levels are identical. The first index has the *EmployeeID* added because it's the clustering key (therefore the bookmark). The third index has *EmployeeID* already in the index—there's no need to add it again. However, in the first index, because it was not defined as unique, SQL Server had to add the clustering key all the way up the tree. For the second index—which was unique on *SSN* alone—SQL Server did not include *EmployeeID* all the way up the tree. If you're interested, you can continue to analyze these structures using *DBCC IND* and *DBCC PAGE* to view the physical row structures further.

Nonclustered Index Rows with Included Columns (Using INCLUDE)

In all nonclustered indexes so far, we have focused on the physical aspects of indexes created by constraints or indexes created to test physical structures. Nowhere have we approached the limits of index key size, which are 900 bytes or 16 columns, whichever comes first. The reason that these limits exist is to help to ensure index tree scalability. However, this has also traditionally limited the maximum number of columns that can be indexed.

In some cases, adding columns in an index allows SQL Server to eliminate the bookmark lookup when accessing data for a range query, a concept called *covering indexes*. A covering index is a nonclustered index in which all the information needed to satisfy a query can be found in the leaf level, so SQL Server doesn't have to access the data pages at all. This can be a powerful tool for optimizing some of your more complex range-based queries.

Instead of adding columns to the nonclustered index key, and making the tree deeper, columns for a covering index can be added to the index rows without becoming part of the key using the *INCLUDE* syntax. It is a very simple addition to your *CREATE INDEX* command:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name
  ON table_name (column_name [ASC | DESC][,...n])
     [ INCLUDE ( column_name [ ,...n ] ) ]
```

These columns listed after the keyword *INCLUDE* allow you to exceed the 900-byte or 16-key column limits in the leaf level of a nonclustered index. The included columns appear only in the leaf level and do not affect the sort order of the index rows in any way. In certain situations, SQL Server can silently add an included column to your indexes. This might happen when an index is created on a partitioned table and no ON filegroup or ON *partition_scheme_name* is specified.

Nonclustered Index Rows with Filters (Filtered Indexes)

Without using filters, the leaf level of a nonclustered index contains one index row for every row of data in the table, in logical order based on the index definition. New in SQL Server 2008, you can add a filter predicate to your nonclustered index definition. This allows SQL Server to create nonclustered index rows only for data that matches your predicate, thus limiting the size of the nonclustered index. This can be extremely useful if you have one of the following situations:

- When a column contains mostly NULL values and where queries retrieve only the rows where the data is NOT NULL. This is especially useful when combined with *SPARSE* columns.
- When a column contains only a limited number of interesting values or you want to enforce uniqueness only for a set of values. For example, what if you wanted to allow NULL values for the *SSN* column of the *Employee* table? Using a constraint, SQL Server allows only a single row to be NULL. However, using a filtered index you can create a unique index over only the rows where the *SSN* is not NULL. The syntax would look like the following:

  ```
  CREATE UNIQUE NONCLUSTERED INDEX SSN_NOT_NULLs
  ```

```
ON Employee (SSN)
WHERE SSN IS NOT NULL;
```

- When queries retrieve only a particular range of data and you want to add indexes to this data but not the entire table. For example, you have a table which is partitioned by month and covers three years worth of data (2008, 2007, and 2006) and a team wants to heavily analyze data in the fourth quarter of 2007. Instead of creating wider nonclustered indexes for all your data, you can create indexes (possibly using *INCLUDE* as well) that focus only on:

```
WHERE SalesDate > '20071001' AND SalesDate < '20080101';
```

The end result of an index created with a filter is that the leaf level of the nonclustered index contains a row only if the row matches the filter definition. And the column over which the filter is defined does not need to be in the key, or even in an included column; however, that can help to make the index more useful for certain queries. You can use *DBCC IND, DBCC PAGE,* and, the previously mentioned, DMVs to review the size and structure for indexes with filters.

Special Index Structures

SQL Server 2008 allows you to create several special kinds of indexes: indexes on computed columns, indexes on views, spatial indexes, full-text indexes, and XML indexes. This section covers the requirements and the structural differences of creating these types of indexes.

Indexes on Computed Columns and Indexed Views

Without indexes, some of these constructs—computed columns and views—are purely logical. There is no physical storage for the data involved. A computed column is not stored with the table data; it is recomputed every time a row is accessed (unless the computed column is marked as PERSISTED). A view does not save any data; it basically saves a *SELECT* statement that is executed again every time the data in the view is accessed. With these special indexes, SQL Server actually materializes what was only logical data into the physical leaf level of an index.

Prerequisites

Before you can create indexes on either computed columns or views, certain prerequisites must be met. The biggest issue is that SQL Server must be able to guarantee that given the identical base table data, the same values are always returned for any computed columns or for the rows in a view (that is, the computed columns and views are *deterministic*). To guarantee that the same values are always generated, these special indexes have three categories of requirements. First, a number of session-level options must be set to a specific value. Second, there are some restrictions on the functions that can be used within the computed column or view definition. The third requirement, which applies only to indexed views, is that the tables that the view is based on must meet certain criteria.

SET Options

The following seven SET options can affect the resulting value of an expression or predicate, so you must set them as shown to create indexed views or indexes on computed columns:

```
SET CONCAT_NULL_YIELDS_NULL ON
SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
SET ANSI_PADDING ON
SET ANSI_WARNINGS ON
SET NUMERIC_ROUNDABORT OFF
```

Note that all the options have to be ON except the NUMERIC_ROUNDABORT option, which has to be OFF. Technically, the option ARITHABORT must also be set to ON. And, when your database is set to 90 compatibility mode or higher, setting ANSI_WARNINGS to ON automatically sets ARITHABORT to ON, so you do not need to set it separately. If any of these options are not set as specified, you get an error message when you try to create a special index. In addition, if you've already created one of these indexes, after which you change the SET option settings, and then attempt to modify the computed column or view on which the index is based, you get an error. If you issue a *SELECT* that normally should use the index, and if the SET options do not have the values indicated, the index is ignored but no error is generated.

There are a couple of ways to determine whether the SET options are set appropriately before you create one of these special indexes. You can use the function *SESSIONPROPERTY* to test the settings for your current connection. A returned value of 1 means that the setting is ON, and a 0 means that it is OFF. The following example checks the current session setting for the option NUMERIC_ROUNDABORT:

```
SELECT SESSIONPROPERTY ('NUMERIC_ROUNDABORT');
```

Alternatively, you can use the *sys.dm_exec_sessions* DMV to check the SET options for any connection. The following query returns the values for five of the previously discussed six SET options for the current session:

```
SELECT quoted_identifier, arithabort, ansi_warnings,
       ansi_padding, ansi_nulls, concat_null_yields_null
FROM sys.dm_exec_sessions
WHERE session_id = @@spid;
```

Unfortunately, NUMERIC_ROUNDABORT is not included in the *sys.dm_exec_sessions* DMV results. There is no way to see the setting for that value for any other connections besides the current one.

Permissible Functions

A function is either *deterministic* or *nondeterministic*. If the function returns the same result every time it is called with the same set of input values, it is deterministic. If it can return different results when called with the same set of input values, it is nondeterministic. For the purposes of indexes, a function is considered deterministic if it always returns the same values for the same input values when all the SET options have the required settings. Any function used in a computed column's definition or used in the *SELECT* list or WHERE clause of an indexable view must be deterministic.

> **More Info** *SQL Server Books Online* contains a complete list of which supplied functions are deterministic and which are nondeterministic. Some functions can be either deterministic or nondeterministic, depending on how they are used, and *SQL Server Books Online* also describes these functions.

It might seem that the list of nondeterministic functions is quite restrictive, but SQL Server must be able to guarantee that the values stored in the index are consistent. In some cases, the restrictions might be overly cautious, but the downside of being not cautious enough would be that your indexed views or indexes on computed columns are meaningless. The same restrictions apply to functions you use in your own user-defined functions (UDFs)—that is, your own functions cannot be based on any nondeterministic built-in function. You can verify the determinism property of any function by using the *OBJECTPROPERTY* function:

```
SELECT OBJECTPROPERTY (object_id('<function_name>'), 'IsDeterministic')
```

Even if a function is deterministic, if it contains *float* or *real* expressions, the result of the function might vary with different processors depending on the processor architecture or microcode version. Expressions or functions containing values of the data type *float* or *real* are therefore considered to be *imprecise*. To guarantee consistent values even when moving a database from one machine to another (by detaching and attaching, or by performing backup and restore), imprecise values can be used only in key columns of indexes if they are physically stored in the database and not recomputed. An imprecise value can be used if it is the value of a stored column in a table or if it is a computed column that is marked as persisted. We discuss persisted columns in more detail in the upcoming section entitled "Indexes on Computed Columns."

Schema Binding

To create an indexed view, a requirement on the table itself is that the definition of any underlying object's schema cannot change. To prevent a change in schema definition, the *CREATE VIEW* statement allows the WITH SCHEMABINDING option. When you specify WITH SCHEMABINDING, the *SELECT* statement that defines the view must include the two-part names (*schema.object*) of all referenced tables. You can't drop the table or alter the columns that participate in a view created with the WITH SCHEMABINDING clause unless you've dropped that view or changed the view so that it's no longer schemabound. Otherwise, SQL Server raises an error. If any of the tables on which the view is based are owned by someone other than the user creating the view, the view creator doesn't automatically have the right to create the view with schema binding because that would restrict the table's owner from making changes to her own table. A user must be granted REFERENCES permission on a table to create a view with schema binding on that table. We will see an example of schema binding in a moment.

Indexes on Computed Columns

SQL Server 2008 allows you to build indexes on deterministic, precise (and persisted imprecise) computed columns where the resulting data type is otherwise indexable. This means that the column's data type cannot be any of the LOB data types (such as *text*, *varchar(max)*, or *XML*). Such a computed column can be an index key, included column, or part of a PRIMARY KEY or UNIQUE constraint. You cannot define a FOREIGN KEY, CHECK, or DEFAULT constraint on a computed column, and computed columns are always considered nullable unless you enclose the expression in the *ISNULL* function. When you create an index on computed columns, the six previously mentioned SET options must first have the correct values set.

Here's an example:

```
CREATE TABLE t1 (a INT, b as 2*a);
GO
CREATE INDEX i1 ON t1 (b);
GO
```

If any of your SET options does not have the correct value when you create the table, you get this message when you try to create the index:

```
Server: Msg 1935, Level 16, State 1, Line 2
Cannot create index. Object '<tname>' was created with the following SET options off:
    '<option(s)>'.
```

If more than one option has an incorrect value, the error message reports them all.

Here's an example that creates a table with a nondeterministic computed column:

```
CREATE TABLE t2 (a INT, b DATETIME, c AS DATENAME(MM, b));
GO
CREATE INDEX i2 ON t2 (c);
GO
```

When you try to create the index on the computed column *c*, you get this error:

```
Msg 2729, Level 16, State 1, Line 1
Column 'c' in table 't2' cannot be used in an index or statistics or as a partition key
    because it is nondeterministic.
```

Column *c* is nondeterministic because the month value of *DATENAME()* can have different values depending on the language you're using.

Using the COLUMNPROPERTY Function You can use the *IsDeterministic* column property to determine before you create an index on a computed column (or on a view) whether that column is deterministic. If you specify this property, the *COLUMNPROPERTY* function returns 1 if the column is deterministic and 0 otherwise. The result is undefined for columns that are neither computed columns nor columns in a view, so you should consider checking the *IsComputed* property before you check the *IsDeterministic* property. The following example detects that column *c* in table *t2* in the previous example is nondeterministic:

```
SELECT COLUMNPROPERTY (OBJECT_ID('t2'), 'c', 'IsDeterministic');
```

The value 0 is returned, which means that column *c* is nondeterministic. Note that the *COLUMNPROPERTY* function requires an object ID for the first argument and a column name for the second argument.

However, the *COLUMNPROPERTY* function also has a property of *IsIndexable*. That's probably the easiest to use for a quick check, but it won't give you the reason if the column is not indexable. For that, you should check these other properties.

Implementation of a Computed Column

If you create a clustered index on a computed column, the computed column is no longer a virtual column in the table. The computed value physically exists in the rows of the table, which is the leaf level of the clustered index. Updates to the columns that the computed column is based on also update the computed column in the table itself. For example, in the *t1* table created previously, if we insert a row with the value 10 in column *a*, the row is created with both the values 10 and 20 in the actual data row. If we then update the 10 to 15, the second column is updated automatically to 30.

Persisted Columns The ability to mark a computed column as PERSISTED (a feature introduced in SQL Server 2005) allows storage of computed values in a table, even before you build an index. In fact, this feature was added to the product to allow columns of computed values from underlying table columns of type *float* or *real* to have indexes built on them. The alternative, when you want an index on such a column, would be to drop and re-create the underlying column, which can involve an enormous amount of overhead on a large table.

Here's an example. In the *Northwind* database, the *Order Details* table has a column called *Discount* that is of type *real*. The following code adds a computed column called *Final* that shows the total price for an item after the discount is applied. The statement to build an index on *Final* fails because the resultant column involving the *real* value is imprecise and not persisted:

```
USE Northwind;
GO
```

```
ALTER TABLE [Order Details]
   ADD Final AS
    (Quantity * UnitPrice) - Discount * (Quantity * UnitPrice);
GO
CREATE INDEX OD_Final_Index on [Order Details](Final);
GO

Error Message:
Msg 2799, Level 16, State 1, Line 1
Cannot create index or statistics 'OD_Final_Index' on table 'Order Details'
    because the computed column 'Final' is imprecise and not persisted. Consider removing
    column from index or statistics key or marking computed column persisted.
```

Without persisted computed columns, the only way to create an index on a computed column containing the final price would be to drop the *Discount* column from the table and redefine it. Any existing indexes on *Discount* would have to be dropped as well, and then rebuilt. With persisted computed columns, all you need to do is drop the computed column (which is a metadata-only operation) and then redefine it as a persisted computed column. You can then build the index on the column:

```
ALTER TABLE [Order Details]
   DROP COLUMN Final;
GO
ALTER TABLE [Order Details]
   ADD Final AS
   (Quantity * UnitPrice) - Discount * (Quantity * UnitPrice) PERSISTED;
GO
CREATE INDEX OD_Final_Index on [Order Details](Final);
```

When determining whether you have to use the PERSISTED option, use the *COLUMNPROPERTY* function and the *IsPrecise* property to determine whether a deterministic column is precise:

```
SELECT COLUMNPROPERTY (OBJECT_ID ('Order Details'), 'Final', 'IsPrecise');
```

You can also use persisted computed columns when you define partitions. A computed column that is used as the partitioning column must be explicitly marked as PERSISTED, whether it is precise or imprecise. We look at partitioning in Chapter 7.

Indexed Views

Indexed views in SQL Server are similar to what other products call *materialized views*. One of the most important benefits of indexed views is the ability to materialize summary aggregates of large tables. For example, consider a customer table containing rows for several million U.S.-based customers, from which you want information regarding customers in each state. You can create a view based on a *GROUP BY* query, grouping by state and containing the count of orders per state. Normal views are only named, saved queries and do not store the results. Every time the view is referenced, the aggregation to produce the grouped results must be recomputed. When you create an index on the view, the aggregated data is stored in the leaf level of the index. So instead of millions of customer rows, your indexed view has only 50 rows—one for each state. Your aggregate reporting queries can then be processed using the indexed views without having to scan the underlying, large tables.

The first index you must build on a view is a clustered index, and because the clustered index contains all the data at its leaf level, this index actually does the materialization of the view. The view's data is physically stored at the leaf level of the clustered index.

Additional Requirements

In addition to the requirement that all functions used in the view must be deterministic, and that the required SET options must be set to the appropriate values, the view definition can't contain any of the following:

- *TOP*
- LOB columns
- *DISTINCT*
- *MIN*, *MAX*, *COUNT(*)*, *COUNT(<expression>)*, *STDEV*, *VARIANCE*, *AVG*
- *SUM* on a nullable expression

- A derived table
- The *ROWSET* function
- Another view (you can reference only base tables)
- *UNION*
- Subqueries, OUTER joins, or self-joins
- Full-text predicates (*CONTAINS*, *FREETEXT*)
- *COMPUTE*, *COMPUTE BY*
- *ORDER BY*

Also, if the view definition contains *GROUP BY,* the *SELECT* list must include the aggregate *COUNT_BIG (*).* *COUNT_BIG* returns a *BIGINT*, which is an 8-byte integer. A view that contains *GROUP BY* can't contain *HAVING, CUBE, ROLLUP,* or *GROUP BY ALL.* Also, all *GROUP BY* columns must appear in the *SELECT* list. Note that if your view contains both *SUM* and *COUNT_BIG (*)*, you can compute the equivalent of the *AVG* aggregate function even though *AVG* is not allowed in indexed views. Although these restrictions might seem severe, remember that they apply to the view definitions, not to the queries that might use the indexed views.

To verify that you've met all the requirements, you can use the *OBJECTPROPERTY* function's *IsIndexable* property. The following query tells you whether you can build an index on a view called *Product Totals*:

```
SELECT OBJECTPROPERTY (OBJECT_ID ('Product_Totals'), 'IsIndexable');
```

A return value of 1 means you've met all requirements and can build an index on the view.

Creating an Indexed View

The first step in building an index on a view is to create the view itself. Here's an example from the *AdventureWorks2008* database:

```
USE AdventureWorks2008;
GO
CREATE VIEW Vdiscount1
WITH SCHEMABINDING
AS SELECT SUM (UnitPrice*OrderQty) AS SumPrice
   , SUM (UnitPrice * OrderQty * (1.00 - UnitPriceDiscount))
         AS SumDiscountPrice
   , COUNT_BIG (*) AS Count
   , ProductID
FROM Sales.SalesOrderDetail
GROUP BY ProductID;
```

Notice the WITH SCHEMABINDING clause and the specification of the schema name (*Sales*) for the table. At this point, we have a normal view—a stored *SELECT* statement that uses no storage space. In fact, if we look at the data in *sys.dm_db_partition_stats* for this view, we see that no rows are returned:

```
SELECT si.name AS index_name,
     ps.used_page_count, ps.reserved_page_count, ps.row_count
FROM sys.dm_db_partition_stats AS ps
    JOIN sys.indexes AS si
        ON ps.[object_id] = si.[object_id]
WHERE ps.[object_id] = OBJECT_ID ('dbo.Vdiscount1');
```

To create an indexed view, you must first create a *unique clustered index*. The clustered index on a view contains all the data that makes up the view definition. This statement defines a unique clustered index for the view:

```
CREATE UNIQUE CLUSTERED INDEX VDiscount_Idx ON Vdiscount1 (ProductID);
```

Once the indexed view has been created, re-run the previous *SELECT* statement to see the pages materialized by the index on the view.

```
RESULT:
index_name     used_page_count      reserved_page_count  row_count
```

```
------------- ------------------- ------------------- ---------
VDiscountIdx   4                   4                   266
```

Data that comprises the indexed view is persistent, with the indexed view storing the data in the clustered index's leaf level. You could construct something similar by using temporary tables to store the data you're interested in. But a temporary table is static and doesn't reflect changes to underlying data. In contrast, SQL Server automatically maintains indexed views, updating information stored in the clustered index whenever anyone changes data that affects the view.

After you create the unique clustered index, you can create multiple nonclustered indexes on the view. You can determine whether a view is indexed by using the *OBJECTPROPERTY* function's *IsIndexed* property. For the *Vdiscount1* indexed view, the following statement returns a 1, which means the view is indexed:

```
SELECT OBJECTPROPERTY (OBJECT_ID ('Vdiscount1'), 'IsIndexed');
```

Once a view is indexed, metadata about space usage and location is available through the catalog views, just as for any other index.

Using an Indexed View

One of the most valuable benefits of indexed views is that your queries don't have to reference a view directly to use the index on the view. Consider the *Vdiscount1* indexed view. Suppose that you issue the following *SELECT* statement:

```
SELECT ProductID, total_sales = SUM (UnitPrice * OrderQty)
FROM  Sales.SalesOrderDetail
GROUP BY ProductID;
```

The Query Optimizer recognizes that the precomputed sums of all the *UnitPrice * OrderQty* values for each *ProductID* are already available in the index for the *Vdiscount1* view. The Query Optimizer evaluates the cost of using that indexed view in processing the query, and the indexed view very likely is used to access the information required to satisfy this query—the *Sales.SalesOrderDetail* table might never be touched at all.

> **Note** Although you can create indexed views in any edition of SQL Server 2008, for the Query Optimizer to consider using them even when they aren't referenced in the query, the engine edition of your SQL Server 2008 must be Enterprise, Developer, or Evaluation.

Just because you have an indexed view doesn't mean the Query Optimizer will always choose it for the query's execution plan. In fact, even if you reference the indexed view directly in the FROM clause, the Query Optimizer might access the base table directly instead. To make sure that an indexed view in your FROM clause is not expanded into its underlying *SELECT* statement, you can use the NOEXPAND hint in the FROM clause. Some of the internals of index selection, query optimization, and indexed view usage are discussed in more detail in Chapter 8.

Full-Text Indexes

Full-text indexes are special-purpose indexes that support the full-text search feature—the ability to search efficiently through character and binary columns in a table. The specifics of creating and using full-text indexes are beyond the scope of this book, but *SQL Server Books Online* has a comprehensive section describing full-text indexing.

Full-text indexes are inverted, stacked, and compressed indexes that are stored in the database in internal tables for convenience. The full-text index data is stored in regular index rows in the internal tables, but the majority of the row is opaque to everything except the full-text engine itself (tools like *DBCC PAGE* cannot properly crack open all fields in the rows).

The storage for full-text indexes is the same as for regular indexes, but as they are stored as internal tables, regular methods of finding their structures do not work. For instance, the *HumanResources.JobCandidate* table in the *AdventureWorks2008* database has a full-text index. To find the object IDs of the internal table(s) in which the full-text index is stored, the following T-SQL can be used to query the *sys.internal_tables* catalog view:

```
USE AdventureWorks2008;
GO
SELECT [name], [object_id] FROM sys.internal_tables
WHERE parent_object_id = OBJECT_ID ('HumanResources.JobCandidate');
GO

RESULT:
name                                          object_id
--------------------------------------------- -----------
```

```
fulltext_index_docidstatus_1333579789          2046630334
fulltext_docidfilter_1333579789                2062630391
fulltext_indexeddocid_1333579789               2078630448
fulltext_avdl_1333579789                       2094630505
```

The regular methods for examining index structures can then be employed, using the object ID returned from *sys.internal_tables.* The same method works for spatial indexes and XML indexes, described next.

As you can see, internal tables have a different root in the system catalogs compared to regular tables and indexes, although their space usage is tracked in exactly the same way (using IAM pages) and their structures are the same as regular indexes. The *SQL Server Books Online* section “Internal Tables” contains a detailed explanation of them, including an entity-relationship diagram of the relevant system catalogs and various queries to view information about them.

Spatial Indexes

A spatial index contains a decomposed view of all values in a spatial data type column in a table. The decomposed values are used for fuzzy-pruning of matching values during spatial comparison operations. As for full-text indexes, the specifics of creating and using spatial indexes are beyond the scope of this book, but *SQL Server Books Online* has an excellent section describing them. See the topic “Spatial Indexing Overview.”

A spatial index is a clustered index that is stored as an internal table. Apart from storing the decomposed spatial values, it has exactly the same structure as a regular index.

XML Indexes

An XML index provides an efficient mechanism for searching XML BLOB values by storing a shredded representation of the XML data that can be searched with regular B-tree methods, instead of having to walk through a (potentially large) XML BLOB. As with full-text and spatial indexes, the specifics of creating and using XML indexes are beyond the scope of this book, but *SQL Server Books Online* has an excellent section describing them. See the topic “Indexes on XML Data Types.” There are two types of XML indexes; primary XML indexes and secondary XML indexes. A primary XML index is a shredded representation of each value in the XML column being indexed, with one row for each node in the XML BLOB. A primary XML index is a clustered index and is stored as an internal table. A secondary XML index is a nonclustered index on the primary XML index and provides the same function as a regular nonclustered index; a different access path to the data using a different sort order. The internal structures of these indexes are the same as those for regular indexes.

Data Modification Internals

We’ve seen how SQL Server stores data and index information. Now we look at what SQL Server actually does internally when your data is modified. We’ve seen how clustered indexes define logical order to your data and how a heap is nothing more than a collection of unordered pages. We’ve seen how nonclustered indexes are structures stored separately from the data and how that data is a copy of the actual table’s data, defined by the index definition. And, as a rule of thumb, you should always have a clustered index on a table. The SQL Customer Advisory Team published a white paper in mid-2007 that compares various table structures and essentially supports this view; see *http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/ clusivsh.mspx*. In this section, we review how SQL Server deals with the existence of indexes when processing data modification statements.

Note that for every *INSERT, UPDATE,* and *DELETE* operation on a table, the equivalent operation also happens to every nonclustered index on the table. The mechanisms described in this section apply equally to clustered and nonclustered indexes. Any modifications to the table are made to the heap or clustered index first, then to each nonclustered index in turn.

In SQL Server 2008, the exception to this rule is filtered indexes, where the filter predicate means the filtered nonclustered index may not have a matching row for the table row being modified. When changes are made to the table, the filtered index predicate is evaluated to determine whether it is necessary to apply the same operation to the filtered nonclustered index.

Inserting Rows

When inserting a new row into a table, SQL Server must determine where to put the data, as well as insert a corresponding row into each nonclustered index. Each operation follows the same pattern: modify the appropriate data page (based on whether or not the table has a clustered index) and then insert the corresponding index rows into the leaf

level of each nonclustered index.

When a table has no clustered index—that is, when the table is a heap—a new row is always inserted wherever room is available in the table. In Chapter 3, you learned how IAMs keep track of which extents in a file already belong to a table and in Chapter 5, you saw how the PFS pages indicate which of the pages in those extents have available space. If no pages with space are available, SQL Server tries to find unallocated pages from existing uniform extents that already belong to the object. If none exists, SQL Server must allocate a whole new extent to the table. Chapter 3 discussed how the Global Allocated Maps (GAMs) and Shared Global Allocation Maps (SGAMs) are used to find extents available to be allocated to an object. So, although locating space in which to do an *INSERT* is relatively efficient using the PFS and IAM, because the location of a row (on *INSERT*) is not defined, determining where to place a row within a heap is usually less efficient than if the table has a clustered index.

For an *INSERT* into a table with a clustered index and for index rows being inserted into nonclustered indexes, the row (regardless of whether it's a data row or an index row) always has a specific location within the index where it must be inserted, based on the value the new row has for the index key columns. An *INSERT* occurs either when the new row is the direct result of an *INSERT* or when it's the result of an *UPDATE* statement that either causes the row to move or for an index key column to change. When a row has to move to a new page, the *UPDATE* statement is internally executed using a *DELETE* followed by an *INSERT* (the *DELETE/ INSERT* strategy). New rows are inserted based on their index key position, and SQL Server splices in a new page via a page split if the current leaf level (a data page if this is the clustered index or an index page if this is a nonclustered index) has no room. Because the index dictates a particular ordering for the rows in the leaf level of the index, every new row has a specific location where it belongs. If there's no room for the new row on the page where it belongs, a new page must be allocated and linked into the B-tree. If possible, this new page is allocated from the same extent as the other pages to which it is linked. If the extent is already full (which is usually the case), a new extent (eight pages or 64 KB) is allocated to the object. As described in Chapter 3, SQL Server uses the GAM pages to find an available extent.

Splitting Pages

After SQL Server finds the new page, the original page must be split; half the rows (the first half based on the slot array on the page) are left on the original page, and the other half are moved to the new page (or as close to a 50/50 split as possible). In some cases, SQL Server finds that even after the split, there's not enough room for the new row, which, because of variable-length fields, could potentially be much larger than any of the existing rows on the pages. As part of the split, SQL Server must add a corresponding entry for every new page into the parent page of the level above. One row is added if only a single split is needed. However, if the new row still won't fit after a single split, there can be potentially multiple new pages and multiple additions to the parent page. For example, consider a page with 32 rows on it. Suppose that SQL Server tries to insert a new row with 8,000 bytes. It splits the page once, and the new 8,000-byte row won't fit. Even after a second split, the new row won't fit. Eventually, SQL Server recognizes that the new row cannot fit on a page with any other rows, and it allocates a new page to hold only the new row. Quite a few splits occur, resulting in many new pages, and many new rows on the parent page.

An index tree is always searched from the root down, so during an *INSERT* operation, it is split on the way down. This means that while the index is being searched on an *INSERT,* the index is protected in anticipation of possibly being updated. The protection mechanism is a latch, which you can think of as something like a lock. (Locks are discussed in detail in Chapter 10.)

A latch is acquired while a page is being read from or written to disk and protects the physical integrity of the contents of the page. A parent node is latched (and protected) until the child node's needed split(s) are complete and no further updates to the parent node are required from the current operation. Then the parent latch can be released safely.

Before the latch on a parent node is released, SQL Server determines whether the page can accommodate another two rows; if not, it splits the page. This occurs only if the page is being searched with the objective of adding a row to the index. The goal is to ensure that the parent page always has room for the row or rows that result from a child page splitting. (Occasionally, this results in pages being split that don't need to be—at least not yet. In the long run, it's a performance optimization that helps to minimize deadlocks in an index and allows for free space to be added for future rows that may require it.) The type of split depends on the type of page being split: a root page of an index, an intermediate index page, or a leaf-level page. And, when a split occurs, it is committed independently of the transaction that caused the page to split (using special internal transactions called *system transactions*). Therefore, even if the *INSERT* transaction is rolled back, the split is not rolled back.

Splitting the Root Page of an Index

If the root page of an index needs to be split for a new index row to be inserted, two new pages are allocated to the index. All the rows from the root are split between these two new pages, and the new index row is inserted into the appropriate place on one of these pages. The original root page is still the root, but now it has only two rows on it, pointing to each of the newly allocated pages. Keeping the original root page means that an update to the index metadata in the system catalogs (that contains a pointer to the index root page) is avoided. A root page split creates a new level in the index. Because indexes are usually only a few levels deep and typically very scalable, this type of split doesn't occur often.

Splitting an Intermediate Index Page

An intermediate index page split is accomplished simply by locating the midpoint of the index keys on the page, allocating a new page, and then copying the lower half of the old index page into the new page. A new row is added to the index page in the level above the page that split, corresponding to the newly added page. Again, this doesn't occur often, although it's much more common than splitting the root page.

Splitting a Leaf-Level Page

A leaf-level page split is the most interesting and potentially common case, and it's probably the only split that you, as a developer or DBA, should be concerned with. The mechanism is the same for splitting clustered index data pages or nonclustered index leaf-level index pages.

Data pages split only under *INSERT* activity and only when a clustered index exists on the table. Although splits are caused only by *INSERT* activity, that activity can be a result of an *UPDATE* statement, not just an *INSERT* statement. As you're about to learn, if the row can't be updated in place or at least on the same page, the update is performed as a *DELETE* of the original row followed by an *INSERT* of the new version of the row. The insertion of the new row can cause a page to split.

Splitting a leaf-level (data or index) page is a complicated operation. Much like an intermediate index page split, it's accomplished by locating the midpoint of the index keys on the page, allocating a new page, and then copying half of the old page into the new page. It requires that the index manager determine the page on which to locate the new row and then handle large rows that don't fit on either the old page or the new page. When a data page is split, the clustered index key values don't change, so the nonclustered indexes aren't affected.

Let's look at what happens to a page when it splits. The following script creates a table with large rows—so large, in fact, that only five rows fit on a page. Once the table is created and populated with five rows, we find its first (and only, in this case) page by inserting the output of *DBCC IND* in the *sp_tablepages* table, finding the information for the data page with no previous page, and then using *DBCC PAGE* to look at the contents of the page. Because we don't need to see all 8,020 bytes of data on the page, we look at only the slot array at the end of the page and then see what happens to those rows when we insert a sixth row:

```
USE AdventureWorks2008;
GO

DROP TABLE bigrows;
GO

CREATE TABLE bigrows
(
    a int  primary key,
    b varchar(1600)
);
GO

/* Insert five rows into the table */
INSERT INTO bigrows
    VALUES (5, REPLICATE('a', 1600));
INSERT INTO bigrows
    VALUES (10, replicate('b', 1600));
INSERT INTO bigrows
    VALUES (15, replicate('c', 1600));
INSERT INTO bigrows
    VALUES (20, replicate('d', 1600));
INSERT INTO bigrows
    VALUES (25, replicate('e', 1600));
GO
```

```
TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND ( AdventureWorks2008, bigrows, -1)'  );
GO

SELECT PageFID, PagePID
FROM sp_tablepages
WHERE PageType = 1;
GO

RESULTS: (Yours may vary.)
PageFID PagePID
------- -----------
1       742

DBCC TRACEON(3604);
GO
DBCC PAGE(AdventureWorks2008, 1, 742, 1);
GO
```

Here is the slot array from the *DBCC PAGE* output:

```
Row - Offset
4 (0x4) - 6556 (0x199c)
3 (0x3) - 4941 (0x134d)
2 (0x2) - 3326 (0xcfe)
1 (0x1) - 1711 (0x6af)
0 (0x0) - 96 (0x60)
```

Now we insert one more row and look at the slot array again:

```
INSERT INTO bigrows
    VALUES (22, REPLICATE('x', 1600));
GO
DBCC PAGE (AdventureWorks2008, 1, 742, 1);
GO
```

The new page always contains the second half of the rows from the original page, but the new row value may be inserted on either page depending on the value of its index keys. In this example, the new row, with a clustered key value of 22, would have been inserted in the second half of the page. So when this page split occurs, the first three rows stay on page 742, the original page. You can inspect the page header to find the location of the next page, which contains the new row.

The page number is indicated by the *m_nextPage* field. This value is expressed as a *file number:page number* pair, in decimal, so you can easily use it with the *DBCC PAGE* command. In this case, *m_nextPage* returned a value of 1:21912 (nowhere near the current page). Using *DBCC PAGE* for the “next page” shows the rows there:

```
DBCC PAGE (AdventureWorks2008, 1, 21912, 1);
```

Here’s the slot array after the *INSERT* for the second page:

```
Row - Offset
2 (0x2) - 1711 (0x6af)
1 (0x1) - 3326 (0xcfe)
0 (0x0) - 96 (0x60)
```

Note that after the page split, three rows are on the page: the last two original rows, with keys of 20 and 25, and the new row, with a key of 22. If you examine the actual data on the page, you notice that the new row is at slot position 1, even though the row itself is physically the last one on the page. Slot 1 (with value 22) starts at offset 3,326, and slot 2 (with value 25) starts at offset 1,711. The clustered key ordering of the rows is indicated by the slot number of the row, not by the physical position on the page. If a table has a clustered index, the row at slot 1 always has a key value less than the row at slot 2 and greater than the row at slot 0. Only the slot numbers are rearranged, not the data. This is an optimization so that only a small number of offsets are rearranged instead of the entire page’s contents. It is a myth that rows in an index are always stored in the exact same physical order as their keys—in fact, SQL Server can store the rows anywhere on a page so long as the slot array provides the correct logical ordering.

Page splits are expensive operations, involving updates to multiple pages (the page being split, the new page, the page that used to be the *m_nextPage* of the page being split, and the parent page), all of which are fully logged. As such, you want to minimize the frequency of page splits in your production system, especially during peak usage times. You can

avoid negatively affecting performance by minimizing splits. Splits can often be minimized by choosing a better clustering key (one where new rows are inserted at the end of the table, rather than randomly, as with a GUID clustering key) or, especially when splits are caused by update to variable-width columns, by reserving some free space on pages using the FILLFACTOR option when you're creating or rebuilding the indexes. You can use this setting to your advantage during your least busy operational hours by periodically rebuilding (or reorganizing) the indexes with the desired FILLFACTOR. That way, the extra space is available during peak usage times, and you save the overhead of splitting then. The pros and cons of various maintenance options are discussed later in this chapter.

Deleting Rows

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

Deleting Rows from a Heap

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

```
USE AdventureWorks2008;
GO

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

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

TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (AdventureWorks2008, smallrows, -1)'  );

SELECT PageFID, PagePID
FROM sp_tablepages
WHERE PageType = 1;

Results:
PageFID PagePID
------- -----------
1       4536

DBCC TRACEON(3604);
GO
DBCC PAGE(AdventureWorks2008, 1, 4536,1);
```

Here is the output from *DBCC PAGE*:

```
DATA:

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

```
00000010:   20200200 fc††††††††††††††††††††††††††††  ...

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

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

Slot 3, Offset 0x9f, Length 21, DumpStyle BYTE
Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x61D9C09F
00000000:   10001200 04000000 726f7720 34202020 †........row 4
00000010:   20200200 fc††††††††††††††††††††††††††††  ...

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

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

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

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

DBCC PAGE(AdventureWorks2008, 1, 4536, 1);
GO
```

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

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

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

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

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

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

Using *DBCC PAGE* with style 1 on a heap, the row doesn't show up in the page itself—only in the slot array. The slot array at the bottom of the page shows that the third row (at slot 2) is now at offset 0 (which means there really is no row using slot 2), and the row using slot 3 is at its same offset as before the *DELETE.* The data on the page is *not* compacted.

In addition to space on pages not being reclaimed, empty pages in heaps frequently cannot be reclaimed. Even if you delete all the rows from a heap, SQL Server does not mark the empty pages as unallocated, so the space is not available for other objects to use. The catalog view *sys.dm_db_partition_stats* still shows the space as belonging to the heap.

Deleting Rows from a B-tree

In the leaf level of an index, either clustered or nonclustered, rows are marked as *ghost records* when they are deleted. This means that the row stays on the page, but a bit is changed in the row header to indicate that the row is really deleted (a *ghost*). The page header also reflects the number of ghost records on a page. Ghost records are used for several purposes. They can be used to make rollbacks much more efficient—if the row hasn't been removed physically, all SQL Server has to do to roll back a *DELETE* is to change the bit indicating that the row is a ghost. It is also a concurrency optimization for key-range locking (which is discussed in Chapter 10), along with other locking modes. In addition, ghost records are used to support row-level versioning; that topic also is discussed in Chapter 10.

Ghost records are cleaned up sooner or later, depending on the load on your system, and sometimes they can be cleaned up before you have a chance to inspect them. There is a background thread called the *ghost-cleanup thread,* whose job it is to remove ghost records that are no longer needed to support active transactions, or any other feature. In the code shown here, if you perform the *DELETE* and then wait a minute or two to run *DBCC PAGE*, the ghost record might really disappear. That is why we look at the page number for the table before we run the *DELETE*, so we can execute the *DELETE* and the *DBCC PAGE* with a single click from the query window. To guarantee that the ghost record is not cleaned up, we can put the *DELETE* into a user transaction and not commit or roll back the transaction before examining the page. The ghost-cleanup thread does not clean up ghost records that are part of an active transaction. Alternatively, we can use the undocumented trace flag 661 to disable ghost cleanup to ensure consistent results when running tests such as in this script. As usual, keep in mind that undocumented trace flags are not guaranteed to continue to work in any future release or service pack, and no support is available for them. Also, be sure to turn off the trace flag when you're done with your testing. You can also force SQL Server to clean up the ghost records. The procedure *sp_clean_db_free_space* will remove all ghost records from an entire database (as long as they are not part of an uncommitted transaction) and the procedure *sp_clean_db_file_free_space* will do the same for a single file of a database.

The following example builds the same table used in the previous *DELETE* example, but this time, the table has a primary key declared, which means a clustered index is built. The data is the leaf level of the clustered index, so when the row is removed, it is marked as a ghost:

```
USE AdventureWorks2008;
GO
DROP TABLE smallrows;
GO
CREATE TABLE smallrows
(
    a int IDENTITY PRIMARY KEY,
    b char(10)
);
GO
INSERT INTO smallrows
    VALUES ('row 1');
INSERT INTO smallrows
    VALUES ('row 2');
INSERT INTO smallrows
    VALUES ('row 3');
INSERT INTO smallrows
    VALUES ('row 4');
INSERT INTO smallrows
    VALUES ('row 5');
```

```
GO
TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (AdventureWorks2008, smallrows, -1)'  );
SELECT PageFID, PagePID
FROM sp_tablepages
WHERE PageType = 1;

Results:
PageFID PagePID
------- -----------
1       4568

DELETE FROM smallrows
WHERE a = 3;
GO
DBCC TRACEON(3604);
DBCC PAGE(AdventureWorks2008, 1, 4544, 1);
GO
```

Here is the output from *DBCC PAGE*:

```
PAGE HEADER:
Page @0x064AE000
m_pageId = (1:4568)                   m_headerVersion = 1                   m_type = 1
m_typeFlagBits = 0x4                  m_level = 0                           m_flagBits = 0x8000
m_objId (AllocUnitId.idObj) = 172     m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594049200128
Metadata: PartitionId = 72057594043105280                                   Metadata:
    IndexId = 1
Metadata: ObjectId = 1179867270       m_prevPage = (0:0)                    m_nextPage = (0:0)
pminlen = 18                          m_slotCnt = 5                         m_freeCnt = 7981
m_freeData = 201                      m_reservedCnt = 0                     m_lsn = (233:499:2)
m_xactReserved = 0                    m_xdesId = (0:18856)                  m_ghostRecCnt = 1
m_tornBits = 0


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

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

Slot 2, Offset 0x8a, Length 21, DumpStyle BYTE
Record Type = GHOST_DATA_RECORD       Record Attributes =  NULL_BITMAP
Memory Dump @0x61B6C08A
00000000:   1c001200 03000000 726f7720 33202020 †........row 3
00000010:   20200200 fc†††††††††††††††††††††††††  ...

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

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

OFFSET TABLE:
Row - Offset
```

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

Note that the row still shows up in the page itself (using *DBCC PAGE* style 1) because the table has a clustered index. Also, you can experiment using different output styles to see how both a heap and a clustered index work with ghosted records, but you still see empty slots, GHOST_DATA_RECORD types, or both for clarification. The header information for the row shows that this is really a ghost record. The slot array at the bottom of the page shows that the row at slot 2 is still at the same offset and that all rows are in the same location as before the deletion. In addition, the page header gives us a value (*m_ghostRecCnt*) for the number of ghost records in the page. To see the total count of ghost records in a table, you can look at the *sys.dm_db_index_physical_stats* function.

More Info A detailed discussion of the ghost cleanup mechanism and an examination of the transaction logging involved are available at Paul Randal's blog—see the blog post at *http://www.SQLskills.com/BLOGS/PAUL/post/Inside-the- Storage-Engine-Ghost-cleanup-in-depth.aspx.*

Deleting Rows in the Non-Leaf Levels of an Index

When you delete a row from a table, all nonclustered indexes must be maintained because every nonclustered index has a pointer to the row that's now gone. Rows in index non-leaf pages aren't ghosted when deleted, but just as with heap pages, the space isn't compacted until new index rows need space in that page.

Reclaiming Pages

When the last row is deleted from a data page, the entire page is deallocated by the ghost cleanup background thread. The exception is if the table is a heap, as we discussed earlier. (If the page is the only one remaining in the table, it isn't deallocated. A table always contains at least one page, even if it's empty.) Deallocation of a data page results in the deletion of the row in the index page that pointed to the deallocated data page. Non-leaf index pages are deallocated if an index row is deleted (which, again, for an update might occur as part of a *DELETE/INSERT* strategy), leaving only one entry in the index page. That entry is moved to its neighboring page if there is space, and then the empty page is deallocated.

The discussion so far has focused on the page manipulation necessary for deleting a single row. If multiple rows are deleted in a single *DELETE* operation, you must be aware of some other issues. Because the issues of modifying multiple rows in a single query are the same for *INSERT*s, *UPDATE*s, and *DELETE*s, we discuss this issue in its own section, later in this chapter.

Updating Rows

SQL Server updates rows in multiple ways, automatically and invisibly choosing the fastest update strategy for the specific operation. In determining the strategy, SQL Server evaluates the number of rows affected, how the rows are accessed (via a scan or an index retrieval, and via which index), and whether changes to the index keys occur. Updates can happen either in place, by just changing one column's value to a new value in the original row, or as a *DELETE* followed by an *INSERT*. In addition, updates can be managed by the query processor or by the storage engine. In this section, we examine only whether the update happens in place or whether SQL Server treats it as two separate operations: delete the old row and insert a new row. The question of whether the update is controlled by the query processor or the storage engine is actually relevant to all data modification operations (not just updates), so we look at that in a separate section.

Moving Rows

What happens if a table row has to move to a new location? In SQL Server 2008, this can happen because a row with variable-length columns is updated to a new, larger size so that it no longer fits on the original page. It can also happen when the clustered or nonclustered index column(s) change because rows are logically ordered by the index key. For example, if we have a clustered index on *lastname*, a row with a *lastname* value of *Abbot* is stored near the beginning of the table. If the *lastname* value is then updated to *Zappa*, this row has to move to near the end of the table.

Earlier in this chapter, we looked at the structure of indexes and saw that the leaf level of nonclustered indexes contains a row locator, or bookmark, for every single row in the table. If the table has a clustered index, that row locator is the clustering key for that row. So if—and only if—the clustered index key is being updated, modifications are required in every nonclustered index (with the possible exception of filtered nonclustered indexes). Keep this in mind when you decide on

which columns to build your clustered index. It's a great idea to cluster on a nonvolatile column, such as an identity.

If a row moves because it no longer fits on the original page, it still has the same row locator (in other words, the clustering key for the row stays the same), and no nonclustered indexes have to be modified. This is true even if the table is moved to a new physical location (filegroup or partitioning scheme). Nonclustered indexes are updated only if the clustering key changes, and moving the physical location of a table row does not change its clustering key.

In our discussion of index internals, you also saw that if a table has no clustered index (in other words, if it's a heap), the row locator stored in the nonclustered index is actually the physical location of the row. In SQL Server 2008, if a row in a heap moves to a new page, the row leaves a forwarding pointer in the original location. The nonclustered indexes won't need to be changed; they still refer to the original location, and from there, they are directed to the new location. In this case, if the table moves to a new location (filegroup or partitioning scheme), the nonclustered indexes are updated, as the physical location of all records in the heap must change, thus invalidating the prior row locators in the nonclustered indexes.

Let's look at an example. We have created a table a lot like the one we created for doing inserts, but this table has a third column of variable length. After we populate the table with five rows, which fill the page, we update one of the rows to make its third column much longer. The row no longer fits on the original page and has to move. We can then load the output from *DBCC IND* into the *sp_tablepages* table to get the page numbers used by the table:

```
USE AdventureWorks2008;
GO
DROP TABLE bigrows;
GO
CREATE TABLE bigrows
(   a int IDENTITY ,
    b varchar(1600),
    c varchar(1600));
GO
INSERT INTO bigrows
    VALUES (REPLICATE('a', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('b', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('c', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('d', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('e', 1600), '');
GO
UPDATE bigrows
SET c = REPLICATE('x', 1600)
WHERE a = 3;
GO

TRUNCATE TABLE sp_tablepages;
INSERT INTO sp_tablepages
    EXEC ('DBCC IND (AdventureWorks2008, bigrows, -1)'  );
SELECT PageFID, PagePID
FROM sp_tablepages
WHERE PageType = 1;

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

DBCC TRACEON(3604);
GO
DBCC PAGE(AdventureWorks2008, 1, 2252, 1);
GO
```

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

```
Slot 2, Offset 0x1feb, Length 9, DumpStyle BYTE
Record Type = FORWARDING_STUB        Record Attributes =
```

```
Memory Dump @0x61ADDFEB
00000000:   04ea1100 00010000 00††††††††††††††††.........
```

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

Managing Forwarding Pointers

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

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

To get a count of forwarded records in a table, you can look at the output from the *sys.dm_db_ index_physical_stats* function.

Updating in Place

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

In cases where a row can't be updated in place, the cost of a not-in-place update is minimal because of the way the nonclustered indexes are stored and because of the use of forwarding pointers. In fact, you can have an update not in place, for which the row stays on the original page. Updates happen in place if a heap is being updated (and there is enough space on the page) or if a table with a clustered index is updated without any change to the clustering keys. You can also get an update in place if the clustering key changes but the row does not need to move at all. For example, if you have a clustered index on a *lastname* column containing consecutive key values of *Able*, *Becker*, and *Charlie*, you might want to update *Becker* to *Baker*. Because the row stays in the same location even after the clustered index key changes, SQL Server performs this as an update in place. On the other hand, if you update *Able* to *Buchner*, the update cannot occur in place, but the new row might stay on the same page.

Updating Not in Place

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

Table-Level vs. Index-Level Data Modification

We've been discussing only the placement and index manipulation necessary for modifying either a single row or a few rows with no more than a single index. If you are modifying multiple rows in a single operation (*INSERT, UPDATE,* or *DELETE*) or by using *BCP* or the *BULK INSERT* command and the table has multiple indexes, you must be aware of some

other issues. SQL Server 2008 offers two strategies for maintaining all the indexes that belong to a table: table-level modification and index-level modification. The Query Optimizer chooses between them based on its estimate of the anticipated execution costs for each strategy.

Table-level modification is sometimes called *row-at-a-time,* and index-level modification is sometimes called *index-at-a-time*. In table-level modification, all indexes are maintained for each row as that row is modified. If the update stream isn't sorted in any way, SQL Server has to do a lot of random index accesses, one access per index per update row. If the update stream is sorted, it can't be sorted in more than one order, so nonrandom index accesses can occur for at most one index.

In index-level modifications, SQL Server gathers all the rows to be modified and sorts them for each index. In other words, there are as many sort operations as there are indexes. Then, for each index, the updates are merged into the index, and each index page is never accessed more than once, even if multiple updates pertain to a single index leaf page.

Clearly, if the update is small—say, less than a handful of rows—and the table and its indexes are sizable, the Query Optimizer usually considers table-level modification the best choice. Most OLTP operations use table-level modification. On the other hand, if the update is relatively large, table-level modifications require a lot of random I/O operations and might even read and write each leaf page in each index multiple times. In that case, index-level modification offers much better performance. The amount of logging required is the same for both strategies.

You can determine whether your updates were done at the table level or the index level by inspecting the query execution plan. If SQL Server performs the update at the index level, you see a plan produced that contains an *UPDATE* operator for each of the affected indexes. If SQL Server performs the update at the table level, you see only a single *UPDATE* operator in the plan.

Logging

Standard *INSERT, UPDATE,* and *DELETE* statements are always logged to ensure atomicity, and you can't disable logging of these operations. The modification must be known to be safely on disk in the transaction log (write-ahead logging) before the commit of the statement or transaction can be acknowledged to the calling application. Page allocations and deallocations, including those done by *TRUNCATE TABLE,* are also logged. As we saw in Chapter 4, "Logging and Recovery," certain operations can be minimally logged when your database is in the *BULK_LOGGED* recovery mode, but even then, information about allocations and deallocations is written to the log, along with the fact that a minimally logged operation has been executed.

Locking

Any data modification must always be protected with some form of exclusive lock. For the most part, SQL Server makes all the locking decisions internally; a user or programmer doesn't need to request a particular kind of lock. Chapter 10 explains the different types of locks and their compatibility. However, because locking is closely tied to data modification, you should always be aware of the following points:

- Every type of data modification performed in SQL Server requires some form of exclusive lock. For most data modification operations, SQL Server considers row locking as the default, but if many locks are required, SQL Server can lock pages or even the whole table.
- Update locks can be used to signal the intention to do an update, and they are important for avoiding deadlock conditions. But ultimately, the update operation requires that an exclusive lock be performed. The update lock serializes access to ensure that an exclusive lock can be acquired, but the update lock isn't sufficient by itself.
- Exclusive locks must always be held until the end of a transaction in case the transaction needs to be undone (unlike shared locks, which can be released as soon as the scan moves off the page, such as when the *READ COMMITTED* isolation is in effect).
- If a full table scan must be employed to find qualifying rows for an *UPDATE* or a *DELETE,* SQL Server has to inspect every row to determine the row to modify. Other processes that need to find individual rows are blocked even if they ultimately modify different rows. Without inspecting the row, SQL Server has no way of knowing whether the row qualifies for the modification. If you're modifying only a subset of rows in the table, as determined by a WHERE clause, be sure that you have indexes available to allow SQL Server to access the needed rows directly so it doesn't have to scan every row in the table.

Fragmentation

Fragmentation is a general term used to describe various effects that can occur in indexes because of data modifications. There are two general types of fragmentation: internal and external.

Internal fragmentation (often called *physical fragmentation* or *page density*) is where there is wasted space on index pages, both at the leaf and non-leaf levels. This can occur because of any or all of the following:

- Page splits (described earlier) leaving empty space on the page that was split and the newly allocated page
- *DELETE* operations that leave pages less than full
- Row sizes that contribute to under-full pages (for instance, a fixed-width, 5,000-byte data record in a clustered index leads to 3,000 wasted bytes per clustered index data page)

Internal fragmentation means the index is taking more space than necessary, leading to increased disk space usage, more pages to read to process the data, and more memory used to hold the pages in the buffer pool. Sometimes internal fragmentation can be advantageous, as it allows more rows to be inserted on pages without *causing* page splits. Deliberate internal fragmentation can be achieved using the FILLFACTOR and PAD_INDEX options, which are described in the next section.

External fragmentation (often called *logical fragmentation* or *extent fragmentation*) is where the pages or extents comprising the leaf level of a clustered or nonclustered index are not in the most efficient order. The most efficient order is where the logical order of the pages and extents (as defined by the index keys, following the next-page pointers from the page headers) is the same as the physical order of the pages and extents within the data file(s). In other words, the index leaf-level page that has the row with the next index key is also the next physically contiguous page in the data file. This is separate from fragmentation at the *file-system* level, where the actual data files may be comprised of several physical sections.

External fragmentation is caused by page splits and reduces the efficiency of ordered scans of part of a clustered or nonclustered index. The more external fragmentation there is, the less likely it is that the storage engine can perform efficient prereading of the pages necessary for the scan.

The methods of detecting and removing fragmentation are discussed in the next section.

Managing Index Structures

SQL Server maintains your indexes automatically, in terms of making sure the correct rows are there. As you add new rows, it automatically inserts them into the correct position in a table with a clustered index, and it adds new leaf-level rows to your nonclustered indexes that point to the new data rows. When you remove rows, SQL Server automatically deletes the corresponding leaf-level rows from your nonclustered indexes.

So, although your indexes continue to contain all the correct index rows in the B-tree to help SQL Server find the rows you are looking for, you might still occasionally need to perform maintenance operations on your indexes, especially to deal with fragmentation in its various forms. In addition, several properties of indexes can be changed.

Dropping Indexes

One of the biggest differences between managing indexes created using the *CREATE INDEX* command and indexes that support constraints is in how you can drop the index. The *DROP INDEX* command allows you to drop only indexes that were built with the *CREATE INDEX* command. To drop indexes that support constraints, you must use *ALTER TABLE* to drop the constraint. In addition, to drop a PRIMARY KEY or UNIQUE constraint that has any FOREIGN KEY constraints referencing it, you must first drop the FOREIGN KEY constraint. This can leave you with a window of vulnerability if your goal is to drop indexes and immediately rebuild them, perhaps with a new *fillfactor*. Although the FOREIGN KEY constraint is gone, an *INSERT* statement can add a row to the table that violates your referential integrity.

One way to avoid this problem is to use *ALTER INDEX*, which allows you to drop and rebuild one or all of your indexes on a table in a single statement, without requiring the auxiliary step of removing FOREIGN KEY constraints. Alternatively, you can use the *CREATE INDEX* command with the DROP_EXISTING option if you want to rebuild existing indexes without having to drop and re-create them in two steps. Although you can normally use *CREATE INDEX* with DROP_EXISTING to redefine the properties of an index—such as the key columns or included columns, or whether the index is unique—if you

use *CREATE INDEX* with DROP_EXISTING to rebuild an index that supports a constraint, you cannot make these kinds of changes. The index must be re-created with the same columns, in the same order, and the same values for uniqueness and clustering.

ALTER INDEX

SQL Server 2005 introduced the *ALTER INDEX* command to allow you to use a single command to invoke various kinds of index changes that in previous versions required an eclectic collection of different commands, including *sp_indexoption, UPDATE STATISTICS, DBCC DBREINDEX*, and *DBCC INDEXDEFRAG.* Instead of having individual commands or procedures for each different index maintenance activity, they all can be done by using *ALTER INDEX*. For a complete description of all the options to *ALTER INDEX*, see the *SQL Server Books Online* topic "ALTER INDEX."

Basically, you can make four types of changes using *ALTER INDEX*, three of which have corresponding options that you can specify when you create an index using *CREATE INDEX*.

Rebuilding an Index

Rebuilding the index replaces the *DBCC DBREINDEX* command and can be thought of as replacing the DROP_EXISTING option to the *CREATE INDEX* command. However, this option allows indexes to be moved or partitioned, too. A new option allows indexes to be rebuilt online, in the same way you can create indexes online (as we mentioned in the section entitled "Index Creation Options," earlier in this chapter). We discuss online index building and rebuilding shortly.

Disabling an Index

Disabling an index makes it completely unavailable, so it can't be used for finding rows for any operations. Disabling the index also means that it won't be maintained as changes to the data are made. You can disable one index or all indexes with a single command. There is no ENABLE option. Because no maintenance is performed while an index is disabled, indexes must be completely rebuilt to make them useful again. Re-enabling, which can take place either online or offline, is done with the REBUILD option to *ALTER INDEX*. This feature was introduced mainly for the internal purposes of SQL Server when applying upgrades and service packs, but there are a few interesting uses for disabling an index. First, you can use it if you want to ignore the index temporarily for troubleshooting purposes. Second, instead of dropping nonclustered indexes before loading data, you can disable them. However, you cannot disable the clustered index. If you disable the clustered index on a table, the table's data will be unavailable because the leaf level of the clustered index *is* the data. Disabling the clustered index essentially disables the table. However, if your data is going to be loaded in clustered index order (for an ever-increasing clustering key) such that all new data goes to the end of the table, then disabling the nonclustered indexes can help to improve load performance. Once the data has been loaded, then you can rebuild the nonclustered indexes without having to supply the entire index definition. All the metadata has been saved while the index was disabled.

Changing Index Options

Most of the options that you can specify during a *CREATE INDEX* operation can also be specified with the *ALTER INDEX* command. These options are ALLOW_ROW_LOCKS, ALLOW_ PAGE_LOCKS, IGNORE_DUP_KEY, FILLFACTOR, PAD_INDEX, STATISTICS _NORECOMPUTE, MAXP_DOP, and SORT_IN_TEMPDB. IGNORE_DUP_KEY was described in the section entitled "Index Creation Options," earlier in this chapter.

FILLFACTOR and PAD_INDEX FILLFACTOR is probably the most commonly used of these options and lets you reserve some space on each leaf page of an index. In a clustered index, because the leaf level contains the data, you can use FILLFACTOR to control how much space to leave in the table itself. By reserving free space, you can later avoid the need to split pages to make room for a new entry. An important fact about FILLFACTOR is that the value is not maintained; it indicates only how much space is reserved with the existing data at the time the index is built or rebuilt. If you need to, you can use the *ALTER INDEX* command to rebuild the index and reestablish the original FILLFACTOR specified. If you don't specify a new FILLFACTOR when using *ALTER INDEX*, the previously used FILLFACTOR is used.

FILLFACTOR should always be specified on an index-by-index basis. If FILLFACTOR isn't specified, the serverwide default is used. The value is set for the server via the *sp_configure* procedure, with the *fillfactor* option. This configuration value is 0 by default (and is the same as 100), which means that leaf pages of indexes are made as full as possible. It is a best practice *not* to change this serverwide setting. FILLFACTOR applies only to the index's leaf pages.

In specialized and high-use situations, you might want to reserve space in the intermediate index pages to avoid page splits there, too. You can do this by specifying the PAD_INDEX option, which instructs SQL Server to use the same

FILLFACTOR value at all levels of the index. Just as for FILLFACTOR, PAD_INDEX is applicable only when an index is created (or re-created).

When you create a table that includes PRIMARY KEY or UNIQUE constraints, you can specify whether the associated index is clustered or nonclustered, and you can also specify the *fillfactor*. Because the *fillfactor* applies only at the time the index is created, and because there is no data when you first create the table, it might seem that specifying the *fillfactor* at that time is completely useless. However, if you decide to rebuild your indexes after the table is populated and if no new *fillfactor* is specified, the original value is used. You can also specify a *fillfactor* when you use *ALTER TABLE* to add a PRIMARY KEY or UNIQUE constraint to a table; if the table already has data in it, the *fillfactor* value is applied when you build the index to support the new constraint.

DROP_EXISTING The DROP_EXISTING option specifies that a given index should be dropped and rebuilt as a single transaction. This option is particularly useful when you rebuild clustered indexes. Normally, when a developer drops a clustered index, SQL Server must rebuild every nonclustered index to change its bookmarks to RIDs instead of the clustering keys. Then, if a developer builds (or rebuilds) a clustered index, SQL Server must again rebuild all nonclustered indexes to update the bookmarks. The DROP_EXISTING option of the *CREATE INDEX* command allows a clustered index to be rebuilt without having to rebuild the nonclustered indexes twice. If you are creating the index on exactly the same keys that it had previously, the nonclustered indexes do not need to be rebuilt at all. If you are changing the key definition, the nonclustered indexes are rebuilt only once, after the clustered index is rebuilt. Instead of using the DROP_EXISTING option to rebuild an existing index, you can use the *ALTER INDEX* command.

SORT_IN_TEMPDB The SORT_IN_TEMPDB option allows you to control where SQL Server performs the sort operation on the key values needed to build an index. The default is that SQL Server uses space from the filegroup on which the index is to be created. While the index is being built, SQL Server scans the data pages to find the key values and then builds leaf-level index rows in internal sort buffers. When these sort buffers are filled, they are written to disk. If the SORT_IN_TEMPDB option is specified, the sort buffers are allocated from *tempdb*, so much less space is needed in the source database. If you don't specify SORT_IN_TEMPDB, not only does your source database require enough free space for the sort buffers and a copy of the index (or the data, if a clustered index is being built), but the disk heads for the database need to move back and forth between the base table pages and the work area where the sort buffers are stored. If, instead, your *CREATE INDEX* command includes the SORT_IN_TEMPDB option, performance can be greatly improved if your *tempdb* database is on a separate physical disk from the database you're working with. You can optimize head movement because two separate heads read the base table pages and manage the sort buffers. You can speed up index creation even more if your *tempdb* database is on a faster disk than your user database and you use the SORT_IN_TEMPDB option.

Reorganizing an Index

Reorganizing an index is the only change that doesn't have a corresponding option in the *CREATE INDEX* command. The reason for this is that when you create an index, there is nothing to reorganize. The REORGANIZE option replaces the *DBCC INDEXDEFRAG* command and removes some of the fragmentation from an index, but it is not guaranteed to remove all the fragmentation, just as *DBCC INDEXDEFRAG* may not remove all the fragmentation (in spite of its name). Before we discuss removing fragmentation, we must first discuss detecting fragmentation, which we do in the next section.

Detecting Fragmentation

As we've already seen in numerous examples, the output of *sys.dm_db_index_physical_stats* returns a row for each level of an index. However, when a table is partitioned, it effectively treats each partition as a table, so this DMV actually returns a row for each level of each partition of each index. For a small index with only in-row data (no row-overflow or LOB pages) and only the one default partition, we might get only two or three rows back (one for each index level). But if there are multiple partitions and additional allocation units for the row-overflow and LOB data, we might see many more rows. For example, a clustered index on a table containing row-overflow data, built on 11 partitions and being two levels deep, have 33 rows (2 levels × 11 partitions + 11 partitions for the *row_overflow* allocation units) in the fragmentation report returned by *sys.dm_db_index_ physical_stats*.

The section entitled "Tools for Analyzing Indexes," earlier in this chapter, has a comprehensive discussion of the input parameters and the output results, but the following columns give fragmentation information that is not obvious:

- **Forwarded_record_count** Forwarded records (discussed in the section entitled "Data Modification Internals," earlier in this chapter) are possible only in a heap and occur when a row with variable-length columns increases in size due to updates so that it no longer fits in its original location. If a table has lots of forwarded records, scanning the table can be

very inefficient.

- **Ghost_Record_Count and version_ghost_record_count** Ghost records are rows that physically still exist on a page but logically have been removed, as discussed in the section entitled "Data Modification Internals." Background processes in SQL Server clean up ghost records, but until that happens, no new records can be inserted in their place. So if there are lots of ghost records, your table has the drawback of lots of internal fragmentation (that is, the table is spread out over more pages and takes longer to scan) with none of the advantages (there is no room on the pages to insert new rows to avoid external fragmentation). A subset of ghost records is measured by *version_ghost_record_count*. This value reports the number of rows that have been retained by an outstanding Snapshot isolation transaction. These are not cleaned up until all relevant transactions have been committed or rolled back. Snapshot isolation is discussed in Chapter 10.

Removing Fragmentation

If fragmentation becomes too severe and is affecting query performance, you have several options for removing it. You might also wonder how severe is too severe. First of all, fragmentation is not always a bad thing. The biggest performance penalty from having fragmented data arises when your application needs to perform an ordered scan on the data. The more the logical order differs from the physical order, the greater the cost of scanning the data. If, on the other hand, your application needs only one or a few rows of data, it doesn't matter whether the table or index data is in logical order or is physically contiguous, or whether it is spread all over the disk in totally random locations. If you have a good index to find the rows you are interested in, SQL Server can find one or a few rows very efficiently, wherever they happen to be physically located.

If you are doing ordered scans of an index (such as table scans on a table with a clustered index, or a leaf-level scan of a nonclustered index), it is frequently recommended that if your *avg_fragmentation_in_percent* value is between 5 and 20, you should reorganize your index to remove the fragmentation. As we see shortly, reorganizing an index (also called *defragging*) compacts the leaf-level pages back to their originally specified fillfactor and then rearranges the pages in the leaf level to correct the logical fragmentation, using the same pages that the index originally occupied. No new pages are allocated, so this is a much more space-efficient operation than rebuilding the index.

If the *avg_fragmentation_in_percent* value is greater than 30, you should consider completely rebuilding your index. Rebuilding an index means that a whole new set of pages is allocated for the index. This removes almost all fragmentation, but it is not guaranteed to eliminate it completely. If the free space in the database is itself fragmented, you might not be able to allocate enough contiguous space to remove all gaps between extents. In addition, if other work is going on that needs to allocate new extents while your index is being rebuilt, the extents allocated to the two processes can end up being interleaved.

Defragmentation is designed to remove logical fragmentation from the leaf level of an index while keeping the index online and as available as possible. When defragmenting an index, SQL Server acquires an Intent-Exclusive lock on the index B-tree. Exclusive page locks are taken on individual pages only while those pages are being manipulated, as we see later in this chapter when we describe the defragmentation algorithm. Defragmentation in SQL Server 2008 is initiated using the *ALTER INDEX* command. The general form of the command to remove fragmentation is as follows:

```
ALTER INDEX { index_name | ALL }
    ON <object>
      REORGANIZE
            [ PARTITION = partition_number ]
            [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
```

ALTER INDEX with the REORGANIZE option offers enhanced functionality compared to *DBCC INDEXDEFRAG* in SQL Server 2000. It supports partitioned indexes, so you can choose to defragment just one particular partition (the default is to defragment all the partitions), and it allows you to control whether the LOB data is affected by the defragmenting.

As mentioned earlier, every index is created with a specific fillfactor. The initial fillfactor value is stored with the index metadata, so when defragmenting is requested, SQL Server can inspect this value. During defragmentation, SQL Server attempts to reestablish the initial fillfactor if it is greater than the current fillfactor on a leaf-level page. Defragmentation is designed to compact data, and this can be done by putting more rows per page and increasing the fullness percentage of each page. SQL Server might end up then removing pages from the index after the defragmentation. If the current fillfactor is greater than the initial fillfactor, SQL Server cannot *reduce* the fullness level of a page by moving rows out of it. The compaction algorithm inspects adjacent pages (in logical order) to see if there is room to move rows from the second page to the first. From SQL Server 2005 onwards, the process is even more efficient by looking at a sliding window of eight logically consecutive pages. It determines whether enough rows can be moved around within the eight pages to allow a

single page to be emptied and removed, and moves rows only if this is the case.

As mentioned earlier, SQL Server 2005 also introduced the option to compact your LOB pages. The default is ON. Reorganizing a specified clustered index compacts all LOB columns that are contained in the clustered index before it compacts the leaf pages. Reorganizing a nonclustered index compacts all LOB columns that are non-key (*INCLUDEd*) columns in the index.

In SQL Server 2000, the only way a user can compact LOBs in a table is to unload and reload the LOB data. LOB compaction in SQL Server 2005 onwards finds low-density extents—those that are used at less than 75 percent. It moves pages out of these low-density uniform extents and places the data from them in available space in other uniform extents already allocated to the LOB allocation unit. This functionality allows much better use of disk space, which can be wasted with low-density LOB extents. No new extents are allocated, either during this compaction phase or during the next phase.

The second phase of the reorganization operation actually moves data to new pages in the in-row allocation unit with the goal of having the logical order of data match the physical order. The index is kept online because only two pages at a time are processed in an operation similar to a heapsort or smoothsort (the details of which are beyond the scope of this book). The following example is a simplification of the actual process of reorganization. Consider an index on a column of *datetime* data. Monday's data logically precedes Tuesday's data, which precedes Wednesday's data, which precedes Thursday's data, and so on. If, however, Monday's data is on page 88, Tuesday's is on page 50, Wednesday's is on page 100, and Thursday's is on page 77, the physical and logical ordering doesn't match in the slightest, and we have logical fragmentation. When defragmenting an index, SQL Server determines the first physical page belonging to the leaf level (page 50, in our case) and the first logical page in the leaf level (page 88, which holds Monday's data) and swaps the data on those two pages using one additional new page as a temporary storage area. After this swap, the first logical page with Monday's data is on page 50, the lowest numbered physical page. After each page swap, all locks and latches are released and the key of the last page moved is saved. The next iteration of the algorithm uses the saved key to find the next logical page—Tuesday's data, which is now on page 88. The next physical page is 77, which holds Thursday's data. So another swap is made to place Tuesday's data on page 77 and Thursday's on page 88. This process continues until no more swaps need to be made. Note that no defragmenting is done for pages on mixed extents.

You need to be aware of some restrictions on using the REORGANIZE option. Certainly, if the index is disabled it cannot be defragmented. Also, because the process of removing fragmentation needs to work on individual pages, you get an error if you try to reorganize an index that has the option ALLOW_PAGE_LOCKS set to OFF. Reorganization cannot happen if a concurrent online index is built on the same index or if another process is concurrently reorganizing the same index.

You can observe the progress of each index's reorganization in the *sys.dm_exec_requests* DMV in the *percent_complete* column. The value in this column reports the percentage completed in one index's reorganization. If you are reorganizing multiple indexes in the same command, you might see the value go up and down as each index is defragmented in turn.

Rebuilding an Index

You can completely rebuild an index in several ways. You can use a simple combination of *DROP INDEX* followed by *CREATE INDEX*, but this method is probably the least preferable. In particular, if you are rebuilding a clustered index in this way, all the nonclustered indexes must be rebuilt when you drop the clustered index. This nonclustered index rebuilding is necessary to change the row locators in the leaf level from the clustered key values to row IDs. Then, when you rebuild the clustered index, all the nonclustered indexes must be rebuilt again. In addition, if the index supports a PRIMARY KEY or UNIQUE constraint, you can't use the *DROP INDEX* command at all—unless you first drop all the FOREIGN KEYs. Although this is possible, it is not preferable.

Better solutions are to use the *ALTER INDEX* command or to use the DROP_EXISTING clause along with *CREATE INDEX*. As an example, here are both methods for rebuilding the *PK_TransactionHistory_TransactionID* index on the *Production.TransactionHistory* table:

```
ALTER INDEX PK_TransactionHistory_TransactionID
          ON Production.TransactionHistory REBUILD;

CREATE UNIQUE CLUSTERED INDEX PK_TransactionHistory_TransactionID
              ON Production.TransactionHistory
                   (TransactionDate, TransactionID)
              WITH DROP_EXISTING;
```

Although the *CREATE* method requires knowing the index schema, it is actually more powerful and offers more options that

you can specify. You can change the columns that make up the index, change the uniqueness property, or change a nonclustered index to clustered, as long as there isn't already a clustered index on the table. You can also specify a new filegroup or a partition scheme to use when rebuilding. Note that if you do change the clustered index key properties, all nonclustered indexes must be rebuilt, but only once (not twice, as would happen if we were to execute *DROP INDEX* followed by *CREATE INDEX*).

When using the *ALTER INDEX* command to rebuild a clustered index, the nonclustered indexes never need to be rebuilt just as a side effect because you can't change the index definition at all. However, you can specify ALL instead of an index name and request that all indexes be rebuilt. Another advantage of the *ALTER INDEX* method is that you can specify just a single partition to be rebuilt—if, for example, the fragmentation report from *sys.dm_db_index_physical_stats* shows fragmentation in just one partition or a subset of the partitions.

Online Index Building

The default behavior of either method of rebuilding an index is that SQL Server takes an exclusive lock on the index, so it is completely unavailable while the index is being rebuilt. If the index is clustered, the entire table is unavailable; if the index is nonclustered, there is a shared lock on the table, which means no modifications can be made but other processes can *SELECT* from the table. Of course, they cannot take advantage of the index you're rebuilding, so the query might not perform as well as it should.

SQL Server 2005 introduced the option to rebuild one or all indexes online. The ONLINE option is available with both *ALTER INDEX* and *CREATE INDEX*, with or without the DROP_EXISTING option. Here's the syntax for building the preceding index, but doing it online:

```
ALTER INDEX PK_TransactionHistory_TransactionID
          ON Production.TransactionHistory REBUILD WITH (ONLINE = ON);
```

The online build works by maintaining two copies of the index simultaneously: the original (the source) and the new one (the target). The target is used only for writing any changes made while the rebuild is going on. All reading is done from the source, and modifications are applied to the source as well. SQL Server row-level versioning is used, so anyone retrieving information from the index can read consistent data. Figure 6-3 (taken from *SQL Server Books Online*) illustrates the source and target, and it shows three phases that the build process goes through. For each phase, the illustration describes what kind of access is allowed, what is happening in the source and target tables, and what locks are applied.

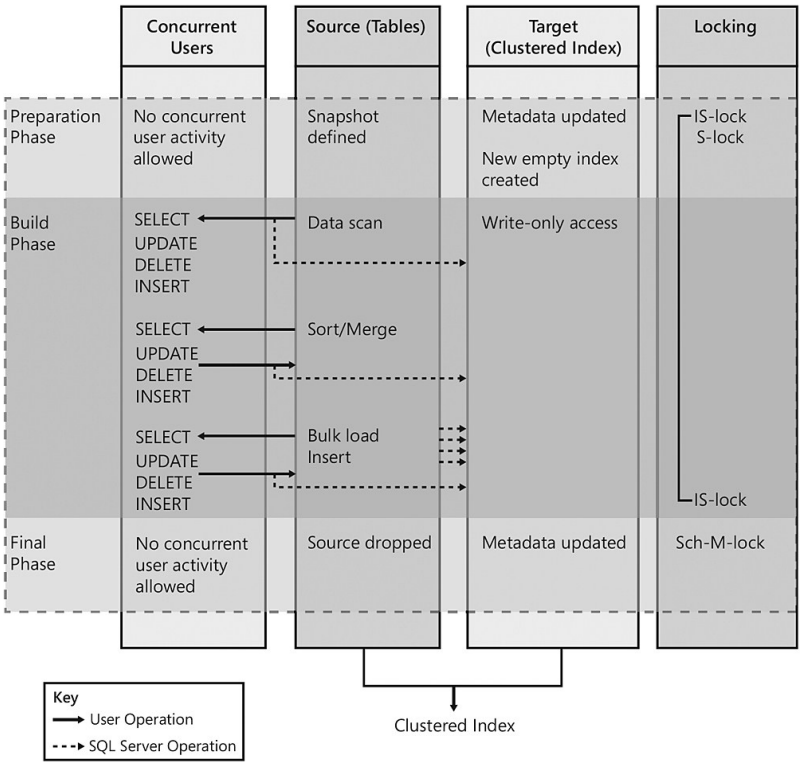

Figure 6-3: The structures and phases of online index building

The actual processes might differ slightly depending on whether the index is being built initially or being rebuilt and

whether the index is clustered or nonclustered.

Here are the steps involved in rebuilding a nonclustered index:

1. A Shared lock (S-lock) is taken on the index, which prevents any data modification queries, and an Intent-Shared lock (IS-lock) is taken on the table.
2. The index is created with the same structures as the original and marked as write-only.
3. The Shared lock is released on the index, leaving only the Intent-Shared lock on the table.
4. A versioned scan (discussed in detail in Chapter 10) is started on the original index, which means modifications made during the scan are ignored. The scanned data is copied to the target.
5. All subsequent modifications write to both the source and the target. Reads use only the source.
6. The scan of the source and copy to the target continues while normal operations are performed. SQL Server uses a proprietary method for reconciling obvious problems such as a record being deleted before the scan has inserted it into the new index.
7. The scan completes.
8. A Schema-Modification lock (Sch-M-lock)—the strictest of all types of locks—is taken to make the table completely unavailable.
9. The source index is dropped, metadata is updated, and the target index is made to be read-write.
10. The Schema-Modification lock is released.

Building a new nonclustered index involves exactly the same steps except there is no target index so the versioned scan is done on the base table, and write operations need to maintain only the target index rather than both indexes. A clustered index rebuild works exactly like a nonclustered rebuild, so long as there is no schema change (a change of index keys or uniqueness property).

For a build of new clustered index, or a rebuild of a clustered index with a schema change, there are a few more differences. First, an intermediate mapping index is used to translate between the source and target physical structures. In addition, all existing nonclustered indexes are rebuilt one at a time after a new base table has been built. For example, creating a clustered index on a heap with two nonclustered indexes involves the following steps:

1. Create a new write-only clustered index.
2. Create a new nonclustered index based on the new clustered index.
3. Create another new nonclustered index based on the new clustered index.
4. Drop the heap and the two original nonclustered indexes.

Before the operation is completed, SQL Server will be maintaining six structures at once. Online index building is not really considered a performance enhancement because an index can actually be built faster offline, and all these structures do not need to be maintained simultaneously. Online index building is an availability feature—you can rebuild indexes to remove all fragmentation or reestablish a fillfactor even if your data must be fully available at all times.

Note There are two exceptions to being able to perform online index operations:

- If the index contains a LOB column, online index operations are not available. This means that if the table contains a LOB column, the clustered index cannot be rebuilt online. Online operations are prevented only if a nonclustered index specifically includes a LOB column.
- A single partition of a clustered or nonclustered index cannot be rebuilt online.

Summary

In this chapter, we have discussed index concepts, index internals, special index structures, data modifications, and index management. We covered many best practices along the way and, although performance tuning wasn't our primary goal, the more you know about how indexes work internally, the more optimal structures you can create. In addition, by understanding how SQL Server organizes indexes on disk, you can be more adept at troubleshooting problems and

managing changes within your database.