CHAPTER 15

Implementing Indexes and Statistics

Exam objectives in this chapter:

- Create Database Objects
  - Create and alter views (simple statements).
- Troubleshoot & Optimize
  - Optimize queries.

In Chapter 14, "Using Tools to Analyze Query Performance", you learned about the tools that help you find performance problems. Indexes are mentioned in that chapter many times. This is not a coincidence. Proper indexing is necessary for good performance of your databases. In order to create appropriate indexes, you need to understand how Microsoft SQL Server stores data in tables and indexes, and how it then accesses this data. You learn about this in the longest lesson in this chapter, Lesson 1, "Implementing Indexes."

No indexes can help you if you write inefficient queries. In Lesson 2, "Using Search Arguments," you learn how to write arguments that SQL Server can use for seeks over indexes. However, even if you have indexes and proper searchable arguments, SQL Server might still decide not to use an index. This might happen because statistical information about the index is not present or is outdated. In Lesson 3, "Understanding Statistics," you learn how to get information about statistics and maintain it.

Lessons in this chapter:

- Lesson 1: Implementing Indexes
- Lesson 2: Using Search Arguments
- Lesson 3: Understanding Statistics

Before You Begin

To complete the lessons in this chapter, you must have:

- An understanding of relational database concepts.
- Experience working with SQL Server Management Studio (SSMS).
- Some experience writing T-SQL code.
- Access to a SQL Server 2012 instance with the sample database TSQL2012 installed.

Lesson 1: Implementing Indexes

SQL Server internally organizes data in a data file in pages. A page is an 8 KB unit and belongs to a single object; for example, to a table or an index. A page is the smallest unit of reading and writing. Pages are further organized into extents. An *extent* consists of eight consecutive pages. Pages from an extent can belong to a single object or to multiple objects. If the pages belong to multiple objects, then the extent is called a *mixed* extent; if the pages belong to a single object, then the extent is called a *uniform* extent. SQL Server stores the first eight pages of an object in mixed extents. When an object exceeds eight pages, SQL Server allocates additional uniform extents for this object. With this organization, small objects waste less space and big objects are less fragmented.

Although the previous information provides a brief introduction to the *physical* structure of SQL Server, from a database developer perspective, logical structures are much more important. This lesson focuses on logical structures.

After this lesson, you will be able to:

- Understand how SQL Server uses pages and extents.
- Describe heaps and balanced trees.
- Create clustered and nonclustered indexes.
- Create indexed views.

Estimated lesson time: 60 minutes

Heaps and Balanced Trees

Pages are physical structures. SQL Server organizes data in pages in logical structures.

SQL Server organizes tables as heaps or as *balanced trees*. A table organized as a balanced tree is also known as a clustered table or a clustered index. (You can use these two terms interchangeably.)

Indexes are always organized as balanced trees. Other indexes, such as indexes that do not contain all of the data and serve as pointers to table rows for quick seeks, are called *non-clustered indexes*.

Heaps

A *heap* is a quite simple structure. Data in a heap is not organized in any logical order. A heap is just a bunch of pages and extents.

SQL Server traces which pages and extents belong to an object through special system pages called Index Allocation Map (IAM) pages. Every table or index has at least one IAM page, called *first IAM*. A single IAM page can point to approximately 4 GB of space. Large objects can have more than one IAM page. IAM pages for an object are organized as a *doubly linked list*; each page has a pointer to its descendant and antecedent. SQL Server stores pointers to first IAM pages in its own internal system tables.

Figure 15-1 shows what an exemplary table for storing customers' orders looks like when it is organized as a heap.

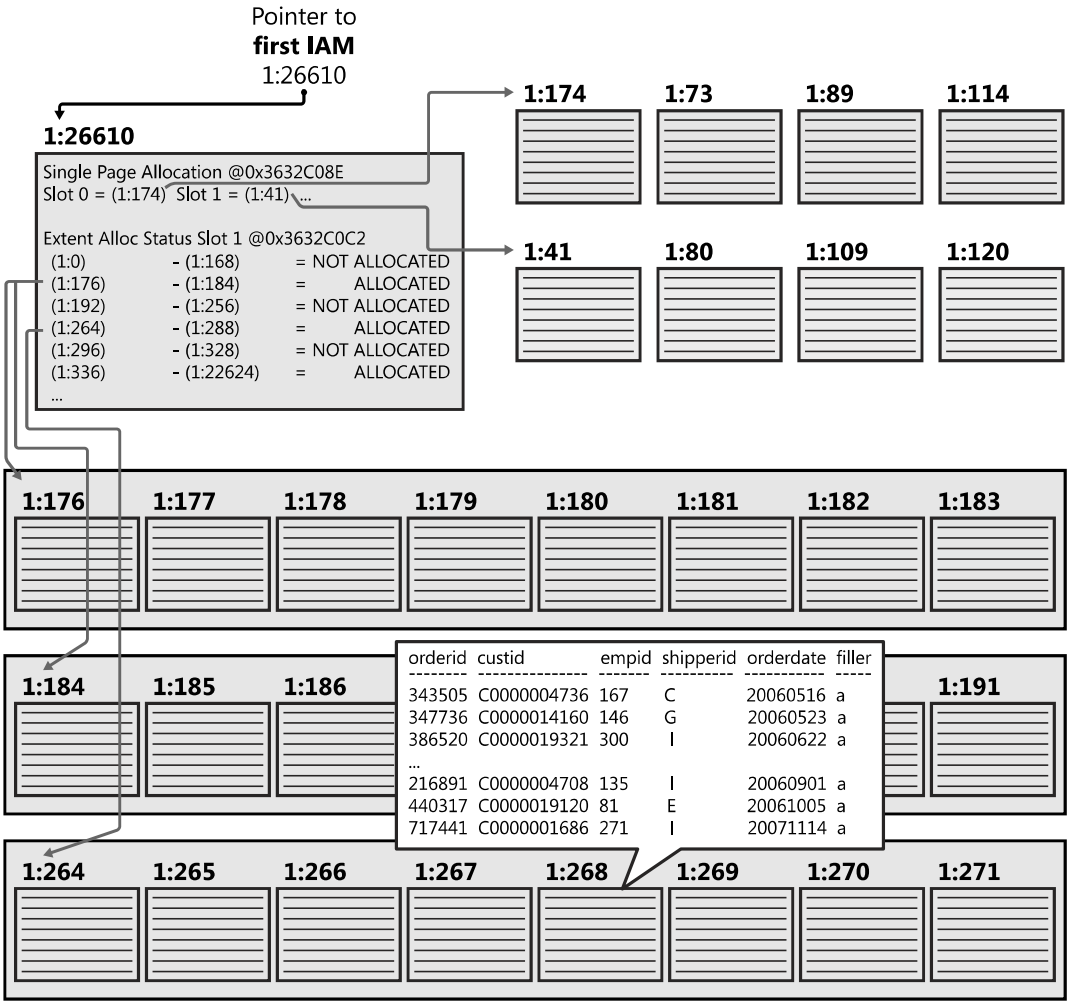

FIGURE 15-1 A table organized as a heap.

SQL Server can find data in a heap only by scanning the whole heap. SQL Server uses IAM pages to scan heaps in physical order, or *allocation* order. Even if your query wants to retrieve only a single row, SQL Server has to scan the entire heap. SQL Server stores new rows anywhere in a heap. It can store a new row in an existing page if the page is not full, or allocate a new page or extent for the object where you are inserting the new row. Of course, this means that heaps can become very fragmented over time.

You can better understand SQL Server structures through examples. The following code creates a table organized as a heap.

```
CREATE TABLE dbo.TestStructure
(
id      INT       NOT NULL,
filler1 CHAR(36)  NOT NULL,
filler2 CHAR(216) NOT NULL
);
```

If you do not create a clustered index explicitly or implicitly through primary key or unique constraints, then a table is organized as a heap. SQL Server does not allocate any pages for a table when you create it. It allocates the first page, and also the first IAM page, when you insert the first row in the table. You can find general information about tables and indexes in the sys.indexes catalog view.

The following query retrieves basic information about the dbo.TestStructure table that was created from the previous code.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS index_name, type, type_desc
FROM sys.indexes
WHERE object_id = OBJECT_ID(N'dbo.TestStructure', N'U');
```

The results of the query are as follows.

```
table_name     index_name  type  type_desc
-------------  ----------  ----  ---------
TestStructure  NULL        0     HEAP
```

The type column stores a value of 0 for heaps, 1 for clustered tables (indexes), and 2 for nonclustered indexes. You can find out how many pages are allocated for an object from the sys.dm_db_index_physical_stats dynamic management view or with the help of the dbo.sp_spaceused system procedure, as shown in the following code. Because this code is reused many times in this lesson, this lesson refers to it as the "heap allocation check" for easy identification.

```
SELECT index_type_desc, page_count,
 record_count, avg_page_space_used_in_percent
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'), NULL, NULL , 'DETAILED');
EXEC dbo.sp_spaceused @objname = N'dbo.TestStructure', @updateusage = true;
```

The output of these two commands is as follows.

```
index_type_desc  page_count  record_count  avg_page_space_used_in_percent
---------------  ----------  ------------  ------------------------------
HEAP             0           0             0

name            rows  reserved  data  index_size  unused
--------------  ----  --------  ----  ----------  ------
TestStructure   0     0 KB      0 KB  0 KB        0 KB
```

You can see that the table is empty, and an empty table does not occupy any space. Note the last column in the output of the first query, the avg_space_used_in_percent column. This column shows internal fragmentation. *Internal fragmentation* means that pages are not full. The more rows you have stored on a single page, the fewer pages SQL Server must read to retrieve these rows, and the less memory it uses for cached pages for the same number of rows. In heaps, you do not get much internal fragmentation, because SQL Server stores new rows in existing pages, as you already know, if there is enough space there. Now insert the first row.

```
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(1, 'a', 'b');
```

If you run the heap allocation check code again, you get the following results.

```
index_type_desc  page_count  record_count  avg_page_space_used_in_percent
---------------  ----------  ------------  ------------------------------
HEAP             1           1             3.24932048430937

name            rows  reserved  data  index_size  unused
--------------  ----  --------  ----  ----------  ------
TestStructure   1     16 KB     8 KB  8 KB        0 KB
```

The table occupies one page with one row. Average page space used is low because there is only a single row in the page. The results of the dbo.sp_spaceused procedure show that the table has two pages reserved, one page for the data and one for the first IAM page. You can see that SQL Server allocates only a page and not an extent for the table. Now fill the page by using the following code.

```
DECLARE @i AS int = 1;
WHILE @i < 30
BEGIN
SET @i = @i + 1;
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(@i, 'a', 'b');
END;
```

After you run the heap allocation, check the code again. You get the following results.

```
index_type_desc  page_count  record_count  avg_page_space_used_in_percent
---------------  ----------  ------------  ------------------------------
HEAP             1           30            98.1961947121324

name           rows  reserved  data  index_size  unused
-------------  ----  --------  ----  ----------  ------
TestStructure  30    16 KB     8 KB  8 KB        0 KB
```

There is still only one page allocated; however, this page has no internal fragmentation because the page cannot accommodate any additional rows. Try to insert an additional row.

```
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(31, 'a', 'b');
```

The heap allocation check code returns the following output.

```
index_type_desc  page_count  record_count  avg_page_space_used_in_percent
---------------  ----------  ------------  ------------------------------
HEAP             2           31            50.7227575982209

name           rows  reserved  data   index_size  unused
-------------  ----  --------  -----  ----------  ------
TestStructure  31    24 KB     16 KB  8 KB        0 KB
```

Now you can see that a single additional page from a mixed extent was allocated. Of course, internal fragmentation has risen, because the second page is nearly empty. Fill up eight pages by using the following code.

```
DECLARE @i AS int = 31;
WHILE @i < 240
BEGIN
SET @i = @i + 1;
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(@i, 'a', 'b');
END;
```

The results of the heap allocation check code are as follows.

```
index_type_desc  page_count  record_count  avg_page_space_used_in_percent
---------------  ----------  ------------  ------------------------------
HEAP             8           240           98.1961947121324

name           rows  reserved  data   index_size  unused
-------------  ----  --------  -----  ----------  ------
TestStructure  240   72 KB     64 KB  8 KB        0 KB
```

Eight pages are full. What happens if you insert a new row? Try it out with the following code.

```
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(241, 'a', 'b');
```

The results of the heap allocation check code are as follows.

```
index_type_desc  page_count  record_count  avg_page_space_used_in_percent
---------------  ----------  ------------  ------------------------------
HEAP             9           241           87.6465530022239

name           rows  reserved  data   index_size  unused
-------------  ----  --------  -----  ----------  ------
TestStructure  241   136 KB    72 KB  8 KB        56 KB
```

> ***NOTE*** **IDENTICAL RESULTS ARE NOT GUARANTEED**
>
> **With a different database configuration—for example, if you have two or more data files—your results might slightly differ.**

Now you can see that although the table occupies only 9 pages, 16 data pages plus the first IAM page are reserved for the table. As the results of the dbo.sp_spaceused procedure show, SQL Server reserved 136 KB for the table, which means 17 pages; 56 KB are still unused. The unused 56 KB of space means that 7 pages from a uniform extent are still empty. The first 8 pages stay on the mixed extents. Because the table is already bigger than 8 pages, SQL Server allocates uniform extents for additional space needed.

Clustered Indexes

You organize a table as a balanced tree when you create a clustered index. The structure is called a balanced tree because it resembles an inverse tree. Every balanced tree has a single root page and at least one or more leaf pages. In addition, it can have zero or more intermediate levels. All data in a clustered table is stored in leaf pages. Data is stored in logical order of the clustering key. A *clustering key* can consist of a single column, or of multiple columns. If the key consists of multiple columns, then this is a *composite key*. You can have up to 16 columns in a key; the size of all columns together in a composite key must not exceed 900 bytes. Note that data is stored logically and is not physically ordered. SQL Server still uses IAM pages to follow the physical allocation.

IMPORTANT **A CLUSTERED INDEX IS A TABLE**

When you create a clustered index, you are not making a copy of data; instead, you reorganize the table. In addition, do not suppose that the data is physically sorted; if you need an ordered output of a query, you have to include the ORDER BY clause.

Pages above leaf level point to leaf-level pages. A row in a page above leaf level contains a clustering key value and a pointer to a page where this value starts in logically ordered leaf level. If a single page can point to all leaf-level pages, then only a root page is allocated. If more than one page is needed to point to leaf-level pages, SQL Server creates the first intermediate-level pages, which point to leaf-level pages. The root page rows point to intermediate-level pages. If the root page cannot point to all first-level intermediate pages, SQL Server creates a new intermediate level. Pages on the same level are organized as a doubly linked list; therefore, SQL Server can find the previous and the next page in logical order for any specific page. In addition to balanced tree pages, SQL Server uses IAM pages to track physical allocation of the balanced tree pages.

You can use a column or columns with unique or nonunique values for a key of a clustered index. However, SQL Server internally always maintains uniqueness of the clustering key. It adds a *uniquifier* value, which is a sequential integer, to the repeating values. The first value is stored without a uniquifier; the first repeating value gets a uniquifier with a value of one, the second with two, and so on. You will understand why clustering key values must be unique internally when you learn about nonclustered indexes later in this lesson.

Figure 15-2 shows the clustered structure of the exemplary table for customers' orders. Note that the order data (the od column in the figure) is used for the clustering key. Because the column is not unique, the uniquifier is added to the repeating values (the unq column in the figure).

SQL Server can seek for a row in a clustered index. To find a specific row in the table represented in Figure 15-2, SQL Server has to read three pages only. If the table was organized as a heap, SQL Server would need to read the whole table, which would be comparable to reading all pages on the leaf level of the clustered index. Of course, if you request all rows, SQL Server scans the leaf level of the clustered index as well. A clustered index scan can be done in logical order or, when the logical order is not needed, in physical or allocation order. In addition, SQL Server can perform a partial scan if sequential rows in the order of the clustering key are requested by your query. These are some of the advantages of clustered indexes over heaps.

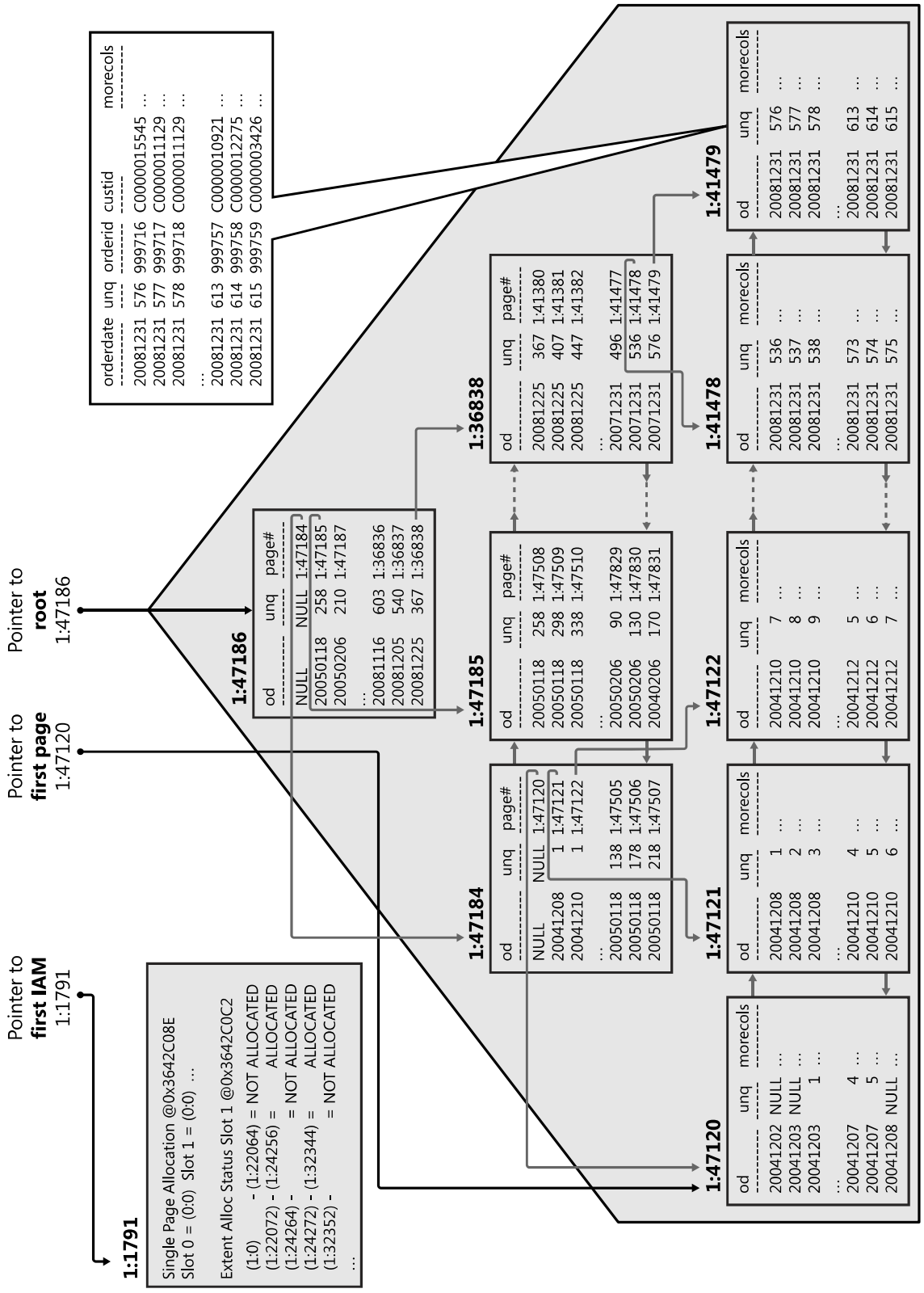

FIGURE 15-2 A table organized as a balanced tree.

Clustered indexes also have some disadvantages compared to heaps. When you insert a new row into a full page, SQL Server has to split the page into two pages and move half of the rows to the second page. This happens because SQL Server needs to maintain the logical order of the rows. This way, you get some internal fragmentation, which you cannot get in a heap. In addition, the new page (or new uniform extent for a large table) can be reserved anywhere in a data file. Physical order of pages and extents of a clustered table do not need to correspond to the logical order. If pages are physically out of order, then the clustered table is *logically* fragmented. This is also known as *external* fragmentation. External fragmentation can slow down full or partial scans in logical order.

In most cases, the advantages of clustered tables overwhelm the disadvantages. You can control the internal fragmentation with the FILLFACTOR option for the leaf-level pages and with the PAD_INDEX option for the higher-level pages of the CREATE INDEX statement. You can rebuild or reorganize an index to get rid of the external fragmentation by using the ALTER INDEX...REORGANIZE or ALTER INDEX...REBUILD statements.

A short clustering key means that more rows can fit on pages above the leaf level. Therefore, fewer levels are potentially needed. Fewer levels means a more efficient index because SQL Server needs to read fewer pages to find a row. A uniquifier extends the key; therefore, having a short and unique key is preferred for seeks. This is very typical for online transaction processing (OLTP) applications. For such applications, selecting a sequential integer as the clustering key is typically a very good choice. However, in data warehousing scenarios, many queries read huge amounts of data, typically ordered. For example, many data warehouse queries search for rows in order of a date or datetime column. If this is the case, then you might prefer to support such a partial scan, and create a clustered index on the date column.

EXAM TIP

Make sure you clearly understand how to select a clustering key in different environments.

You can learn more about clustered tables through examples. The following code truncates the table created and populated in the heap section of this lesson and reorganizes the table into a balanced tree by using the id column as the clustering key.

```
TRUNCATE TABLE dbo.TestStructure;
CREATE CLUSTERED INDEX idx_cl_id ON dbo.TestStructure(id);
```

You can check the sys.indexes catalog view for this table again.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS index_name, type, type_desc
FROM sys.indexes
WHERE object_id = OBJECT_ID(N'dbo.TestStructure', N'U');
```

This query returns the following output.

```
table_name     index_name  type  type_desc
-------------  ----------  ----  ---------
TestStructure  idx_cl_id   1     CLUSTERED
```

As you can see, the type has changed to 1 and the heap does not exist anymore. When you create a clustered index, you actually reorganize the table. Now fill 621 pages of this table by using unique values for the clustering key.

```
DECLARE @i AS int = 0;
WHILE @i < 18630
BEGIN
SET @i = @i + 1;
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(@i, 'a', 'b');
END;
```

Note that if you know that the values have to be unique, you should create a primary key or a unique constraint on the table. You could also create a unique index; however, because uniqueness is constraining the values, you should use constraints instead.

You can check some basic information about the index by querying the sys.dm_db_index_physical_stats dynamic management function. The following piece of code is reused multiple times in this part of the lesson, so further references to this code will be to the "clustered index allocation check" code.

```
SELECT index_type_desc, index_depth, index_level, page_count,
 record_count, avg_page_space_used_in_percent
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'), NULL, NULL , 'DETAILED');
```

The result of the clustered index allocation check code is as follows.

```
index_type_desc index_depth index_level page_count record_count avg_pg_spc_used_in_pct
--------------- ----------- ----------- ---------- ------------ ----------------------
CLUSTERED INDEX 2           0           621         18630       98.1961947121324
CLUSTERED INDEX 2           1           1           621         99.7158388930072
```

> ***NOTE*** **COLUMN NAMES IN THE OUTPUT**
>
> **In the results shown, some column names are slightly shortened from the actual output column names in order to fit the output on the book page.**

You can see that the index has two levels only, the leaf level and the root page. The root page has 621 rows that point to 621 leaf pages. There is no internal fragmentation in this case. Now insert one more row.

```
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(18631, 'a', 'b');
```

By running the clustered index allocation check code, you get the following output.

```
index_type_desc index_depth index_level page_count record_count avg_pg_spc_used_in_pct
--------------- ----------- ----------- ---------- ------------ ----------------------
CLUSTERED INDEX 3           0           622        18631        98.0435507783543
CLUSTERED INDEX 3           1           2          622          49.9258710155671
CLUSTERED INDEX 3           2           1          2            0.296515937731653
```

Now the index has three levels. Because a new page was allocated on the leaf level, the original root page could not reference all leaf pages anymore. SQL Server added an intermediate level with two pages pointing to 622 leaf pages, and a new root page pointing to the two intermediate-level pages.

In order to demonstrate the influence of the uniquifier, the following code truncates the table and fills 423 pages by using nonunique values for the clustering key.

```
TRUNCATE TABLE dbo.TestStructure;
DECLARE @i AS int = 0;
WHILE @i < 8908
BEGIN
SET @i = @i + 1;
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(@i % 100, 'a', 'b');
END;
```

If you run the clustered index allocation check code, you get the following output.

```
index_type_desc index_depth index_level page_count record_count avg_pg_spc_used_in_pc
--------------- ----------- ----------- ---------- ------------ ----------------------
CLUSTERED INDEX 2           0           423        8908         70.9815171732147
CLUSTERED INDEX 2           1           1          423          99.8393872003954
```

Note that the root page can refer to 423 leaf-level pages only. To fill two levels of the index, only 8,908 rows were needed, whereas with unique values for the clustering key in the previous case, SQL Server could accommodate 18,630 rows in two levels.

To prove that, add another row.

```
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(8909 % 100, 'a', 'b');
```

The clustered index allocation check code returns the following output.

```
index_type_desc index_depth index_level page_count record_count avg_pg_spc_used_in_pc
--------------- ----------- ----------- ---------- ------------ ----------------------
CLUSTERED INDEX 3           0           424        8909         70.8220039535458
CLUSTERED INDEX 3           1           2          424          50.0370644922165
CLUSTERED INDEX 3           2           1          2            0.395354583642204
```

You can see that SQL Server has to add an additional level to the index much earlier when the values of the key are not unique.

So far, the values of the clustering key were sequential. What happens if they are not? The following code truncates the dbo.TestStructure table, drops the existing clustered index, and creates a new one by using the filler1 column as the clustering key, and then inserts 9,000 rows in the table with unique sequential values in the clustering key.

```
TRUNCATE TABLE dbo.TestStructure;
DROP INDEX idx_cl_id ON dbo.TestStructure;
CREATE CLUSTERED INDEX idx_cl_filler1 ON dbo.TestStructure(filler1);
DECLARE @i AS int = 0;
WHILE @i < 9000
BEGIN
SET @i = @i + 1;
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(@i, FORMAT(@i,'0000'), 'b');
END;
```

Now check the fragmentation. The following code, referred to as the "fragmentation check" code later in this part, checks the internal fragmentation (the avg_page_space_used_in_percent column) and the external fragmentation (the avg_fragmentation_in_percent column).

```
SELECT index_level, page_count,
 avg_page_space_used_in_percent, avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'), NULL, NULL , 'DETAILED');
```

The output of the fragmentation check code in this case is as follows.

```
index_level page_count avg_page_space_used_in_percent avg_fragmentation_in_percent
----------- ---------- ------------------------------ ---------------------------
0           300        98.1961947121324               1.66666666666667
1           3          55.5720286632073               0
2           1          1.64319248826291               0
```

> ***NOTE*** **IDENTICAL RESULTS ARE NOT GUARANTEED**
>
> **The values for the avg_page_space_used_in_percent and the avg_fragmentation_in_percent columns might slightly differ in your results.**

You can see that the index has three levels. There is no internal fragmentation on the leaf level; in addition, there is nearly no external fragmentation. All pages on the leaf level are full and the physical order is nearly the same as the logical order. Now truncate the table and fill it with random values in the filler1 column. The following code uses the NEWID() T-SQL function that generates GUIDs and stores the GUIDs in the filler1 column.

```
TRUNCATE TABLE dbo.TestStructure;
DECLARE @i AS int = 0;
WHILE @i < 9000
BEGIN
SET @i = @i + 1;
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(@i, CAST(NEWID() AS CHAR(36)), 'b');
END;
```

GUIDs generated by the NEWID() function are nearly random. If you run the fragmentation check code again, you get the following output.

```
index_level page_count avg_page_space_used_in_percent avg_fragmentation_in_percent
----------- ---------- ------------------------------ ---------------------------
0           432        68.1842599456387               98.6111111111111
1           4          60.0197677291821               50
2           1          2.19915987150976               0
```

> ***NOTE*** **IDENTICAL RESULTS ARE NOT GUARANTEED**
>
> **The values for the avg_page_space_used_in_percent and the avg_fragmentation_in_percent columns might slightly differ in your results.**

You can see that the leaf-level pages have only 68 percent of the space filled with rows. This is because SQL Server performed multiple page splits. In addition, the external fragmentation is around 99 percent; almost no page is physically in correct logical order. You can see that using GUIDs for clustering keys can lead to quite inefficient indexes. External fragmentation mainly slows down scans, which should not be that frequent in OLTP environments; however, they are very important in the data warehousing area. Internal fragmentation is a problem in both scenarios because the table is much bigger than it would be with a sequential key.

You can get rid of the fragmentation if you rebuild or reorganize the index. Reorganizing an index is a slower but less intrusive process than rebuilding an index. As a general guideline, you should reorganize an index when the external fragmentation is less than 30 percent and rebuild it if it is greater than 30 percent. The following code rebuilds the index.

```
ALTER INDEX idx_cl_filler1 ON dbo.TestStructure REBUILD;
```

If you would prefer to reorganize the table, you should just replace the keyword REBUILD with the keyword REORGANIZE. If you run the fragmentation check code after the rebuild, you see from the output that there is almost no fragmentation anymore.

```
index_level page_count avg_page_space_used_in_percent avg_fragmentation_in_percent
----------- ---------- ------------------------------ ---------------------------
0           300        98.1961947121324               0.666666666666667
1           2          83.3703978255498               0
2           1          1.08722510501606               0
```

NOTE **IDENTICAL RESULTS ARE NOT GUARANTEED**

The values for the avg_page_space_used_in_percent and the avg_fragmentation_in_percent columns might slightly differ in your results.

Quick Check

- What kind of clustering key would you select for an OLTP environment?

Quick Check Answer

- For an OLTP environment, a short, unique, and sequential clustering key might be the best choice.

Implementing Nonclustered Indexes

Nonclustered indexes have a very similar structure to clustered ones. Actually, the root and the intermediate levels look the same as in a clustered index. The leaf level is different because it does not hold all of the data. What is stored on the leaf level of a nonclustered index depends on the underlying table organization, whether it is orgnized as a heap or as a balanced tree. You can have up to 999 nonclustered indexes on a single table.

The leaf level of a nonclustered index contains the index keys and *row locators*. Again, you can have up to 16 columns in a key, and the size of all columns together in a composite key must not exceed 900 bytes. A row locator points to a row in the underlying table. If the table is a heap, then the row locator is called row identifier (RID). This is an 8-byte pointer containing the database file ID and page ID of the target row, and the target row ID on that page.

Figure 15-3 shows a nonclustered index on a heap. It uses the same example of the customers' orders table as other figures in this chapter so far. The orderid column is used for the key of the index.

In order to seek for a row, SQL Server needs to traverse the index to the leaf level, and then read the appropriate page from the heap and retrieve the row from the page. The operation of retrieving the row from the heap is called *RID lookup*. If your query is very selective and searches for one row or a small amount of rows only, then index seek with RID lookup is very efficient. Because pages on the same level of an index are connected in a doubly linked list, SQL Server can also perform a partial or full ordered scan on a nonclustered index, and then perform RID lookups without starting the path from the root page for every row. However, as the number of rows the query retrieves increases, the RID lookup becomes much more expensive, because the cost of RID lookup is typically one page per row.

If a table is organized as a balanced tree, then the row locator is the clustering key. This means that when SQL Server seeks for a row, it has to traverse all levels on a nonclustered index and then also all levels of a clustered index. This operation is called a *key lookup*. At first glimpse, this sounds worse than retrieving a single page from a heap. However, because in this case the row locator is pointing to a logical structure and not to a physical structure, it does not matter where the row in the table is physically located. This means that you can freely reorganize or rebuild the clustered index; as long as you do not change the clustering key, SQL Server does not have to update the nonclustered indexes. If a row moves in a heap, SQL Server needs to update all nonclustered indexes to reflect the new position. SQL Server has an optimization for updates of a heap; if a row has to move to another page, SQL Server leaves a forwarding pointer to a new location in the original page, so SQL Server still does not have to update all nonclustered indexes. However, even with this optimization, it is still a good practice to organize tables as balanced trees. If the clustering key is narrow—for example a 4-byte integer—than SQL Server can also accommodate more rows on a leaf-level page than when RID is used as the row locator.

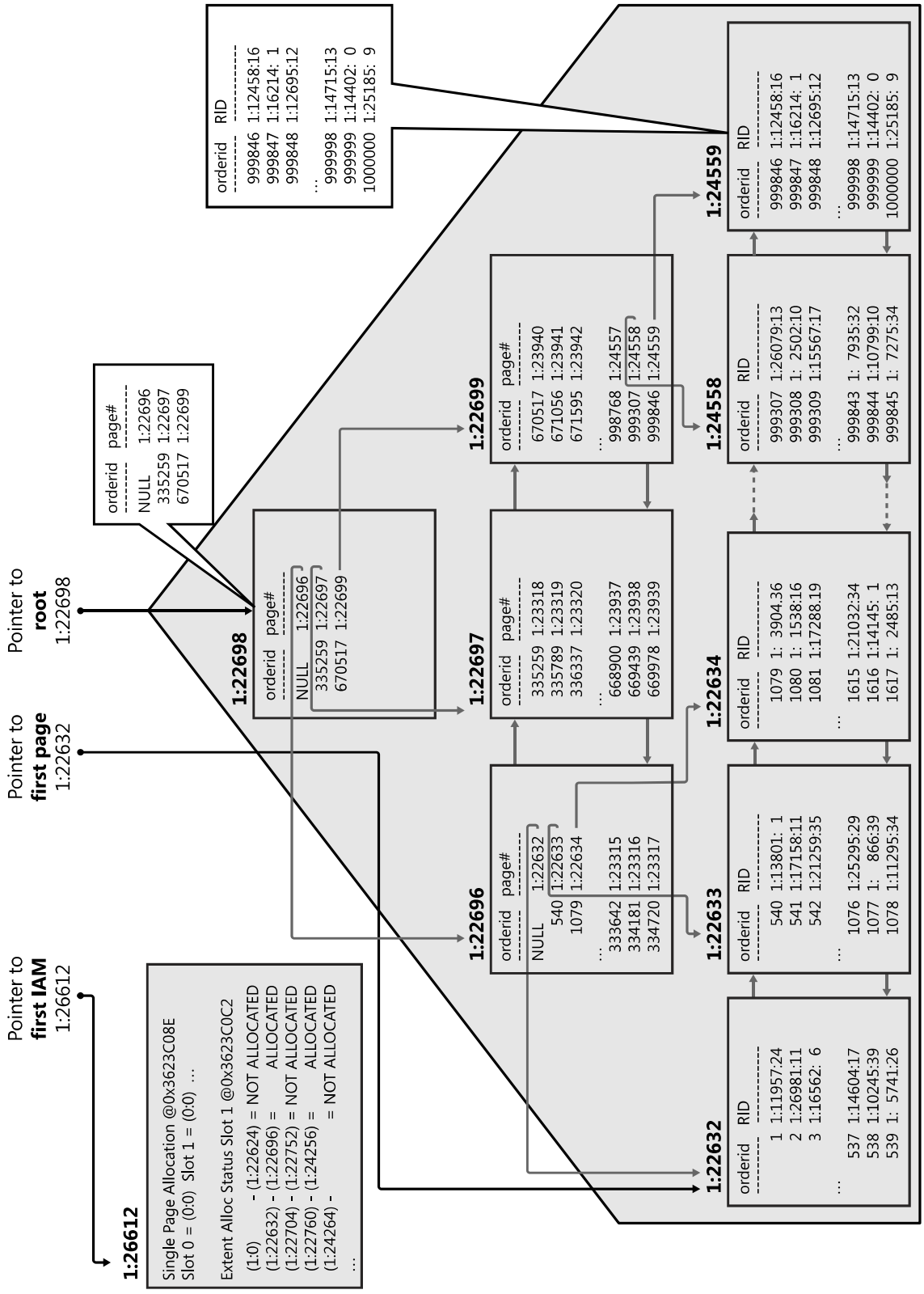

FIGURE 15-3 Nonclustered index on a heap.

Figure 15-4 shows a nonclustered index on a clustered table. It is the same example of the customers' orders table; order data is used for the clustering key, and order ID is used for the key of the nonclustered index.

Note that the clustering key is not unique, and therefore a uniquifier is added to the repeating values. The key of the nonclustered index is unique. If a query searches for a specific order ID (a single row) and gets a clustering key with an order date that is equal for more than one row, then SQL Server would return a wrong result set. SQL Server would return all the rows with the same order date. This is, of course, not acceptable. Therefore, SQL Server has to maintain uniqueness of clustering keys internally.

The clustering key should be short and unique because it appears in all nonclustered indexes. However, note again that this is not a general rule; in data warehousing scenarios, you might prefer to select a clustering key that supports frequent partial scans. In any case, the clustering should not change frequently, or preferably should not change at all. If you update a clustering key, SQL Server has to update all nonclustered indexes. You should also create a clustered index first, and then all nonclustered indexes. If you change the table structure from a heap to a balanced tree or vice-versa by creating or dropping a clustered index and the table has existing nonclustered indexes, SQL Server has to recreate all nonclustered indexes.

You can create a filtered nonclustered index. A filtered index spans a subset of column values only, and thus applies to a subset of table rows. Filtered nonclustered indexes are useful when some values in a column occur rarely, whereas other values occur frequently. In such cases, you would create a filtered index over the rare values only. SQL Server uses this index for seeks of rare values, but performs scans for frequent values. Filtered indexes are inexpensive to maintain, because SQL Server has to update them for changes in the rare values only. You create a filtered index by adding a WHERE clause to the CREATE INDEX statement. You could use a filtered index to enforce a filtered uniqueness. For example, imagine that a column has NULLs in multiple rows; however, known values must be unique. You cannot create a filtered primary key or unique constraint; however, you could create a filtered unique nonclustered index from known values only, which would allow multiple NULLs and reject duplicate known values.

SQL Server 2012 has a method for storing nonclustered indexes. In addition to regular row storage, SQL Server 2012 can store index data column by column in what's called a *columnstore index*. Columnstore indexes can speed up data warehousing queries by a large factor, from 10 to even 100 times.

A columnstore index is just another nonclustered index on a table. The SQL Server Query Optimizer considers using the columnstore index during the query optimization phase just as it does any other index. All you have to do to take advantage of this feature is create a columnstore index on a table.

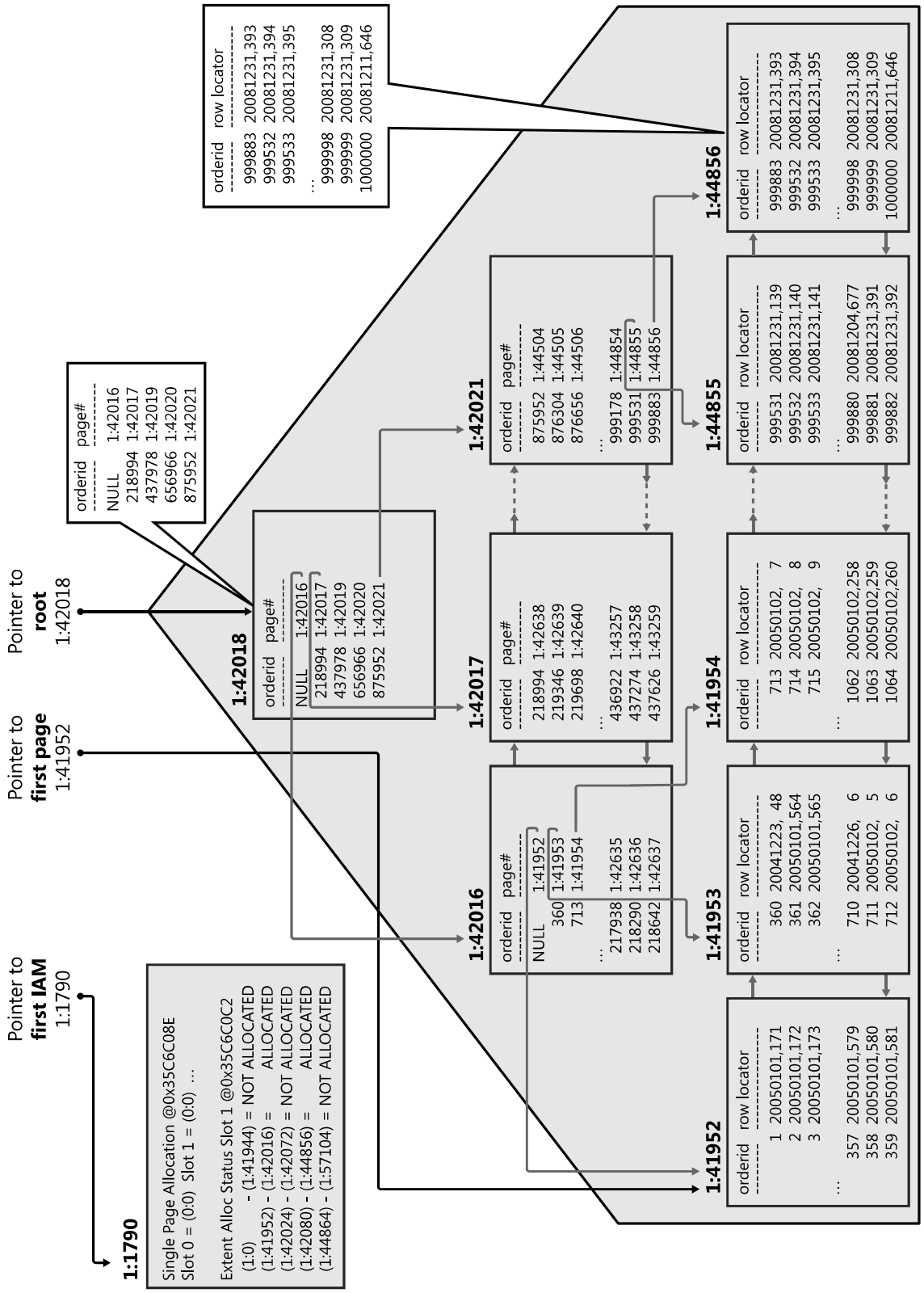

FIGURE 15-4 Nonclustered index on a clustered table.

A columnstore index is stored compressed. The compression factor can be up to 10 times the original size of the index. When a query references a single column that is a part of a columnstore index, then SQL Server fetches only that column from disk; it doesn't fetch entire rows as with row storage. This also reduces disk I/O and memory cache consumption. Columnstore indexes use their own compression algorithm; you cannot use Row or Page compression on a columnstore index.

On the other hand, SQL Server has to return rows. Therefore, rows must be reconstructed when you execute a query. This row reconstruction takes some time and uses some CPU and memory resources. Very selective queries that touch only a few rows might not benefit from columnstore indexes.

Columnstore indexes accelerate data warehouse queries, not OLTP workloads. Because of the row reconstruction issues and other overhead when you update compressed data, tables containing a columnstore index become read only. If you want to update a table by using a columnstore index, you must first drop the columnstore index. If you use table partitioning, you can switch a partition to a different table that does not use a columnstore index, update the data there, create a columnstore index on that table (which has a smaller subset of the data), and then switch the new table data back to a partition of the original table.

The columnstore index is divided into units called segments. Segments are stored as large objects and consist of multiple pages. *Segments* are the unit of transfer from disk to memory. Each segment has metadata that stores the minimum and maximum value of each column for that segment. This enables early segment elimination in the storage engine. SQL Server loads only those segments requested by a query into memory.

You learn more about columnstore indexes and their efficient usage in Chapter 17, "Understanding Further Optimization Aspects," You test regular nonclustered indexes in the practice for this lesson.

Implementing Indexed Views

You can optimize queries that aggregate data and perform multiple joins by permanently storing the aggregated and joined data. For example, you could create a new table by using joined and aggregated data and then maintain that table during your ETL process.

However, creating additional tables for joined and aggregated data is not a best practice, because using these tables means you have to change report queries. Fortunately, you can use another option for storing joined and aggregated tables. You can create a view with a query that joins and aggregates data. Then you can index the view to get an *indexed view*. With indexing, you are materializing a view. In the Enterprise edition of SQL Server 2012, the SQL Server Query Optimizer uses the indexed view automatically—without the need for you to change the query. SQL Server also maintains indexed views automatically. However, to speed up data loads, you can drop or disable the index before a load and then recreate or rebuild it after the load.

MORE INFO **FEATURES SUPPORTED BY SQL SERVER 2012 EDITIONS**

For more information on indexed view usage and other features supported by different editions of SQL Server 2012, see the Books Online for SQL Server article "Features Supported by the Editions of SQL Server 2012" at *http://msdn.microsoft.com/en-us/library/cc645993(SQL.110).aspx*.

Indexed views have many limitations, restrictions, and prerequisites, and you should refer to Books Online for SQL Server for details about them. However, you can create a simple test that shows how indexed views can be useful. The following query aggregates the qty column of the Sales.OrderDetails table over the shipcountry column of the Sales.Orders table in the TSQL2012 sample database. The code also sets STATISTICS IO to ON to measure the I/O.

```
USE TSQL2012;
SET STATISTICS IO ON;
-- Aggregate query with a join
SELECT O.shipcountry, SUM(OD.qty) AS totalordered
FROM Sales.OrderDetails AS OD
 INNER JOIN Sales.Orders AS O
  ON OD.orderid = O.orderid
GROUP BY O.shipcountry;
```

The query makes 11 logical reads in the Sales.OrderDetails table and 21 logical reads in the Sales.Orders table. You can create a view from this query and index it, as shown in the following code.

```
-- Create a view from the query
CREATE VIEW Sales.QuantityByCountry
WITH SCHEMABINDING
AS
SELECT O.shipcountry, SUM(OD.qty) AS total_ordered,
 COUNT_BIG(*) AS number_of_rows
FROM Sales.OrderDetails AS OD
 INNER JOIN Sales.Orders AS O
  ON OD.orderid = O.orderid
GROUP BY O.shipcountry;
GO
-- Index the view
CREATE UNIQUE CLUSTERED INDEX idx_cl_shipcountry
ON Sales.QuantityByCountry(shipcountry);
GO
```

Note that the view must be created with the SCHEMABINDING option if you want to index it. In addition, you must use the COUNT_BIG aggregate function. For details, see the prerequisites for indexed views article "Create Indexed Views" in Books Online for SQL Server 2012 at *http://msdn.microsoft.com/en-us/library/ms191432.aspx*. Nevertheless, after you create the view and the index, execute the aggregate query again and measure the I/O. This time, the query makes only two logical reads in the Sales.QuantityByCountry view.

After you analyze the indexed view, you should set the STATISTICS IO to OFF and clean up your TSQL2012 database by running the following code.

```
SET STATISTICS IO OFF;
DROP VIEW Sales.QuantityByCountry;
```

PRACTICE Analyzing Nonclustered Indexes

In this practice, you analyze nonclustered indexes.

If you encounter a problem completing an exercise, you can install the completed projects from the Solution folder that is provided with the companion content for this chapter and lesson.

EXERCISE 1 Implement a Nonclustered Index on a Heap

In this exercise, you create a nonclustered index on a heap.

1. Start SSMS and connect to your SQL Server instance.
2. Open a new query window by clicking the New Query button.
3. Change the context to the tempdb database. Set the NOCOUNT option to ON to stop the message that shows the count of the number of rows affected by a command. Inserts are much faster this way. Use the following code.

   ```
   USE tempdb;
   SET NOCOUNT ON;
   ```

4. Create a table named **dbo.TestStructure** that has the same structure as the table used for testing the heap and the clustered index structure. Use the following code.

   ```
   CREATE TABLE dbo.TestStructure
   (
   id      INT       NOT NULL,
   filler1 CHAR(36)  NOT NULL,
   filler2 CHAR(216) NOT NULL
   );
   GO
   ```

5. Note that the table is organized as a heap because you did not create a clustered index. Create a nonclustered index on the filler1 column by using the following code.

   ```
   CREATE NONCLUSTERED INDEX idx_nc_filler1 ON dbo.TestStructure(filler1);
   ```

6. Query the sys.indexes catalog view to confirm that the table is stored as a heap and that the nonclustered index exists.

   ```
   SELECT OBJECT_NAME(object_id) AS table_name,
    name AS index_name, type, type_desc
   FROM sys.indexes
   WHERE object_id = OBJECT_ID(N'dbo.TestStructure', N'U');
   ```

7. Insert 24,472 rows into the table. Create sequential values for the id and filler1 columns. Use the following code.

```
DECLARE @i AS int = 0;
WHILE @i < 24472
BEGIN
SET @i = @i + 1;
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(@i, FORMAT(@i,'0000'), 'b');
END;
```

8. Use the sys.dm_db_index_physical_stats dynamic management function to check how many levels the nonclustered index has and how many pages and rows are on each level. Also check the heap. You can use the following code.

```
SELECT index_type_desc, index_depth, index_level,
 page_count, record_count
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'), NULL, NULL , 'DETAILED');
```

9. You should have two levels of the nonclustered index. Now insert another row.

```
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(24473, '24473', 'b');
```

10. Check the number of levels of the nonclustered index and the heap again.

```
SELECT index_type_desc, index_depth, index_level,
 page_count, record_count
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'), NULL, NULL , 'DETAILED');
```

    Now you should have three levels in the nonclustered index.

EXERCISE 2 Implement a Nonclustered Index on a Clustered Table

In this exercise, you change the physical structure of the table in the previous exercise from a heap to a clustered index. You observe the difference between a nonclustered index on a heap and a nonclustered index on a clustered table.

1. Truncate the table you created in the previous exercise and create a clustered index on the id column.

```
TRUNCATE TABLE dbo.TestStructure;
CREATE CLUSTERED INDEX idx_cl_id ON dbo.TestStructure(id);
GO
```

2. Query the sys.indexes catalog view to confirm that the table is stored as a balanced tree and that the nonclustered index exists.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS index_name, type, type_desc
FROM sys.indexes
WHERE object_id = OBJECT_ID(N'dbo.TestStructure', N'U');
```

3. Insert 28,864 rows into the table. Create sequential values for the id and filler1 columns. Use the following code.

```
DECLARE @i AS int = 0;
WHILE @i < 28864
BEGIN
SET @i = @i + 1;
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(@i, FORMAT(@i,'0000'), 'b');
END;
```

4. Check the number of levels of the nonclustered index and the clustered index.

```
SELECT index_type_desc, index_depth, index_level,
 page_count, record_count
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'), NULL, NULL , 'DETAILED');
```

5. The clustered index should have three levels and the nonclustered two. You could accommodate more rows on each page of the nonclustered index on a clustered index than in the nonclustered index on a heap because the clustering key is shorter than the RID. Now add one more row.

```
INSERT INTO dbo.TestStructure
(id, filler1, filler2)
VALUES
(28865, '28865', 'b');
```

6. Check the number of levels of the nonclustered index and the clustered index again.

```
SELECT index_type_desc, index_depth, index_level,
 page_count, record_count
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'), NULL, NULL , 'DETAILED');
```

Now the nonclustered index should have three levels.

7. Clean up the tempdb database.

```
DROP TABLE dbo.TestStructure;
```

8. Close the query window.

Lesson Summary

- You can store a table as a heap or as a balanced tree. If the table is stored as a balanced tree, it is clustered; this is also known as a clustered index.
- You can create a nonclustered index on a heap or on a clustered table.
- You can also index a view.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. What levels can an index have? (Choose all that apply.)
   - **A.** Intermediate level
   - **B.** Heap level
   - **C.** Root level
   - **D.** Leaf level
2. How many clustered indexes can you create on a table?
   - **A.** 999
   - **B.** 16
   - **C.** 1
   - **D.** 900
3. What is the row locator when a table is stored as a balanced tree?
   - **A.** RID.
   - **B.** Columnstore index key.
   - **C.** Clustering key.
   - **D.** A table is never stored as a balanced tree.

Lesson 2: Using Search Arguments

Indexes are useful only if queries use them. You need to know which types of queries can benefit from indexes, and which types of queries does not use indexes, even if indexes exist. In addition, you need to write correct predicates when filtering rows in order to enable the SQL Server Query Optimizer to use indexes.

After this lesson, you will be able to:

- Support queries with indexes.
- Use appropriate search arguments in queries.

Estimated lesson time: 35 minutes

Supporting Queries with Indexes

Writing efficient queries starts by including the WHERE clause to filter rows. The WHERE clause is one of the most important parts of a query that can benefit from an index. You can check whether an index was used by displaying the estimated or actual execution plan. You can also track index usage by querying the sys.dm_db_index_usage_stats dynamic management view. Remember that information provided in dynamic management objects is cumulative from the last start of SQL Server.

The following query shows index usage in the TSQL2012 database. Note that the query was executed right after restarting SQL Server.

```
SELECT OBJECT_NAME(S.object_id) AS table_name,
 I.name AS index_name,
 S.user_seeks, S.user_scans, s.user_lookups
FROM sys.dm_db_index_usage_stats AS S
 INNER JOIN sys.indexes AS i
  ON S.object_id = I.object_id
   AND S.index_id = I.index_id
WHERE S.object_id = OBJECT_ID(N'Sales.Orders', N'U');
```

This query is going to be used in further examples in this lesson; therefore, for ease of reference, future references are to the "index usage" query. The index usage query does not return any rows because SQL Server does not have any information about index usage collected yet. The next query does not include a WHERE clause; it retrieves all rows from the Sales.Orders table.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders;
```

The execution plan for this query shows that SQL Server used a clustered index scan. The whole table was scanned, although there are many indexes on the Sales.Orders table. Note that the scan was unordered, or an allocation scan, as Figure 15-5 shows. The Ordered property of the operator is set to False. Remember that order is not guaranteed if you do not include the ORDER BY clause.

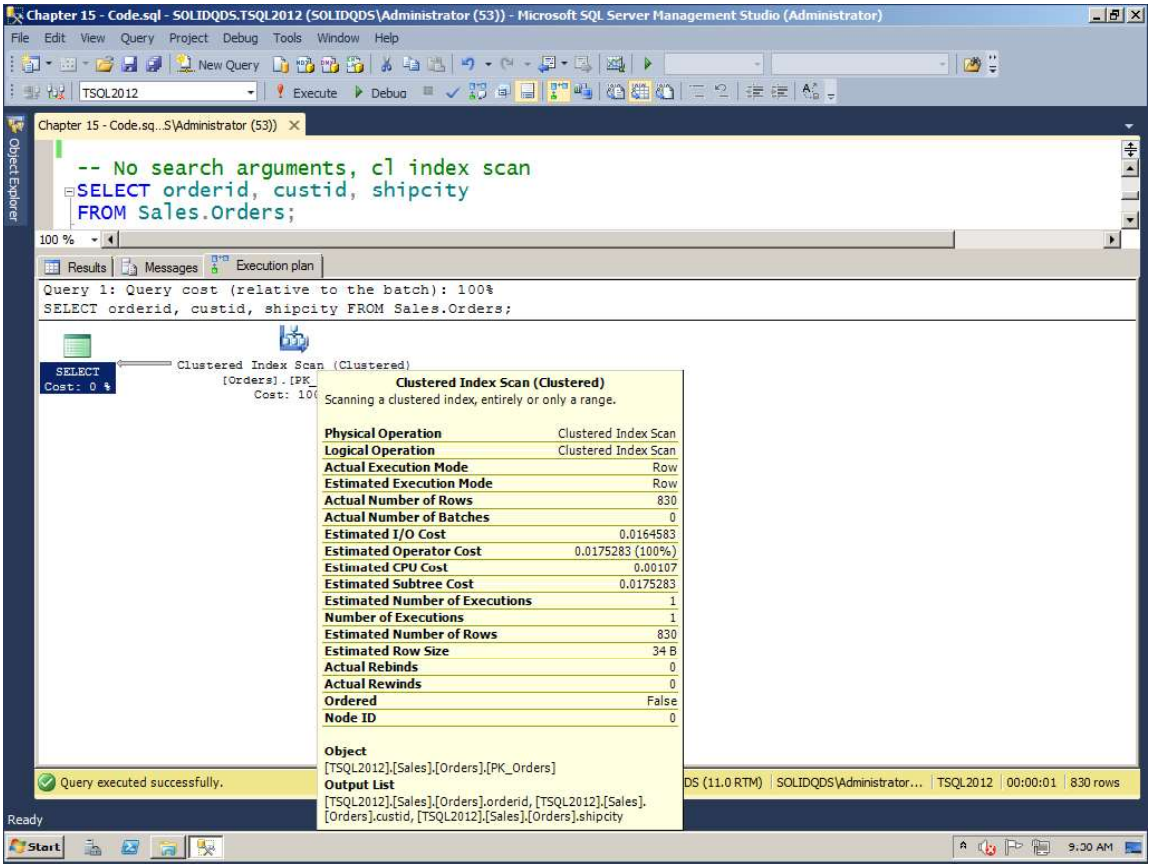

FIGURE 15-5 Unordered clustered index scan.

Adding a WHERE clause to the query does not guarantee that an index is going to be used. The clause has to be supported by an appropriate index, and it must be selective enough. If the query returns too many rows, it is less expensive for SQL Server to perform a table or clustered index scan than to do a nonclustered index seek and then RID or key lookups. For example, although the following query is quite selective, SQL Server still uses the clustered index scan because the WHERE predicate is not supported by an index. There is no index that would have the shipcity column for its key.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

The JOIN clause of a query can benefit from appropriate indexes as well. You will learn more about joins and supporting joins with indexes in Chapter 17.

If your query aggregates data and uses the GROUP BY clause, you should consider supporting this clause with an index. SQL Server can aggregate data by using a hash or a stream aggregate operator. The stream aggregate is faster; however, it needs sorted input. An aggregate query can benefit from an index even if it does not include the GROUP BY clause. For example, if you use the MIN() aggregation function and you have an appropriate index, then SQL Server can seek for the first value of an index only, and does not have to scan the entire

table. The following aggregate query is not supported by in index because there is no index on the Sales.Orders table that would use the shipregion column for the key.

```
SELECT shipregion, COUNT(*) AS num_regions
FROM Sales.Orders
GROUP BY shipregion;
```

Figure 15-6 shows the execution plan for this query.

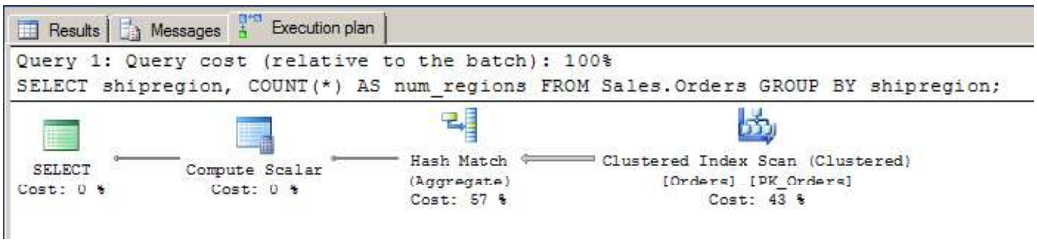

FIGURE 15-6 Hash Match (Aggregate) operator used when an aggregate query is not supported by an index.

Note that the results of the previous query are not ordered. Including the GROUP BY clause in a query does not guarantee a sorted result set. Again, if you need a sorted result, use the ORDER BY clause. However, if you include this clause, you should consider supporting it with an index as well. If there is no appropriate index for the ORDER BY clause, SQL Server must sort data before returning it. Sorting large datasets could be a big performance hit on SQL Server. The data needs to be sorted in memory or must be spilled to tempdb if it does not fit in memory. The following query uses an ORDER BY clause that is not supported by an index.

```
SELECT shipregion
FROM Sales.Orders
ORDER BY shipregion;
```

The execution plan for this query includes the Sort operator, as shown in Figure 15-7.

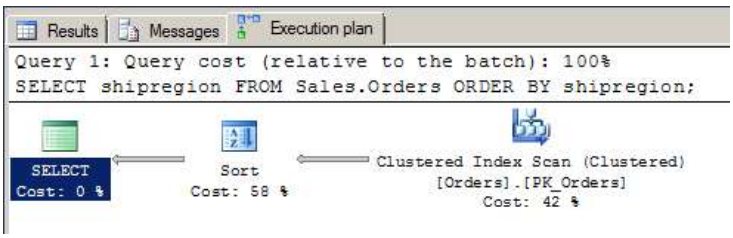

FIGURE 15-7 SQL Server using the Sort operator to sort the output.

Running the index usage query again returns the following output.

```
table_name  index_name  user_seeks  user_scans  user_lookups
----------  ----------  ----------  ----------  ------------
Orders      PK_Orders   0           4           0
```

All four queries executed in this lesson so far used a clustered index scan. It is time to start supporting the queries with appropriate indexes. The next piece of code creates a nonclustered index by using the shipregion column.

```
CREATE NONCLUSTERED INDEX idx_nc_shipregion ON Sales.Orders(shipregion);
```

If you execute the query that aggregates data and the query that requests sorted data again, like the following code shows, you get different execution plans.

```
-- Query that aggregates the data
SELECT shipregion, COUNT(*) AS num_regions
FROM Sales.Orders
GROUP BY shipregion;
-- Query that sorts the output
SELECT shipregion
FROM Sales.Orders
ORDER BY shipregion;
```

The execution plan for the first query uses the Stream Aggregate operator, and the execution plan for the second query does not include a Sort operator, as Figure 15-8 shows.

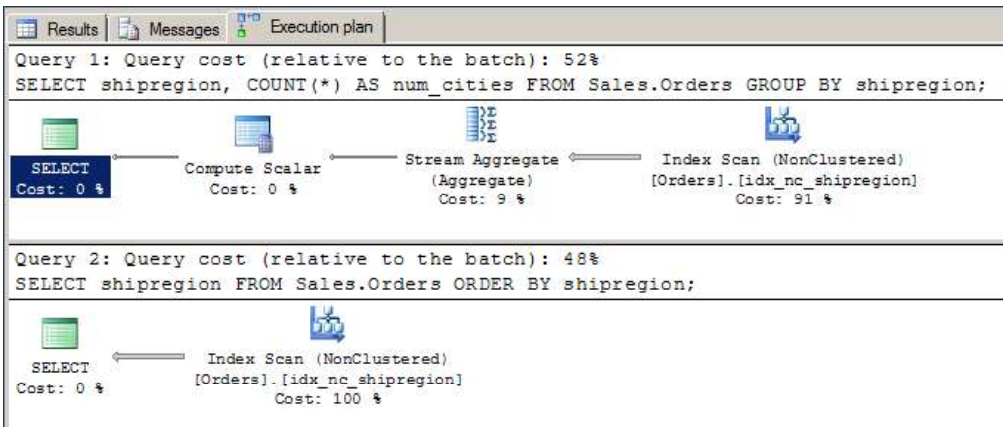

FIGURE 15-8 Execution plans for GROUP BY and ORDER BY queries supported by an index.

Running the index usage query returns the following output.

```
table_name  index_name         user_seeks  user_scans  user_lookups
----------  -----------------  ----------  ----------  ------------
Orders      idx_nc_shipregion  0           2           0
Orders      PK_Orders          0           4           0
```

Note that the idx_nc_shipregion nonclustered index was used for two scans, and there was no additional usage of the clustered index. You could also see this from the execution plans in Figure 15-8. SQL Server found all data for the query in the nonclustered index, and did not have to perform a RID or key lookup. If SQL Server finds all data in nonclustered indexes, then the query is *covered* by the nonclustered indexes, and the indexes are *covering* indexes. Covered queries are very efficient.

You could try to add more columns to a nonclustered index key to cover more queries. However, with a longer key, the index would become less efficient. There is another option in SQL Server 2012. You can also include a column in a nonclustered index on the leaf level only and not as a part of the key. You do this by using the INCLUDE clause of the CREATE INDEX statement. The included column is not part of the key, and SQL Server does not use it for seeks. Included columns help cover queries. However, you should be careful and not include too many columns. For example, if you included all columns of a table, you would actually copy the table.

The idx_nc_shipregion index is not useful for further examples in this lesson, so it's dropped in the following code.

```
DROP INDEX idx_nc_shipregion ON Sales.Orders;
```

Search Arguments

Including the WHERE clause in a query, even if the predicate is very selective and supported by an index, does not guarantee that SQL Server is going to use an index. You need to write an appropriate predicate to allow the Query Optimizer to take advantage of the indexes. The Query Optimizer is not omnipotent. It can decide to use an index only when the arguments in the predicate are searchable. You have to learn how to write appropriate search arguments (SARGs).

To write an appropriate SARG, you must ensure that a column that has an index on it appears in the predicate alone, not as a function parameter. SARGs must take the form of *column inclusive_operator <value>* or *<value> inclusive_operator column*. The column name is alone on one side of the expression, and the constant or calculated value appears on the other side. Inclusive operators include the operators =, >, <, =>, <=, BETWEEN, and LIKE. However, the LIKE operator is inclusive only if you do not use a wildcard % or _ at the beginning of the string you are comparing the column to. For example, the following query returns orders for the dates July 10, 2006, and July 11, 2006.

```
SELECT orderid, custid, orderdate, shipname
FROM Sales.Orders
WHERE DATEDIFF(day, '20060709', orderdate) <= 2
 AND DATEDIFF(day, '20060709', orderdate) > 0;
```

The query returns two rows only; therefore, the WHERE predicate is very selective. There is a nonclustered index on the orderdate column. However, SQL Server did not use the index, as the execution plan in Figure 15-9 shows.

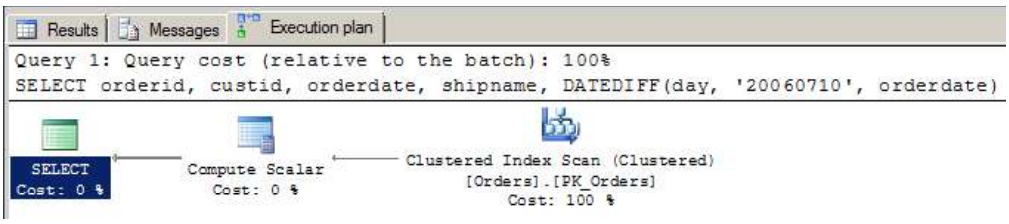

FIGURE 15-9 Query in which the predicate is not a SARG.

The orderdate in the predicate does not appear alone; it is instead an argument of a function. You can rewrite such a query many times. The following query produces the same result, but this time the predicate is a SARG.

```
SELECT orderid, custid, orderdate, shipname
FROM Sales.Orders
WHERE DATEADD(day, 2, '20060709') >= orderdate
 AND '20060709' < orderdate;
```

In Figure 15-10, you can see that SQL Server used the idx_nc_orderdate index for seeks and then performed key lookups

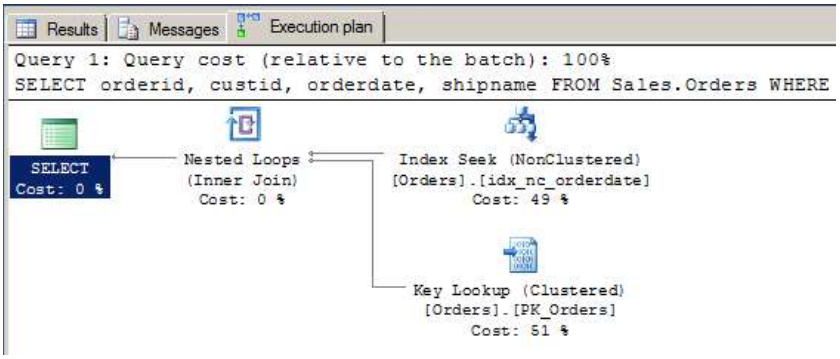

FIGURE 15-10 Query in which the predicate is a SARG.

You could rewrite the query in more different ways. You could use the IN operator to include the list of the dates for which you want the query to retrieve the orders. You could also use the equals operator for each date, and connect the two equals predicates with the logical OR operator. Actually, these two queries would be internally treated as the same; the Query Optimizer converts the IN operator to OR with a separate comparison to each element from the IN operator list. The following two queries return the same two rows and are internally treated as equal.

```
SELECT orderid, custid, orderdate, shipname
FROM Sales.Orders
WHERE orderdate IN ('20060710', '20060711');

SELECT orderid, custid, orderdate, shipname
FROM Sales.Orders
WHERE orderdate = '20060710'
 OR orderdate = '20060711';
```

You can see that SQL Server executed both queries by using the same execution plan, as shown in Figure 15-11.

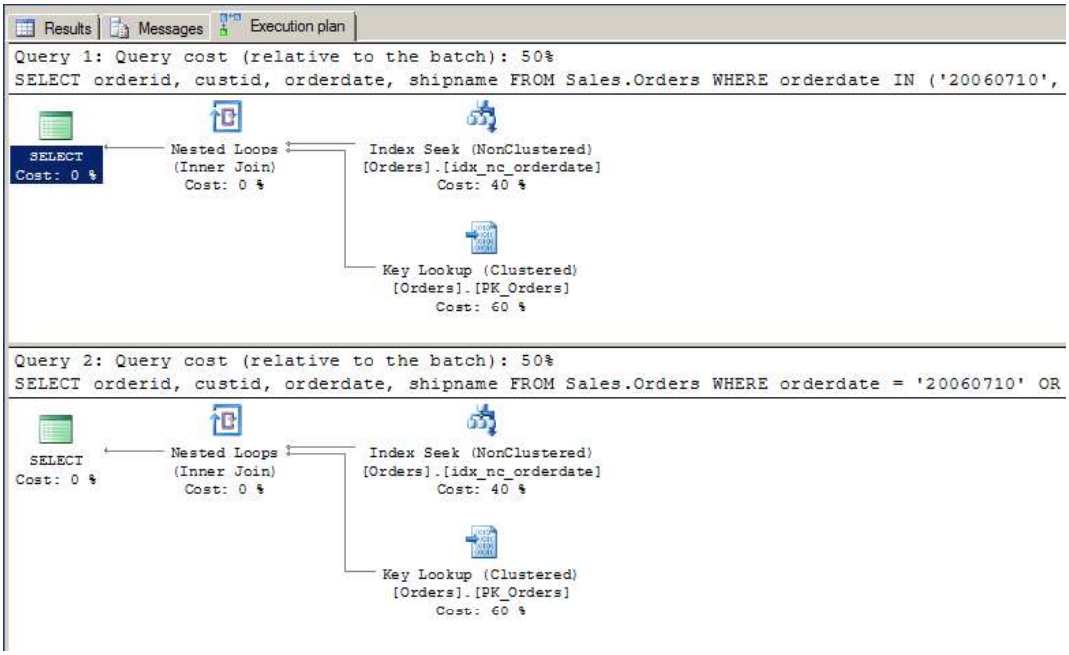

FIGURE 15-11 SQL Server executing the IN and the OR operators in the same way.

Using the AND operator in the WHERE clause predicate means that each part of the predicate limits the result set even more than the previous part. For example, if the first condition limits a query to five rows, then the next condition connected to the first one with the logical AND operator limits the query to five rows at most. The Query Optimizer understands how the logical AND operator works, and can use appropriate indexes.

However, the logical OR operator is inclusive. For example, if the first condition in a predicate would limit the query to 5 rows and the next condition connected to the first condition with the logical OR operator would limit the query to 6 rows, then the result set could have anything between 6 and 11 rows. If the two conditions use two different columns, then SQL Server conservatively takes the worst case and estimates that the query would return 11 rows. Having multiple conditions in a predicate connected with the OR operator lowers the possibility for SQL Server to use indexes. You should consider rewriting the predicate to a logically equivalent predicate that uses the AND operator.

Quick Check

- **Which clauses of a query should you consider supporting with an index?**

Quick Check Answer

- **The list of the clauses you should consider supporting with an index includes, but is not limited to, the WHERE, JOIN, GROUP BY, and ORDER BY clauses.**

PRACTICE Using the OR and AND Logical Operators

In this practice, you use the OR and AND logical operators to connect two conditions of a predicate of a query, and then you check the execution plans.

If you encounter a problem completing an exercise, you can install the completed projects from the Solution folder that is provided with the companion content for this chapter and lesson.

EXERCISE 1 Support the OR Logical Operators

In this exercise, you test the influence of the OR logical operator on query execution. You are going to support queries with appropriate indexes.

1. If you closed SSMS, start it and connect to your SQL Server instance.
2. Open a new query window by clicking the New Query button.
3. Change the context to the TSQL2012 database.
4. Create a nonclustered index on the Sales.Orders table on the shipcity column.

```
CREATE NONCLUSTERED INDEX idx_nc_shipcity ON Sales.Orders(shipcity);
```

5. Retrieve the orderid, custid, and shipcity columns for the city of Vancouver.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

6. The query is selective, because it returns three rows only. Check whether the nonclustered index you just created was used by using the following code.

```
SELECT OBJECT_NAME(S.object_id) AS table_name,
 I.name AS index_name,
 S.user_seeks, S.user_scans, s.user_lookups
FROM sys.dm_db_index_usage_stats AS S
 INNER JOIN sys.indexes AS i
  ON S.object_id = I.object_id
   AND S.index_id = I.index_id
WHERE S.object_id = OBJECT_ID(N'Sales.Orders', N'U')
 AND I.name = N'idx_nc_shipcity';
```

   You should get a single row showing the index was used for a seek.

7. Turn on the actual execution plan. Now retrieve the same columns for the customer with an ID of 42.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42;
```

Note that the same three rows were returned. SQL Server used the idx_nc_custid non-clustered index this time, as you can see from the execution plan. The index uses the custid column for its key.

8. Retrieve the same result set again. However, now include both conditions, the city Vancouver and the customer ID of 42, in the WHERE clause, connected with the OR operator, like the following code shows.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42
 OR shipcity = N'Vancouver';
```

9. Again, the same three rows were retrieved. However, you can see from the execution plan that this time SQL Server scanned the clustered index. By pausing on the Clustered Index Scan operator in the plan, you can see the estimated and actual number of rows, as shown in Figure 15-12.

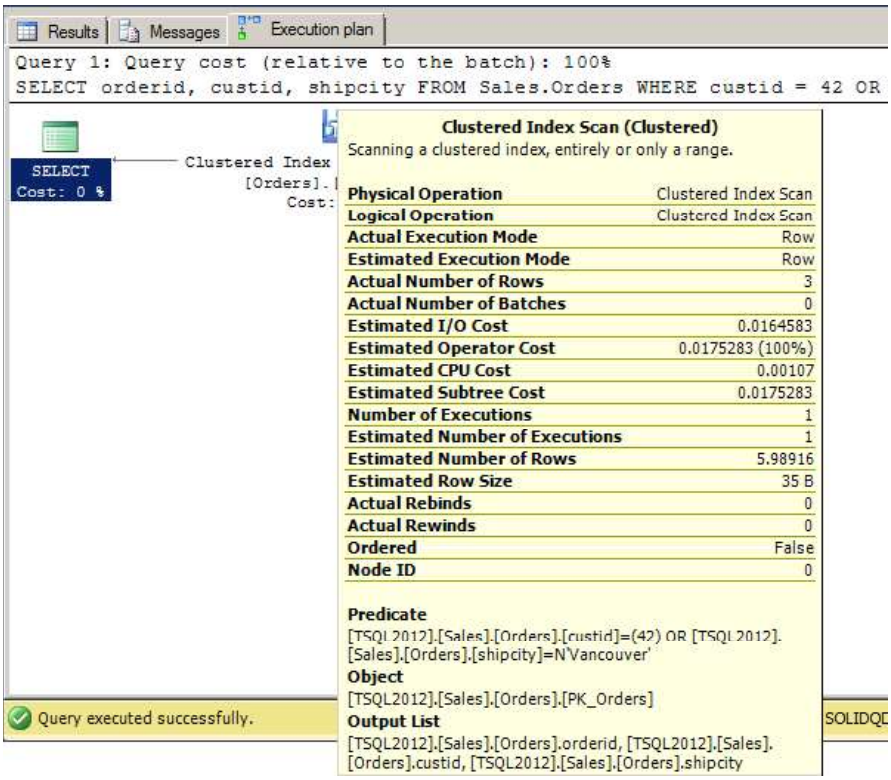

FIGURE 15-12 Estimated and actual number of rows for the Clustered Index Scan operator.

You can see that the estimated number of rows is about six, and SQL Server decided to scan the clustered index.

EXERCISE 2 Support the AND Logical Operator

In this exercise, you test the influence of the AND logical operator on query execution. You also create an index that has an included column.

1. Change the last query from the previous exercise by replacing the OR operator with the AND operator.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42
 AND shipcity = N'Vancouver';
```

   Of course, you are still retrieving the same three rows. However, from the execution plan, you can see that SQL Server used the idx_nc_custid nonclustered index this time.

2. Drop the nonclustered index on the Sales.Orders table on the shipcity column.

```
DROP INDEX idx_nc_shipcity ON Sales.Orders;
```

3. Create a nonclustered index on the Sales.Orders table on the shipcity column again. This time, include the custid column. Use the following code.

```
CREATE NONCLUSTERED INDEX idx_nc_shipcity_i_custid ON Sales.Orders(shipcity)
INCLUDE (custid);
```

4. Again execute the query that includes both conditions, the city Vancouver and the customer ID of 42, in the WHERE clause, connected with the OR operator.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42
 OR shipcity = N'Vancouver';
```

5. You should get a nonclustered index scan. The query is covered by the index with the included column you just created. However, SQL Server still used a scan, because of the OR operator. Change the OR operator to AND again and execute the query.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42
 AND shipcity = N'Vancouver';
```

   This time, SQL Server should use the Index Seek operator to seek for the first occurrence of the city of Vancouver and then perform a partial scan.

6. Turn off the actual execution plan and drop the index you created.

```
DROP INDEX idx_nc_shipcity_i_custid ON Sales.Orders;
```

Lesson Summary

- You support different parts of queries with indexes.
- Consider supporting the WHERE, JOIN, GROUP BY, ORDER BY, and SELECT clauses of queries with appropriate indexes.
- You write appropriate search arguments by not including key columns of indexes in expressions.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. How can you support the SELECT clause of a query by using a nonclustered index that is already used for the WHERE clause?
   - **A.** You could use SELECT *.
   - **B.** You could modify the index that is already used to include the columns from the select list that are not part of the key.
   - **C.** You could add column aliases.
   - **D.** There is no way to support the SELECT clause with indexes.
2. Where does SQL Server sort the data, if a sort is needed?
   - **A.** In the current database
   - **B.** In the master database
   - **C.** In the msdb database
   - **D.** SQL Server sorts data in memory, or spills the data to tempdb if it does not fit in memory.
3. You create an index to support the WHERE clause of a query. However, SQL Server does not use the index. What are the possible reasons? (Choose all that apply.)
   - **A.** The arguments in the predicate are not searchable.
   - **B.** SQL Server does not consider using an index to support the WHERE clause.
   - **C.** The predicate is not selective enough.
   - **D.** You are in the context of the tempdb database, and SQL Server does not use indexes in this database.

Lesson 3: Understanding Statistics

You might have asked yourself when reading and testing the code from the previous lesson how SQL Server knows in advance whether a query is selective enough to perform an index seek. There is no magic behind it; SQL Server maintains statistics of the distribution of key values in special system statistical pages. The Query Optimizer uses these statistics to estimate the cardinality, or number of rows, in the query result set. You learn about statistics in this lesson.

After this lesson, you will be able to:

- Understand SQL Server statistics.
- Manually maintain statistics.

Estimated lesson time: 25 minutes

Auto-Created Statistics

By default, SQL Server creates statistics automatically. SQL Server creates statistics for each index, and for single columns used as searchable arguments in queries. There are three database options that influence the automatic creation of the statistics:

- **AUTO_CREATE_STATISTICS** When this option is set to on, SQL Server creates statistics automatically. This option is on by default, and you should leave this option on in the vast majority of cases.
- **AUTO_UPDATE_STATISTICS** This option, when turned on, enables SQL Server to automatically update statistics when there are enough changes in the underlying tables and indexes. With this option on, SQL Server also updates an out-of-date statistic during query optimization. SQL Server checks for outdated statistics before compiling a query and before executing a cached query. In general, you should leave this option turned on.
- **AUTO_UPDATE_STATISTICS_ASYNC** This option determines whether SQL Server uses synchronous or asynchronous statistics updates during query optimization. If the statistics are updated asynchronously, SQL Server cannot use them for the optimization of the query that triggered the update; however, SQL Server does not wait for the statistics update during the optimization phase. You should turn this option on only if your queries wait for synchronous updates of statistics too frequently and this causes performance problems.

Each statistics object is stored in a statistics binary large object and is created on one or more columns. The statistics include a histogram with the distribution of values in the first column. Statistics objects on multiple columns store additional statistical information about the correlation of values among the columns. These correlation statistics are also called densities. They are derived from the number of distinct rows of combinations of values of columns of a composite index.

There is a limit for the number of steps in histograms. A statistic can have maximally 200 steps. The statistics object also includes a header with metadata about the statistics, and a density vector to measure cross-column correlation. SQL Server computes an estimated number of rows that a query returns, or a cardinality estimate, with any of the data in the statistics object.

You can get information about statistics by querying the sys.stats and sys.stats_columns catalog views. You can get detailed information about statistics with the DBCC SHOW_STATISTICS command. You can manually maintain statistics with the CREATE, DROP, and UPDATE statistics commands. You can also use the sys.sp_updatestats system procedure to manually update statistics for all tables in a database. For example, the following code uses a cursor on the sys.stats catalog view to loop over all automatically created statistics for the columns that are not used as index keys for the Sales.Orders table, dynamically concatenates the DROP STATISTICS command, and drops these statistics.

```
DECLARE @statistics_name AS NVARCHAR(128), @ds AS NVARCHAR(1000);
DECLARE acs_cursor CURSOR FOR
SELECT name AS statistics_name
FROM sys.stats
WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U')
  AND auto_created = 1;
OPEN acs_cursor;
FETCH NEXT FROM acs_cursor INTO @statistics_name;
WHILE @@FETCH_STATUS = 0
BEGIN
 SET @ds = N'DROP STATISTICS Sales.Orders.' + @statistics_name +';';
 EXEC(@ds);
 FETCH NEXT FROM acs_cursor INTO @statistics_name;
END;
CLOSE acs_cursor;
DEALLOCATE acs_cursor;
```

IMPORTANT **UPDATING STATISTICS FOR ALL TABLES IN A DATABASE**

Using the sys.sp_updatestats system procedure to manually update statistics for all tables in a database can take a long time and use a lot of resources; therefore, be careful when you use this command in a large database. You should use it only during off-peak hours.

Now only the statistics for the indexes should exist, as the following query shows.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS statistics_name, auto_created
FROM sys.stats
WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U');
```

The result of this query is as follows.

```
table_name  statistics_name        auto_created
----------  --------------------  ------------
Orders      PK_Orders             0
Orders      idx_nc_custid         0
Orders      idx_nc_empid          0
Orders      idx_nc_shipperid      0
Orders      idx_nc_orderdate      0
Orders      idx_nc_shippeddate    0
Orders      idx_nc_shippostalcode 0
```

The auto_created column gets a value of 1 for statistics that SQL Server generates automatically for single columns used as searchable arguments during query execution. Before showing the statistics, the following line of code rebuilds the idx_nc_empid and the Sales.Orders table to ensure that SQL Server updated the statistics.

```
ALTER INDEX idx_nc_empid ON Sales.Orders REBUILD;
```

The following command shows the histogram of the idx_nc_empid statistics.

```
DBCC SHOW_STATISTICS(N'Sales.Orders',N'idx_nc_empid') WITH HISTOGRAM;
```

The result of this command is as follows.

```
RANGE_HI_KEY RANGE_ROWS    EQ_ROWS       DISTINCT_RANGE_ROWS  AVG_RANGE_ROWS
------------ ------------- ------------- -------------------- --------------
1            0             123           0                    1
2            0             96            0                    1
3            0             127           0                    1
4            0             156           0                    1
5            0             42            0                    1
6            0             67            0                    1
7            0             72            0                    1
8            0             104           0                    1
9            0             43            0                    1
DBCC execution completed. If DBCC printed error messages, contact your system
administrator.
```

There are only nine steps in the histogram because the empid column has only nine distinct values. If you execute the DBCC SHOW_STATISTICS command without the WITH HISTOGRAM option, you get all statistics information, including the header and the density vector. From the header, you can get useful information like when the statistics were last updated, as the following query does.

```
DBCC SHOW_STATISTICS(N'Sales.Orders',N'idx_nc_empid') WITH STAT_HEADER;
```

Partial output of this query (only the seven leftmost columns) is as follows.

```
Name         Updated              Rows Rows Sampled Steps  Density Average key length
------------ -------------------- ---- ------------ -----  ------- ------------------
idx_nc_empid Mar 24 2012  1:57PM 830  830          9      0       8
```

You can also get information like when the statistics were last updated by using the STATS_DATE() system function.

As mentioned, SQL Server automatically creates statistics for searchable nonkey columns during query execution. To test this, start with the following code that adds a nonclustered index to the Sales.Orders table.

```
CREATE NONCLUSTERED INDEX idx_nc_custid_shipcity ON Sales.Orders(custid, shipcity);
```

The following query retrieves three rows for the customer with the ID of 42 from the Sales.Orders table.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42;
```

Now check whether the auto-created statistics exist. The following query returns zero rows, meaning that there are no auto-created statistics for the Sales.Orders table yet. SQL Server had enough information through the index statistics from idx_nc_custid_shipcity to execute the previous query.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS statistics_name
FROM sys.stats
WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U')
  AND auto_created = 1;
```

Now select the same three rows from the Sales.Orders table; however, this time use the shipcity column as a SARG and limit output to the city of Vancouver.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

The following query checks whether the auto-created statistics exist, and also adds information about the columns for which the statistics were created.

```
SELECT OBJECT_NAME(s.object_id) AS table_name,
 S.name AS statistics_name, C.name AS column_name
FROM sys.stats AS S
 INNER JOIN sys.stats_columns AS SC
  ON S.stats_id = SC.stats_id
 INNER JOIN sys.columns AS C
  ON S.object_id= C.object_id AND SC.column_id = C.column_id
WHERE S.object_id = OBJECT_ID(N'Sales.Orders', N'U')
  AND auto_created = 1;
```

The output of this query is as follows.

```
table_name  statistics_name            column_name
----------  ------------------------  -----------
Orders      _WA_Sys_0000000B_20C1E124  shipcity
```

You can see that SQL Server created statistics for the shipcity column. All auto-created statistics names start with string _WA_Sys_. To clean up, delete the index you created for this test.

```
DROP INDEX idx_nc_custid_shipcity ON Sales.Orders;
```

If you want, you could also drop all auto-created statistics again.

Manually Maintaining Statistics

There are only a few possible reasons to create statistics manually. One example is when a query predicate contains multiple columns that have cross-column relationships; statistics on the multiple columns can help improve the query plan. Statistics on multiple columns contain cross-column densities that are not available in single-column statistics. However, if the columns are already in the same index, the multicolumn statistics object already exists, so you should not create an additional one manually.

Similarly to filtered indexes, you can also create filtered statistics. Statistics created by SQL Server automatically are always created on all rows of a table. If queries frequently select from a subset of rows that has a unique data distribution, filtered statistics can improve query plans.

Sometimes you can get a warning in the execution plan that a particular statistic is missing. You can create this statistic manually. However, before creating it manually, you should verify that AUTO_CREATE_STATISTICS and AUTO_UPDATE_STATISTICS database options are on and that the database is not read-only. If the database is read-only, the Query Optimizer cannot save statistics.

Consider updating statistics manually in the following circumstances:

- When query execution times are slow, and you know that the queries are written correctly and supported with appropriate indexes. Before you use query hints, update the statistics. SQL Server does not consider using the index with outdated statistics. Check also whether auto-updating statistics is turned off for the database.
- When insert operations occur on ascending or descending key columns. Statistics are not updated for every single row; therefore, the number of rows inserted might be too small to trigger a statistics update. If queries select from the recently added rows, the current statistics might not have cardinality estimates for these new values. In addition, bulk inserting rows to a table or truncating a table can change the distribution of data a lot. Queries executed immediately after these operations might get a suboptimal execution plan because the statistics were not updated automatically yet.
- After an upgrade from a previous version of SQL Server. Statistics information can change with a new version of SQL Server; to be on the safe side, you should update the statistics for the upgraded databases.

Quick Check

- How would you quickly update statistics for the whole database after an upgrade?

Quick Check Answer

- You should use the sys.sp_updatestats system procedure.

PRACTICE Manually Maintaining Statistics

In this practice, you manually maintain statistics.

If you encounter a problem completing an exercise, you can install the completed projects from the Solution folder that is provided with the companion content for this chapter and lesson.

EXERCISE 1 Disable Statistics Auto-Creation

In this exercise, you disable statistics auto-creation.

1. If you closed SSMS, start it and connect to your SQL Server instance.
2. Open a new query window by clicking the New Query button.
3. Change the context to the TSQL2012 database.
4. To have a clean start, drop all auto-created statistics for the Sales.Orders table. Use the following code.

```
DECLARE @statistics_name AS NVARCHAR(128), @ds AS NVARCHAR(1000);
DECLARE acs_cursor CURSOR FOR
SELECT name AS statistics_name
FROM sys.stats
WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U')
  AND auto_created = 1;
OPEN acs_cursor;
FETCH NEXT FROM acs_cursor INTO @statistics_name;
WHILE @@FETCH_STATUS = 0
BEGIN
 SET @ds = N'DROP STATISTICS Sales.Orders.' + @statistics_name +';';
 EXEC(@ds);
 FETCH NEXT FROM acs_cursor INTO @statistics_name;
END;
CLOSE acs_cursor;
DEALLOCATE acs_cursor;
```

5. Disable statistics auto-creation for the TSQL2012 database by using the ALTER DATABASE command.

```
ALTER DATABASE TSQL2012
 SET AUTO_CREATE_STATISTICS OFF WITH NO_WAIT;
```

EXERCISE 2 Observe the Effects When Statistics Auto-Creation Is Disabled

In this exercise, you observe the effects on index usage when statistics auto-creation is disabled.

1. Add a composite nonclustered index on the Sales.Orders table by using the custid and shipcity columns for the key.

```
CREATE NONCLUSTERED INDEX idx_nc_custid_shipcity
 ON Sales.Orders(custid, shipcity);
```

2. Turn on the actual execution plan. Use the following query to select the three orders where the shipcity is Vancouver.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

3. Check the execution plan. You should get either a Clustered Index Scan operator with a warning sign on it, or an Index Scan (NonClustered) operator, again with a warning sign on it. You can open the Properties window from the View menu or by pressing the F4 key to check the warnings property. Click the three dots at the end of the text of the Warnings property to get a pop-up window for this property, as Figure 15-13 shows.

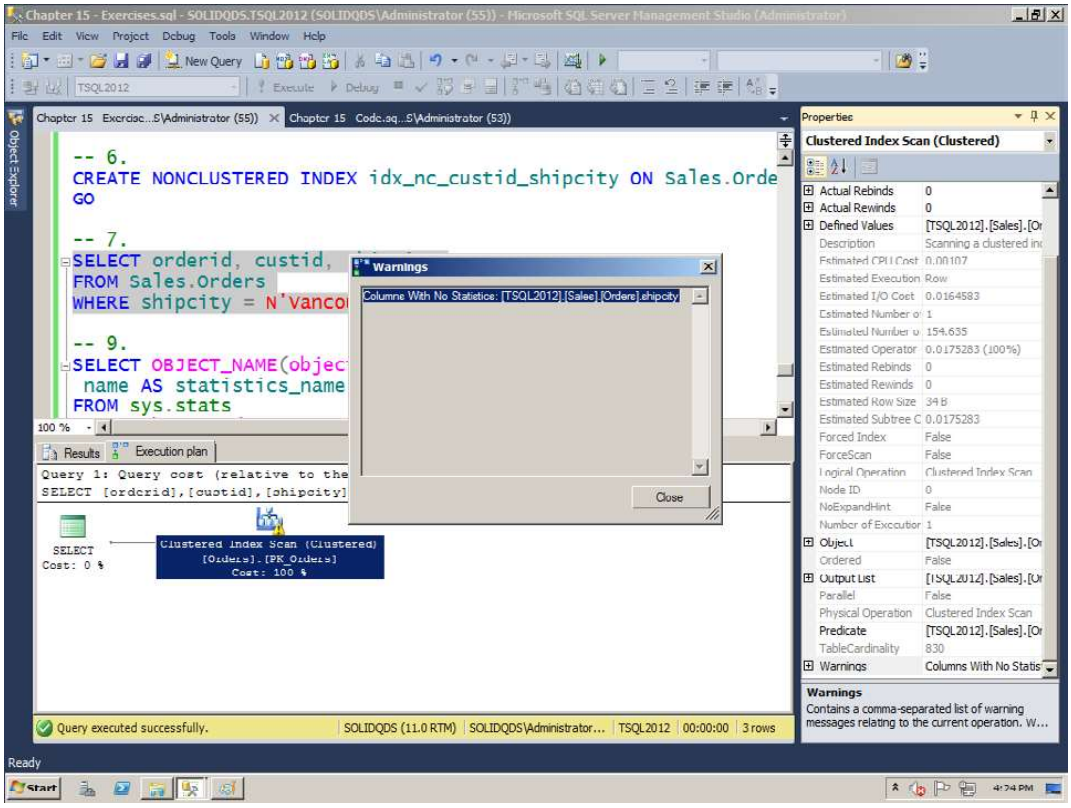

FIGURE 15-13 Missing statistics warning.

4. Check whether any auto-created statistics exist for the Sales.Orders table by using the following query.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS statistics_name
FROM sys.stats
WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U')
  AND auto_created = 1;
```

5. The query should not return any rows. You can create the missing statistics manually. In addition to creating the statistics, you should also clear the cached plan in order to prevent SQL Server from reusing it. Use the following code.

```
CREATE STATISTICS st_shipcity ON Sales.Orders(shipcity);
DBCC FREEPROCCACHE;
```

6. Execute the same query that searches for orders from Vancouver again.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

7. Check the execution plan. You shouldn't get any warning this time.

8. To clean up, turn auto-creating statistics to on, update all statistics in the TSQL2012 database, and drop the index and the statistics you created in this exercise. You can use the following code.

```
ALTER DATABASE TSQL2012
 SET AUTO_CREATE_STATISTICS ON WITH NO_WAIT;
EXEC sys.sp_updatestats;
DROP STATISTICS Sales.Orders.st_shipcity;
DROP INDEX idx_nc_custid_shipcity ON Sales.Orders;
```

9. Exit SSMS.

Lesson Summary

- The SQL Server Query Optimizer uses statistics to determine the cardinality of a query.
- Besides leaving it to SQL Server to maintain statistics automatically, you can also maintain statistics manually.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. How can SQL Server estimate the cardinality of a query?
   A. SQL Server stores the cardinality information on leaf-level pages of indexes.
   B. SQL Server quickly executes the query on 10 percent of sample data.
   C. SQL Server cannot estimate the cardinality of a query if you do not provide a table hint.
   D. SQL Server uses statistics to estimate the cardinality of a query.
2. Which of the following is not a reason to update statistics manually?
   A. You just rebuilt an index.
   B. You bulk-inserted a large amount of data to a table and want to query this table immediately after the insert.
   C. You upgraded the database.
   D. Query execution times are slow; however, you know that the queries are written correctly and supported with appropriate indexes.
3. What is the limit for the number of steps in statistic histograms?
   A. 10 steps per histogram
   B. 200 histograms per column
   C. 200 pages per histogram
   D. 200 steps per histogram

Case Scenarios

In the following case scenarios, you apply what you've learned about implementing indexes and statistics. You can find the answers to these questions in the "Answers" section at the end of this chapter.

Case Scenario 1: Table Scans

Database administrators from a company where you are maintaining a SQL Server database complain that SQL Server scans entire tables for most of the queries, although the queries are very selective. The performance is not acceptable. You need to help them improve the performance.

1. What physical structures should you check?
2. Would you check some code as well?

Case Scenario 2: Slow Updates

End users from a company where you are responsible for the database optimization complain that data updates are slow, even when updating a single row. Seeking for the updated row is supported by appropriate indexes. SELECT queries are performing well. This is what you expected, because you created nonclustered indexes on all columns used in these queries. You need to improve the performance of the database for updates as well.

1. What would you suspect to be the reason for slow updates?
2. How would you investigate for possible problems?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

Learn More About Indexes and How Statistics Influence Query Execution

Using proper indexing and maintaining statistics are very important DBA tasks. You can learn about these two tasks by performing the next two practices.

- **Practice 1** In order to understand how statistical information influences query execution, create a test database with a test table in it. Turn automatic statistics creation and maintenance off for that database. Create a clustered index and one or more nonclustered indexes on that table. Fill the table with test data. Execute test queries and check whether SQL Server used the indexes. Manually create statistics and execute the queries again. Check whether SQL Server used the indexes this time. Add more rows to the table and execute the queries again. After checking index usage, manually update the statistics. Execute queries for the last time and check index usage.
- **Practice 2** Create a table with at least 10 columns. Insert 1,000,000 rows in a loop and measure the time needed for these inserts. Create a nonclustered index on each column. Insert 1,000,000 rows in a loop and measure the time needed for these inserts. You should be able to notice the difference and realize that index maintenance takes some SQL Server resources.

Answers

This section contains answers to the lesson review questions and solutions to the case scenarios in this chapter.

Lesson 1

1. **Correct Answers: A, C, and D**
   - **A. Correct:** An index can have zero or more intermediate levels.
   - **B. Incorrect:** A heap is a separate structure, not a level of an index.
   - **C. Correct:** Every index has the root level, with a single root page.
   - **D. Correct:** The lowest level of an index is the leaf level.
2. **Correct Answer: C**
   - **A. Incorrect:** You can create up to 999 nonclustered indexes on a table.
   - **B. Incorrect:** You can have up to 16 columns in a composite key.
   - **C. Correct:** There can be only one clustered index, because this is the table itself, organized as a balanced tree.
   - **D. Incorrect:** The size of the columns in a key must not exceed 900 bytes.
3. **Correct Answer: C**
   - **A. Incorrect:** RID is used for heaps.
   - **B. Incorrect:** Columns in a columnstore index are not used as row locators.
   - **C. Correct:** The clustering key is the row locator when a table is stored as a balanced tree.
   - **D. Incorrect:** A clustered table is stored as a balanced tree.

Lesson 2

1. **Correct Answer: B**
   - **A. Incorrect:** Using SELECT * is a very bad practice and of course does not help SQL Server to use indexes at all.
   - **B. Correct:** You could modify the index that is already used to include the columns from the SELECT list that are not part of the key.
   - **C. Incorrect:** Adding column aliases has no influence on index usage.
   - **D. Incorrect:** You can support the SELECT clause with indexes.

2. **Correct Answer: D**

   A. **Incorrect:** SQL Server sorts data in memory, or spills the data to tempdb if it does not fit in memory.

   B. **Incorrect:** SQL Server sorts data in memory, or spills the data to tempdb if it does not fit in memory.

   C. **Incorrect:** SQL Server sorts data in memory, or spills the data to tempdb if it does not fit in memory.

   D. **Correct:** SQL Server sorts data in memory, or spills the data to tempdb if it does not fit in memory.

3. **Correct Answers: A and C**

   A. **Correct:** SQL Server does not use an index to support the WHERE clause if the arguments in the predicate are not searchable.

   B. **Incorrect:** SQL Server supports the WHERE clause with indexes.

   C. **Correct:** SQL Server might decide not to use an index to support the WHERE clause if the query is not selective enough.

   D. **Incorrect:** SQL Server considers using indexes in the context of the tempdb database just like in the context of any other database.

Lesson 3

1. **Correct Answer: D**

   A. **Incorrect:** There is no cardinality information on leaf-level pages of indexes.

   B. **Incorrect:** SQL Server does not execute a query in advance on sample data.

   C. **Incorrect:** SQL Server can estimate the cardinality of a query.

   D. **Correct:** SQL Server uses statistics to estimate the cardinality of a query.

2. **Correct Answer: A**

   A. **Correct:** When you rebuild an index, SQL Server updates the statistics automatically.

   B. **Incorrect:** You should update statistics for a table after you bulk-inserted a large amount of data to the table and want to query this table immediately.

   C. **Incorrect:** You should update statistics for the complete database after an upgrade.

   D. **Incorrect:** You should update statistics when queries execute slowly and you know that the queries are written correctly and supported with appropriate indexes.

3. **Correct Answer: D**
   A. **Incorrect:** You can have up to 200 steps in a histogram.
   B. **Incorrect:** You have one histogram per statistics.
   C. **Incorrect:** There is a limit of steps per histogram.
   D. **Correct:** You can have up to 200 steps in a histogram.

Case Scenario 1

1. You should check whether the queries are supported by indexes. In addition, you should check whether the statistics for the indexes are created and updated.
2. You should check whether the queries use appropriate search arguments.

Case Scenario 2

1. Too many indexes might slow updates. You probably created many indexes that are useless; however, because SQL Server has to maintain them, the updates are slow.
2. You can query the sys.dm_db_index_usage_stats dynamic management object to find which indexes are used for seeks and which are used for updates only.

CHAPTER 16

Understanding Cursors, Sets, and Temporary Tables

Exam objectives in this chapter:

- Troubleshoot & Optimize
  - Evaluate the use of row-based operations vs. set-based operations.

This chapter covers two main topics. It starts with a lesson about the differences between row-by-row operations and set-based operations. It then continues with a lesson about the use of temporary objects like local temporary tables and table variables, explaining when you should consider using each kind.

Lessons in this chapter:

- Lesson 1: Evaluating the Use of Cursor/Iterative Solutions vs. Set-Based Solutions
- Lesson 2: Using Temporary Tables vs. Table Variables

Before You Begin

To complete the lessons in this chapter, you must have:

- Experience working with Microsoft SQL Server Management Studio (SSMS).
- Access to a SQL Server 2012 instance with the sample database TSQL2012 installed.
- An understanding of filtering and sorting data.
- An understanding of combining sets.
- An understanding of grouping and windowing.
- An understanding of creating tables and enforcing data integrity.
- An understanding of modifying data.
- An understanding of indexing.