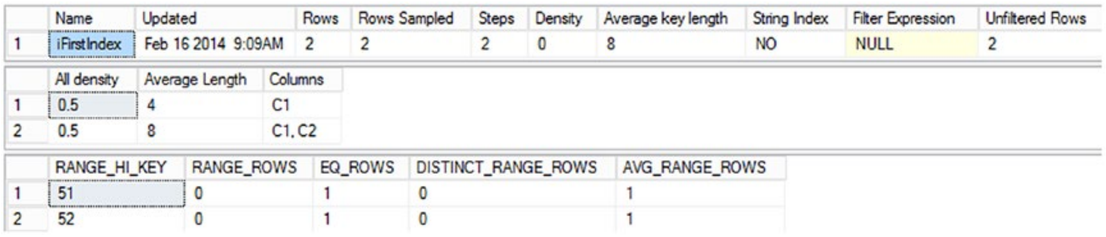
The more traditional, and safer, approach to identify outdated statistics is to examine how close the optimizer's estimation of the number of rows affected is to the actual number of rows affected.

The following example shows you how to identify and resolve an outdated statistics issue. Figure 13-38 shows the statistics on the nonclustered index key on column C1 provided by DBCC SHOW_STATISTICS.

```
DBCC SHOW_STATISTICS (Test1, iFirstIndex);
```



	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	iFirstIndex	Feb 16 2014 9:09AM	2	2	2	0	8	NO	NULL	2

	All density	Average Length	Columns
1	0.5	4	C1
2	0.5	8	C1, C2

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	51	0	1	0	1
2	52	0	1	0	1

***Figure 13-38.** Statistics on index FirstIndex*

These results say that the density value for column C1 is 0.5. Now consider the following SELECT statement:

```
SELECT   *
FROM     dbo.Test1
WHERE    C1 = 51;
```

Since the total number of rows in the table is currently 10,002, the number of matching rows for the filter criteria C1 = 51 can be estimated to be 5,001 (= 0.5 × 10,002). This estimated number of rows (5,001) is way off the actual number of matching rows for this column value. The table actually contains only one row for C1 = 51.

You can get the information on both the estimated and actual number of rows from the execution plan. An estimated plan refers to and uses the statistics only, not the actual data. This means it can be wildly different from the real data, as you're seeing now. The actual execution plan, on the other hand, has both the estimated and actual numbers of rows available.

Executing the query results in the execution plan in Figure 13-39 and the following performance:

```
Reads: 100
Duration: 681 mc
```

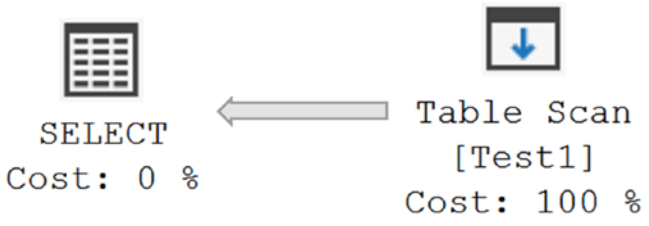
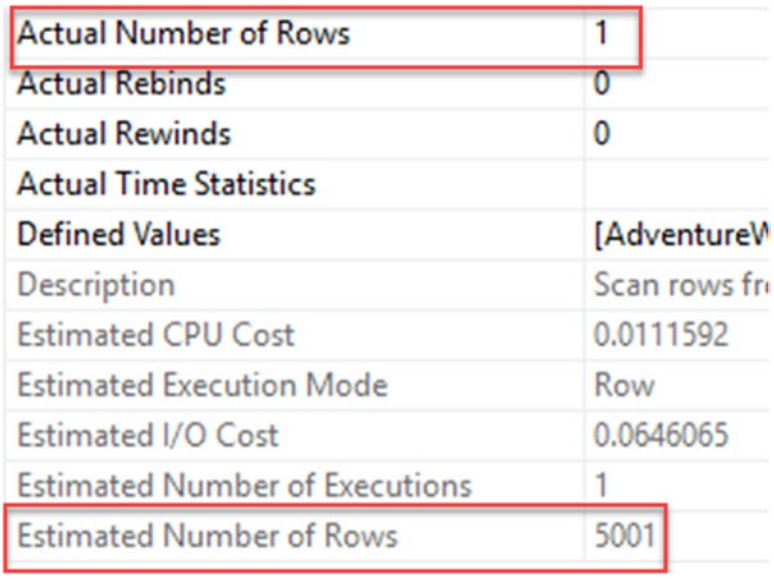


***Figure 13-39.** Execution plan with outdated statistics*

To see the estimated and actual rows, you can view the properties of the `Table Scan` operator (Figure 13-40).

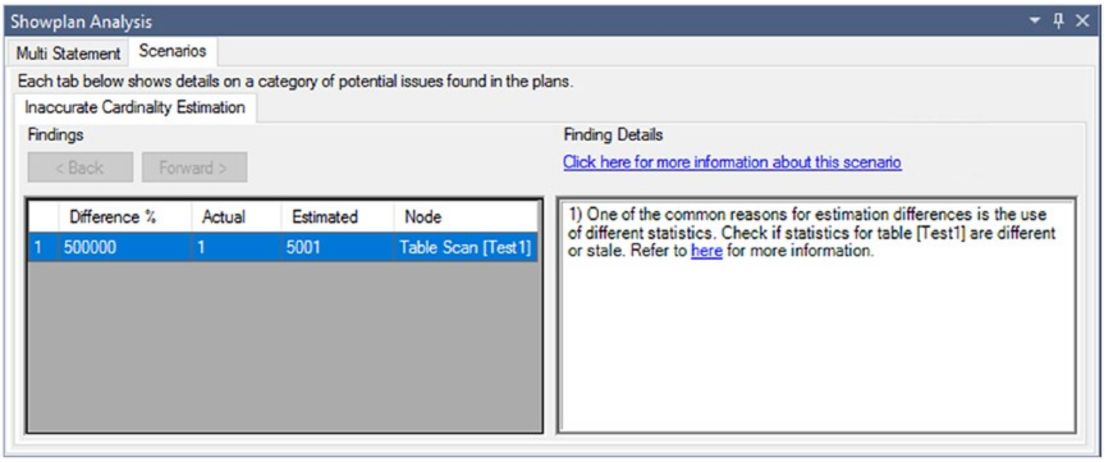


Actual Number of Rows	1
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Defined Values	[AdventureW
Description	Scan rows fr
Estimated CPU Cost	0.0111592
Estimated Execution Mode	Row
Estimated I/O Cost	0.0646065
Estimated Number of Executions	1
Estimated Number of Rows	5001

***Figure 13-40.** Properties showing row count discrepancy*

From the estimated rows value versus the actual rows value, it's clear that the optimizer made an incorrect estimation based on out-of-date statistics. If the difference between the estimated rows and actual rows is more than a factor of 10, then it's quite possible that the processing strategy chosen may not be very cost-effective for the current data distribution. An inaccurate estimation may misguide the optimizer in deciding the processing strategy. Statistics can be off for a number of reasons. Table variables and multistatement user-defined functions don't have statistics at all, so all estimates for these objects assume a single row, without regard to how many rows are actually involved with the objects.

We can also use the Showplan Analysis feature to see the Inaccurate Cardinality Estimation report. Right-click an actual plan and select Analyze Actual Execution Plan from the context menu. When the analysis window opens, select the Scenarios tab. For the previous plan, you'll see something like Figure 13-41.


Showplan Analysis
Multi Statement
Scenarios
Inaccurate Cardinality Estimation
Findings
< Back
Forward >
Finding Details
Click here for more information about this scenario

Each tab below shows details on a category of potential issues found in the plans.

	Difference %	Actual	Estimated	Node
1	500000	1	5001	Table Scan [Test1]

1) One of the common reasons for estimation differences is the use of different statistics. Check if statistics for table [Test1] are different or stale. Refer to here for more information.

Figure 13-41. *Inaccurate Cardinality Estimation report showing the difference between actual and estimated*

To help the optimizer make an accurate estimation, you should update the statistics on the nonclustered index key on column C1 (alternatively, of course, you can just leave the auto update statistics feature on).

```
UPDATE STATISTICS Test1 iFirstIndex WITH FULLSCAN;
```

A FULLSCAN might not be needed here. The sampled method of statistics creation is usually fairly accurate and is much faster. But, on systems that aren't experiencing stress, or during off-hours, I tend to favor using FULLSCAN because of the improved accuracy. Either approach is valid as long as you're getting the statistics you need.

If you run the query again, you'll get the following statistics, and the resultant output is as shown in Figure 13-42:

```
Reads: 4
Duration: 184mc
```

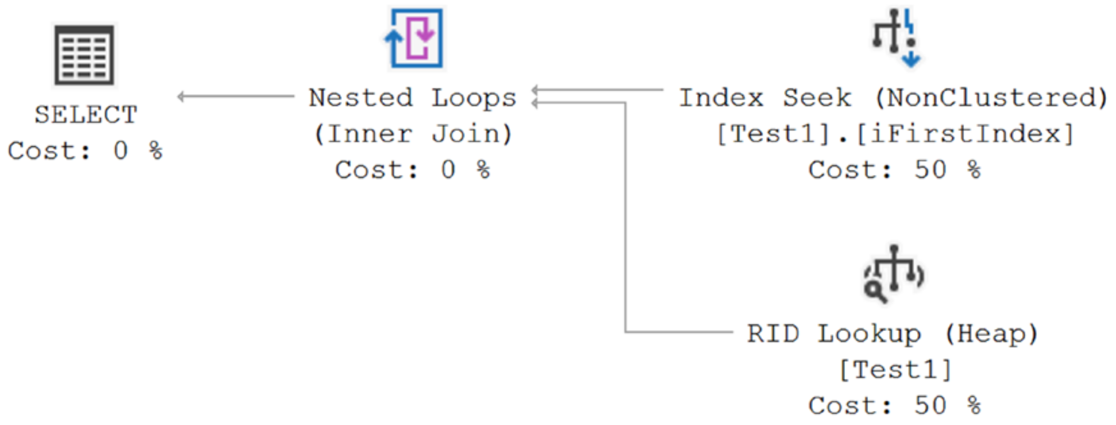


Figure 13-42. *Actual and estimated number of rows with up-to-date statistics*

The optimizer accurately estimated the number of rows using updated statistics and consequently was able to come up with a more efficient plan. Since the estimated number of rows is 1, it makes sense to retrieve the row through the nonclustered index on C1 instead of scanning the base table.

Updated, accurate statistics on the index key column help the optimizer come to a better decision on the processing strategy and thereby reduce the number of logical reads from 84 to 4 and reduce the execution time from 16ms to -0ms (there is a -4ms lag time).

Before continuing, turn the statistics back on for the database.

```
ALTER DATABASE AdventureWorks2017 SET AUTO_CREATE_STATISTICS ON;
ALTER DATABASE AdventureWorks2017 SET AUTO_UPDATE_STATISTICS ON;
```

Recommendations

Throughout this chapter, I covered various recommendations for statistics. For easy reference, I've consolidated and expanded upon these recommendations in the sections that follow.

Backward Compatibility of Statistics

Statistical information in SQL Server 2014 and greater can be generated differently from that in previous versions of SQL Server. SQL Server transfers the statistics during upgrade and, by default, automatically updates these statistics over time as the data changes.

The best approach is to follow the directions outlined in Chapter 10 on the Query Store and let the statistics update over time.

Auto Create Statistics

This feature should usually be left on. With the default setting, during the creation of an execution plan, SQL Server determines whether statistics on a nonindexed column will be useful. If this is deemed beneficial, SQL Server creates statistics on the nonindexed column. However, if you plan to create statistics on nonindexed columns manually, then you have to identify exactly for which nonindexed columns statistics will be beneficial.

Auto Update Statistics

This feature should usually be left on, allowing SQL Server to decide on the appropriate execution plan as the data distribution changes over time. Usually the performance benefit provided by this feature outweighs the cost overhead. You will seldom need to interfere with the automatic maintenance of statistics, and such requirements are usually identified while troubleshooting or analyzing performance. To ensure that you aren't facing surprises from the automatic statistics features, it's important to analyze the effectiveness of statistics while diagnosing SQL Server issues.

Unfortunately, if you come across an issue with the auto update statistics feature and have to turn it off, make sure to create a SQL Server job to update the statistics and schedule it to run at regular intervals. For performance reasons, where possible, ensure that the SQL job is scheduled to run during off-peak hours.

One of the best approaches to statistics maintenance is to run the scripts developed and maintained by Ola Holengren (`http://bit.ly/JijaNI`).

Automatic Update Statistics Asynchronously

Waiting for statistics to be updated before plan generation, which is the default behavior, will be just fine in most cases. In the rare circumstances where the statistics update or the execution plan recompiles resulting from that update are expensive (more expensive than the cost of out-of-date statistics), then you can turn on the asynchronous update of statistics. Just understand that it may mean that procedures that would benefit from more up-to-date statistics will suffer until the next time they are run. Don't forget—you do need automatic update of statistics enabled to enable the asynchronous updates.

Amount of Sampling to Collect Statistics

It is generally recommended that you use the default sampling rate. This rate is decided by an efficient algorithm based on the data size and number of modifications. Although the default sampling rate turns out to be best in most cases, if for a particular query you find that the statistics are not very accurate or missing critical data distributions, then you can manually update them with `FULLSCAN`. You also have the option of setting a specific sample percentage using the `SAMPLE` number. The number can be either a percentage or a set number of rows.

If this is required repeatedly, then you can add a SQL Server job to take care of it. For performance reasons, ensure that the SQL job is scheduled to run during off-peak hours. To identify cases in which the default sampling rate doesn't turn out to be the best, analyze the statistics effectiveness for costly queries while troubleshooting the database performance. Remember that `FULLSCAN` is expensive, so you should run it only on those tables or indexes that you've determined will really benefit from it.

Summary

As discussed in this chapter, SQL Server's cost-based optimizer requires accurate statistics on columns used in filter and join criteria to determine an efficient processing strategy. Statistics on an index key are always created during the creation of the index, and, by default, SQL Server also keeps the statistics on indexed and nonindexed columns updated as the data changes. This enables it to determine the best processing strategies applicable to the current data distribution.

Even though you can disable both the auto create statistics and auto update statistics features, it is recommended that you leave these features *on,* since their benefit to the optimizer is almost always more than their overhead cost. For a costly query, analyze the statistics to ensure that the automatic statistics maintenance lives up to its promise. The best news is that you can rest easy with a little vigilance since automatic statistics do their job well most of the time. If manual statistics maintenance procedures are used, then you can use SQL Server jobs to automate these procedures.

Even with proper indexes and statistics in place, a heavily fragmented database can incur an increased data retrieval cost. In the next chapter, you will see how fragmentation in an index can affect query performance, and you'll learn how to analyze and resolve fragmentation where needed.

CHAPTER 14

Index Fragmentation

As explained in Chapter 8, rowstore index column values are stored in the leaf pages of an index's B-tree structure. Columnstore indexes are also stored in pages, but not within a B-tree structure. When you create an index (clustered or nonclustered) on a table, the cost of data retrieval is reduced by properly ordering the leaf pages of the index and the rows within the leaf pages, whereas a columnstore has the data pivoted into columns and then compressed, again with the intent of assisting in data retrieval. In an OLTP database, data changes continually, causing fragmentation of the indexes. As a result, the number of reads required to return the same number of rows increases over time. A similar situation occurs with the columnstore as data is moved from the deltastore to the segmented storage areas. Both these situations can lead to performance degradation.

In this chapter, I cover the following topics:

- The causes of index fragmentation, including an analysis of page splits caused by `INSERT` and `UPDATE` statements
- The causes of columnstore index fragmentation
- The overhead costs associated with fragmentation
- How to analyze the amount of fragmentation in rowstore and columnstore indexes
- Techniques used to resolve fragmentation
- The significance of the fill factor in helping to control fragmentation in the rowstore indexes
- How to automate the fragmentation analysis process

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_14

Discussion on Fragmentation

There is currently a lot of discussion in the data platform community as to the extent that fragmentation is any kind of an issue at all. Before we get into the full discussion of what fragmentation is, how it may affect your queries, and how you can deal with it if it does, we should immediately address this question: should you defragment your indexes? I have decided to put this discussion ahead of all the details of how fragmentation works, so if that's still a mystery, please skip this section and go straight to "Causes of Fragmentation."

When your indexes and tables are fragmented, they do take up more space, meaning more pages. This spreads them across the disk in different ways depending on the type of index. When dealing with a fragmented index and a point lookup, or a very limited range scan, the fragmentation won't affect performance at all. When dealing with a fragmented index and large scans, having to move through more pages on the disk certainly impacts performance. Taken from this point, you could simply defragment your indexes only if you have lots of scans and large data movement.

However, there's more to it. Defragmentation itself puts a load on the system, causing blocking and additional work that affects the performance of the system. Then, your indexes start the process of fragmenting again, including page splits and row rearrangement, again causing performance headaches. A strong argument can be made that allowing the system to find an equilibrium where the pages are empty enough that the splits stop (or are radically reduced) will achieve better overall performance. This is because instead of stressing the system to rebuild the indexes and then suffering through all the page splits as the indexes shift again, you just reduce the page splits. You're still dealing with slower scans, but with modern disk subsystems, this is not as much of a headache.

With all this in mind, I lean heavily toward the "stop defragmenting your indexes" camp. As long as you set an appropriate fill factor, you should see radical reductions in page split activity. However, your best bet is to use the tools outlined in this chapter to monitor your system. Chances are high all of us are in a mixed mode where some tables and indexes need to be defragmented and others should be left alone. It all comes down to the behavior of your system.

Causes of Fragmentation

Fragmentation occurs when data is modified in a table. This is true of both rowstore and columnstore indexes. When you add or remove data in a table (via `INSERT` or `DELETE`), the table's corresponding clustered indexes and the affected nonclustered indexes are modified. The two types of indexes, rowstore and columnstore, vary a little from this point. We'll address them one at a time starting with the rowstore.

Data Modification and the Rowstore Indexes

Modifying data through `INSERT`, `UPDATE`, or `MERGE` can cause an index leaf page split if the modification to an index can't be accommodated in the same page. Removing data through `DELETE` simply leaves gaps in the existing pages. When a page split occurs, a new leaf page will then be added that contains part of the original page and maintains the logical order of the rows in the index key. Although the new leaf page maintains the *logical* order of the data rows in the original page, this new page usually won't be *physically* adjacent to the original page on the disk. Put a slightly different way, the logical key order of the index doesn't match the physical order within the file.

For example, suppose an index has nine key values (or index rows) and the average size of the index rows allows a maximum of four index rows in a leaf page. As explained in Chapter 9, the 8KB leaf pages are connected to the previous and next leaf pages to maintain the logical order of the index. Figure 14-1 illustrates the layout of the leaf pages for the index.

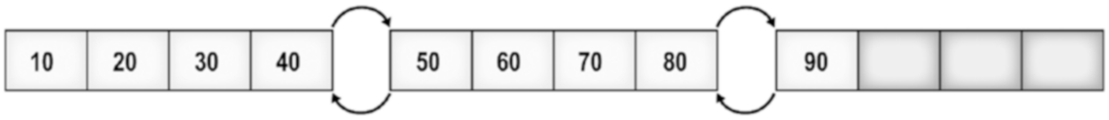


Figure 14-1. *Leaf pages layout*

Since the index key values in the leaf pages are always sorted, a new index row with a key value of 25 has to occupy a place between the existing key values 20 and 30. Because the leaf page containing these existing index key values is full with the four index rows, the new index row will cause the corresponding leaf page to split. A new leaf page will be assigned to the index, and part of the first leaf page will be moved to this new leaf page so that the new index key can be inserted in the correct logical order. The links between the index pages will also be updated so that the pages are logically connected in the order of the index. As shown in Figure 14-2, the new leaf page, even though linked to the other pages in the correct logical order, can be physically out of order.

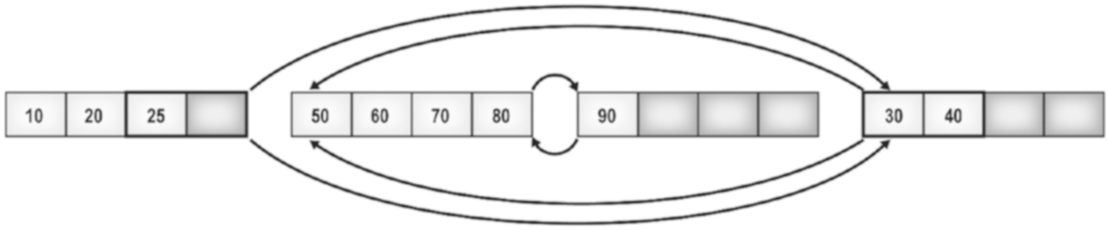


Figure 14-2. *Out-of-order leaf pages*

The pages are grouped together in bigger units called *extents*, which can contain eight pages. SQL Server uses an extent as a physical unit of allocation on the disk. Ideally, the physical order of the extents containing the leaf pages of an index should be the same as the logical order of the index. This reduces the number of switches required between extents when retrieving a range of index rows. However, page splits can physically disorder the pages within the extents, and they can also physically disorder the extents themselves. For example, suppose the first two leaf pages of the index are in extent 1, and say the third leaf page is in extent 2. If extent 2 contains free space, then the new leaf page allocated to the index because of the page split will be in extent 2, as shown in Figure 14-3.

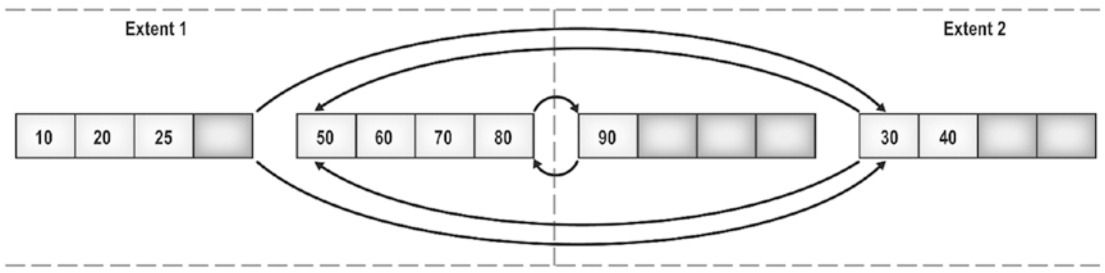


Figure 14-3. *Out-of-order leaf pages distributed across extents*

With the leaf pages distributed between two extents, ideally you expect to read a range of index rows with a maximum of one switch between the two extents. However, the disorganization of pages between the extents can cause more than one extent switch while retrieving a range of index rows. For example, to retrieve a range of index rows between 25 and 90, you will need three extent switches between the two extents, as follows:

- First extent switch to retrieve the key value 30 after the key value 25
- Second extent switch to retrieve the key value 50 after the key value 40
- Third extent switch to retrieve the key value 90 after the key value 80

This type of fragmentation is called *external fragmentation.* External fragmentation can be undesirable.

Fragmentation can also happen within an index page. If an `INSERT` or `UPDATE` operation creates a page split, then free space will be left behind in the original leaf page. Free space can also be caused by a `DELETE` operation. The net effect is to reduce the number of rows included in a leaf page. For example, in Figure 14-3, the page split caused by the `INSERT` operation has created an empty space within the first leaf page. This is known as *internal fragmentation.*

For a highly transactional database, it is desirable to deliberately leave some free space within your leaf pages so that you can add new rows, or change the size of existing rows, without causing a page split. In Figure 14-3, the free space within the first leaf page allows an index key value of 26 to be added to the leaf page without causing a page split.

Note Note that this index fragmentation is different from disk fragmentation. The index fragmentation cannot be fixed simply by running the disk defragmentation tool because the order of pages within a SQL Server file is understood only by SQL Server, not by the operating system.

Heap pages can become fragmented in the same way. Unfortunately, because of how heaps are stored and how any nonclustered indexes use the physical data location for retrieving data from the heap, defragmenting heaps is quite problematic. You can use the `REBUILD` command of `ALTER TABLE` to perform a heap rebuild, but understand that you will force a rebuild of any nonclustered indexes associated with that table.

SQL Server 2017 exposes the leaf and nonleaf pages and other data through a dynamic management view called `sys.dm_db_index_physical_stats`. It stores both the index size and the fragmentation. I'll cover it in more detail in the next section. The DMV is much easier to work with than the old `DBCC SHOWCONTIG`.

Let's now take a look at the mechanics of fragmentation.

Page Split by an UPDATE Statement

To show what happens when a page split is caused by an `UPDATE` statement, I'll use a constructed table. This small test table will have a clustered index, which orders the rows within one leaf (or data) page as follows:

```
USE AdventureWorks2017;
GO
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT,
                        C2 CHAR(999),
                        C3 VARCHAR(10))
INSERT INTO dbo.Test1
VALUES (100, 'C2', "),
       (200, 'C2', "),
       (300, 'C2', "),
       (400, 'C2', "),
       (500, 'C2', "),
       (600, 'C2', "),
       (700, 'C2', "),
       (800, 'C2', ");

CREATE CLUSTERED INDEX iClust ON dbo.Test1 (C1);
```

The average size of a row in the clustered index leaf page (excluding internal overhead) is not just the sum of the average size of the clustered index columns; it's the sum of the average size of all the columns in the table since the leaf page of the clustered index and the data page of the table are the same. Therefore, the average size of a row in the clustered index based on the previous sample data is as follows:

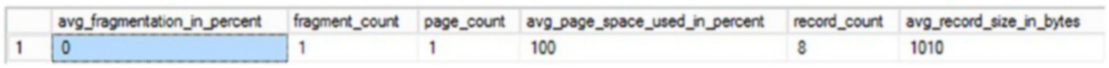
```
= (Average size of [C1]) + (Average size of [C2]) + (Average size of [C3])
bytes = (Size of INT) + (Size of CHAR(999)) + (Average size of data in
[C3]) bytes
= 4 + 999 + 0 = 1,003 bytes
```

The maximum size of a row in SQL Server is 8,060 bytes. Therefore, if the internal overhead is not very high, all eight rows can be accommodated in a single 8KB page.

To determine the number of leaf pages assigned to the `iClust` clustered index, execute the `SELECT` statement against `sys.dm_db_index_physical_stats`.

```
SELECT ddips.avg_fragmentation_in_percent,
       ddips.fragment_count,
       ddips.page_count,
       ddips.avg_page_space_used_in_percent,
       ddips.record_count,
       ddips.avg_record_size_in_bytes
FROM sys.dm_db_index_physical_stats(DB_ID('AdventureWorks2017'),
                                    OBJECT_ID(N'dbo.Test1'),
                                    NULL,
                                    NULL,
                                    'Sampled') AS ddips;
```

You can see the results of this query in Figure 14-4.



	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	0	1	1	100	8	1010

Figure 14-4. *Physical layout of index iClust*

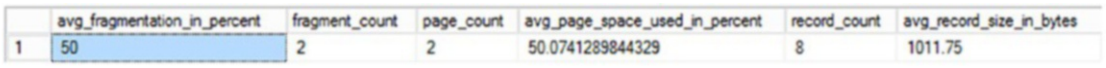
From the page_count column in this output, you can see that the number of pages assigned to the clustered index is 1. You can also see the average space used, 100, in the avg_ page_space_used_in_percent column. From this you can infer that the page has no free space left to expand the content of C3, which is of type VARCHAR(10) and is currently empty.

Note I'll analyze more of the information provided by sys.dm_db_index_ physical_stats in the "Analyzing the Amount of Fragmentation" section later in this chapter.

Therefore, if you attempt to expand the content of column C3 for one of the rows as follows, it should cause a page split:

```
UPDATE dbo.Test1
SET C3 = 'Add data'
WHERE C1 = 200;
```

Selecting the data from sys.dm_db_index_physical_stats results in the information in Figure 14-5.



	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	50	2	2	50.0741289844329	8	1011.75

Figure 14-5. *i1 index after a data update*

From the output in Figure 14-5, you can see that SQL Server has added a new page to the index. On a page split, SQL Server generally moves half the total number of rows in the original page to the new page. Therefore, the rows in the two pages are distributed as shown in Figure 14-6.

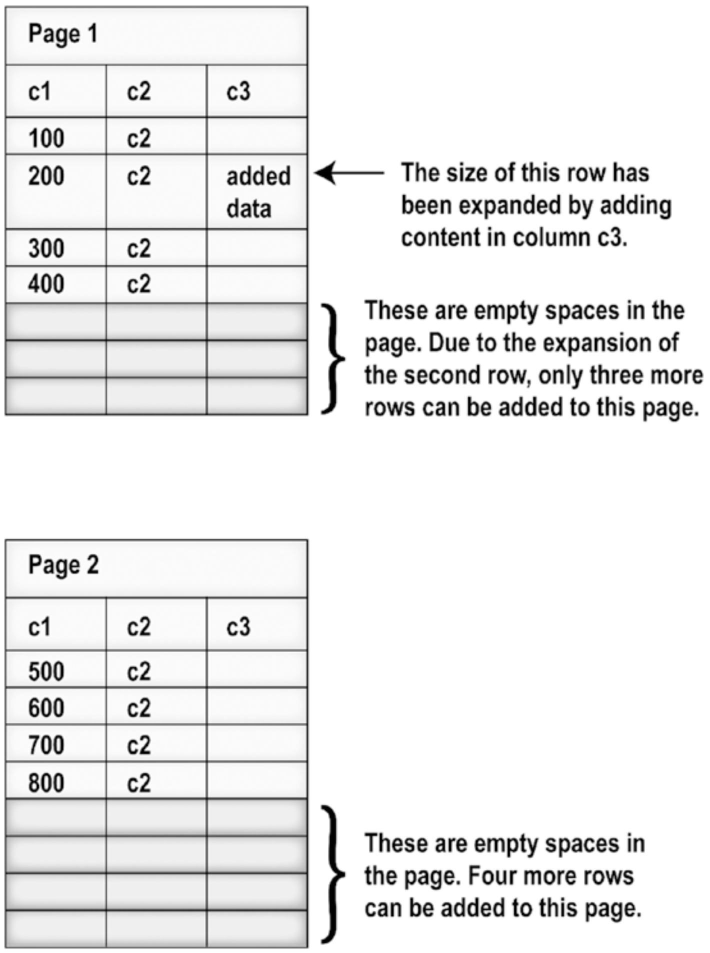

Page 1
c1
c2
c3
100
c2
200
c2
added data
300
c2
400
c2
The size of this row has been expanded by adding content in column c3.
These are empty spaces in the page. Due to the expansion of the second row, only three more rows can be added to this page.
Page 2
c1
c2
c3
500
c2
600
c2
700
c2
800
c2
These are empty spaces in the page. Four more rows can be added to this page.

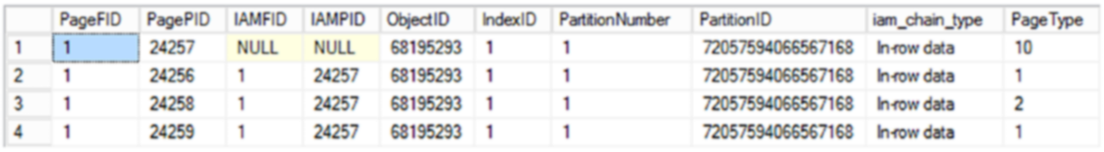
Figure 14-6. *Page split caused by an UPDATE statement*

From the preceding tables, you can see that the page split caused by the UPDATE statement results in an internal fragmentation of data in the leaf pages. If the new leaf page can't be written physically next to the original leaf page, there will be external fragmentation as well. For a large table with a high amount of fragmentation, a larger number of leaf pages will be required to hold all the index rows.

Another way to look at the distribution of pages is to use some less thoroughly documented DBCC commands. First up, you can look at the pages in the table using DBCC IND.

```
DBCC IND(AdventureWorks2017, 'dbo.Test1', -1);
```

This command lists the pages that make up a table. You get output like in Figure 14-7.



	PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType
1	1	24257	NULL	NULL	68195293	1	1	72057594066567168	In-row data	10
2	1	24256	1	24257	68195293	1	1	72057594066567168	In-row data	1
3	1	24258	1	24257	68195293	1	1	72057594066567168	In-row data	2
4	1	24259	1	24257	68195293	1	1	72057594066567168	In-row data	1

Figure 14-7. *Output from DBCC IND showing two pages*

If you focus on the `PageType`, you can see that there are now two pages of `PageType = 1`, which is a data page. There are other columns in the output that also show how the pages are linked together.

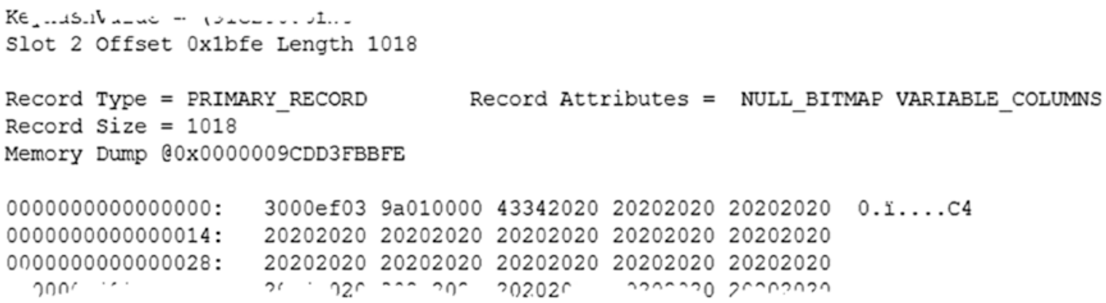
To see the resultant distribution of rows shown in the previous pages, you can add a trailing row to each page.

```
INSERT INTO dbo.Test1
VALUES (410, 'C4', "),
       (900, 'C4', ");
```

These new rows are accommodated in the existing two leaf pages without causing a page split. You can confirm this by querying the other mechanism for looking at page information, `DBCC PAGE`. To call this, you'll need to get the `PagePID` from the output of `DBCC IND`. This will enable you to pull back a full dump of everything on a page.

```
DBCC TRACEON(3604);
DBCC PAGE('AdventureWorks2017',1,24256,3);
```

The output from this is involved to interpret, but if you scroll down to the bottom, you can see the output, as shown in Figure 14-8.



```
[illegible]
Slot 2 Offset 0x1bfe Length 1018

Record Type = PRIMARY_RECORD        Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 1018
Memory Dump @0x0000009CDD3FBBFE

0000000000000000:   3000ef03 9a010000 43342020 20202020 20202020  0.ï....C4
0000000000000014:   20202020 20202020 20202020 20202020 20202020
0000000000000028:   20202020 20202020 20202020 20202020 20202020
[illegible]
```

Figure 14-8. *Pages after adding more rows*

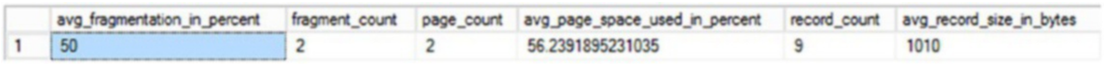
On the right side of the screen, you can see the output from the memory dump, a value, C4. That was added by the previous data. Both rows were added to one page in my tests. Getting into a full explanation of all possible permutations of these two DBCC calls is far beyond the scope of this chapter. Know that you can determine which page data is stored on for any given table.

Page Split by an INSERT Statement

To understand how a page split can be caused by an INSERT statement, create the same test table as you did previously, with the eight initial rows and the clustered index. Since the single index leaf page is completely filled, any attempt to add an intermediate row as follows should cause a page split in the leaf page:

```
INSERT INTO Test1
VALUES (110, 'C2', '');
```

You can verify this by examining the output of sys.dm_db_index_physical_stats (Figure 14-9).



	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	50	2	2	56.2391895231035	9	1010

Figure 14-9. *Pages after insert*

As explained previously, half the rows from the original leaf page are moved to the new page. Once space is cleared in the original leaf page, the new row is added in the appropriate order to the original leaf page. Be aware that a row is associated with only one page; it cannot span multiple pages. Figure 14-10 shows the resultant distribution of rows in the two pages.

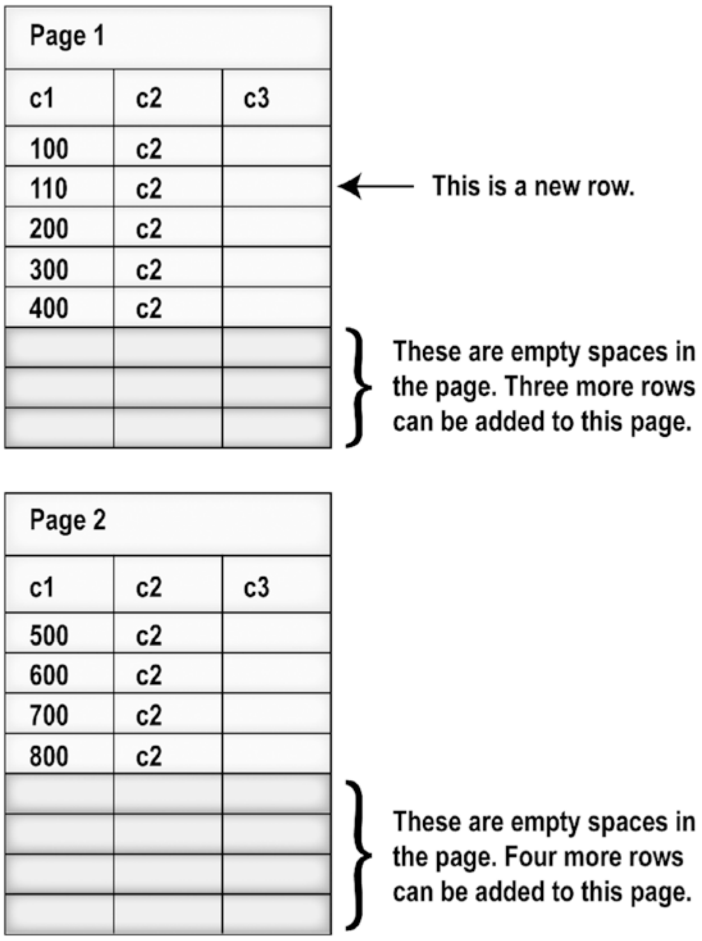

Page 1
c1
c2
c3
100
c2
110
c2
This is a new row.
200
c2
300
c2
400
c2
These are empty spaces in the page. Three more rows can be added to this page.
Page 2
c1
c2
c3
500
c2
600
c2
700
c2
800
c2
These are empty spaces in the page. Four more rows can be added to this page.

Figure 14-10. *Page split caused by an INSERT statement*

From the previous index pages, you can see that the page split caused by the `INSERT` statement spreads the rows sparsely across the leaf pages, causing internal fragmentation. It often causes external fragmentation also, since the new leaf page may not be physically adjacent to the original page. For a large table with a high amount of fragmentation, the page splits caused by the `INSERT` statement will require a larger number of leaf pages to accommodate all the index rows.

To demonstrate the row distribution shown in the index pages, you can run the script to create `dbo.Test1` again, adding more rows to the pages.

```
INSERT  INTO dbo.Test1
VALUES  (410, 'C4', ''),
        (900, 'C4', '');
```

The result is the same as for the previous example: these new rows can be accommodated in the two existing leaf pages without causing any page split. You can validate that by calling `DBCC IND` and `DBCC PAGE`. Note that in the first page, new rows are added in between the other rows in the page. This won't cause a page split since free space is available in the page.

What about when you have to add rows to the trailing end of an index? In this case, even if a new page is required, it won't split any existing page. For example, adding a new row with `C1` equal to 1,300 will require a new page, but it won't cause a page split since the row isn't added in an intermediate position. Therefore, if new rows are added in the order of the clustered index, then the index rows will be always added at the trailing end of the index, preventing the page splits otherwise caused by the `INSERT` statements. However, you'll also get what is called a *hot page* in this scenario. A hot page is when all the inserts are trying to write to a single page in the database leading to blocking. Depending on your system and the load on it, this can be much more problematic than page splits, so be sure to monitor your wait statistics to know how your system is behaving.

Data Modification and the Columnstore Indexes

Like the rowstore indexes, columnstore indexes can also suffer from fragmentation. When a columnstore index is first loaded, assuming at least 102,400 rows, the data is stored into the compressed column segments that make up a columnstore index. Anything less than 102,400 rows is stored in the deltastore, which if you remember from Chapter 9, is just a regular B-tree index. The data stored in the compressed column segments is not fragmented. To avoid fragmentation over time, where possible, all the changes are stored in the deltastore precisely to avoid fragmenting the compressed column segments. All changes, updates, and deletes, until an index is reorganized or rebuilt, are stored in the deltastore as logical changes. By logical changes I mean that for a delete, the data is marked as deleted, but it is not removed. For an update, the old values are marked as deleted and new values are added. While a columnstore doesn't fragment in the same way, as a page split, these logical deletes represent fragmentation of the columnstore index. The more of them there are, the more logically fragmented that index. Eventually, you'll need to fix it.

To see the fragmentation in action, I'm going to use the large columnstore tables I created in Chapter 9. Here I'm going to modify one of the tables to make it into a clustered columnstore index:

```
ALTER TABLE dbo.bigTransactionHistory
DROP CONSTRAINT pk_bigTransactionHistory

CREATE CLUSTERED COLUMNSTORE INDEX cci_bigTransactionHistory
ON dbo.bigTransactionHistory;
```

To see the logical fragmentation within a clustered columnstore index, we're going to look at the system view `sys.column_store_row_groups` in a query like this:

```
SELECT OBJECT_NAME(i.object_id) AS TableName,
       i.name AS IndexName,
       i.type_desc,
       csrg.partition_number,
       csrg.row_group_id,
       csrg.delta_store_hobt_id,
       csrg.state_description,
       csrg.total_rows,
       csrg.deleted_rows,
       100 * (total_rows - ISNULL(deleted_rows,
                                  0)) / total_rows AS PercentFull
FROM sys.indexes AS i
    JOIN sys.column_store_row_groups AS csrg
        ON i.object_id = csrg.object_id
           AND i.index_id = csrg.index_id
WHERE name = 'cci_bigTransactionHistory'
ORDER BY OBJECT_NAME(i.object_id),
         i.name,
         row_group_id;
```

With a new index on `dbo.bigTransactionHistory`, we can anticipate no logical fragmentation caused by deleted rows. You can see this if you run the previous query. It will show 31 row groups with zero rows deleted on any of them. You will see some rowgroups that have less than the max value. It's not a problem; it's just an artifact of the data load. Let's delete a few rows.