Auto Update Statistics Asynchronously

If auto update statistics asynchronously is set to on, the basic behavior of statistics in SQL Server isn't changed radically. When a set of statistics is marked as out-of-date and a query is then run against those statistics, the statistics update does not interrupt the execution of the query, like normally happens. Instead, the query finishes execution using the older set of statistics. Once the query completes, the statistics are updated. The reason this may be attractive is that when statistics are updated, query plans in the procedure cache are removed, and the query being run must be recompiled. This causes a delay in the execution of the query. So, rather than make a query wait for both the update of the statistics and a recompile of the procedure, the query completes its run. The next time the same query is called, it will have updated statistics waiting for it, and it will have to recompile only.

Although this functionality does make the steps needed to update statistics and recompile the procedure somewhat faster, it can also cause queries that could benefit immediately from updated statistics and a new execution plan to work with the old execution plan. Careful testing is required before turning this functionality on to ensure it doesn't cause more harm than good.

Note If you are attempting to update statistics asynchronously, you must also have `AUTO_UPDATE_STATISTICS` set to `ON`.

Manual Maintenance

The following are situations in which you need to interfere with or assist the automatic maintenance of statistics:

- *When experimenting with statistics*: Just a friendly suggestion—please spare your production servers from experiments such as the ones you are doing in this book.
- *After upgrading from a previous version to SQL Server 2017*: In earlier versions of this book I suggested updating statistics immediately on an upgrade to a new version of SQL Server. This was because of the changes in statistics introduced in SQL Server 2014. It made sense to immediately update the statistics so that you were seeing the effects

of the new cardinality estimator. With the addition of the Query Store, I can no longer make this recommendation in the same way. Instead, I'll suggest that you consider it if upgrading from SQL Server 2014 to a newer version, but even then, I wouldn't suggest manually updating the statistics by default. I would test it first to understand how a given upgrade will behave.

- *While executing a series of ad hoc SQL activities that you won't execute again*: In such cases, you must decide whether you want to pay the cost of automatic statistics maintenance to get a better plan for that one case while affecting the performance of other SQL Server activities. So, in general, you might not need to be concerned with such singular events. This is mainly applicable to larger databases, but you can test it in your environment if you think it may apply.
- *When you come upon an issue with the automatic statistics maintenance and the only workaround for the time being is to keep the automatic statistics maintenance feature off*: Even in these cases, you can turn the feature off for the specific table that faces the problem instead of disabling it for the complete database. Issues like this can be found in large data sets where the data is updated a lot but not enough to trigger the threshold update. Also, it can be used in cases where the sampling level of the automatic updates is not adequate for some data distributions.
- *While analyzing the performance of a query, you realize that the statistics are missing for a few of the database objects referred to by the query*: This can be evaluated from the graphical and XML execution plans, as explained earlier in the chapter.
- *While analyzing the effectiveness of statistics, you realize that they are inaccurate*: This can be determined when poor execution plans are being created from what should be good sets of statistics.

SQL Server allows a user to control many of its automatic statistics maintenance features. You can enable (or disable) the automatic statistics creation and update features by using the auto create statistics and auto update statistics settings, respectively, and then you can get your hands dirty.

Manage Statistics Settings

You can control the auto create statistics setting at a database level. To disable this setting, use the `ALTER DATABASE` command.

```
ALTER DATABASE AdventureWorks2017 SET AUTO_CREATE_STATISTICS OFF;
```

You can control the auto update statistics setting at different levels of a database, including all indexes and statistics on a table, or at the individual index or statistics level. To disable auto update statistics at the database level, use the `ALTER DATABASE` command.

```
ALTER DATABASE AdventureWorks2017 SET AUTO_UPDATE_STATISTICS OFF;
```

Disabling this setting at the database level overrides individual settings at lower levels. Auto update statistics asynchronously requires that the auto update statistics be on first. Then you can enable the asynchronous update.

```
ALTER DATABASE AdventureWorks2017 SET AUTO_UPDATE_STATISTICS_ASYNC ON;
```

To configure auto update statistics for all indexes and statistics on a table in the current database, use the `sp_autostats` system stored procedure.

```
USE AdventureWorks2017;
EXEC sp_autostats
    'HumanResources.Department',
    'OFF';
```

You can also use the same stored procedure to configure this setting for individual indexes or statistics. To disable this setting for the `AK_Department_Name` index on `AdventureWorks2017.HumanResources.Department`, execute the following statements:

```
EXEC sp_autostats
    'HumanResources.Department',
    'OFF',
    AK_Department_Name;
```

You can also use the `UPDATE STATISTICS` command's `WITH NORECOMPUTE` option to disable this setting for all or individual indexes and statistics on a table in the current database. The `sp_createstats` stored procedure also has the `NORECOMPUTE` option. The `NORECOMPUTE` option will not disable automatic update of statistics for the database, but it will for a given set of statistics.

Avoid disabling the automatic statistics features, unless you have confirmed through testing that this brings a performance benefit. If the automatic statistics features are disabled, then you are responsible for manually identifying and creating missing statistics on the columns that are not indexed and then keeping the existing statistics up-to-date. In general, you're only going to want to disable the automatic statistics features for very large tables and only after you've carefully measured the blocking and locking so that you know that changing statistics behavior will help.

If you want to check the status of whether a table has its automatic statistics turned off, you can use this:

```
EXEC sp_autostats 'HumanResources.Department';
```

Reset the automatic maintenance of the index so that it is on where it has been turned off.

```
EXEC sp_autostats
    'HumanResources.Department',
    'ON';
EXEC sp_autostats
    'HumanResources.Department',
    'ON',
    AK_Department_Name;
```

Generate Statistics

To create statistics manually, use one of the following options:

- `CREATE STATISTICS`: You can use this option to create statistics on single or multiple columns of a table or an indexed view. Unlike the `CREATE INDEX` command, `CREATE STATISTICS` uses sampling by default.
- `sys.sp_createstats`: Use this stored procedure to create single-column statistics for all eligible columns for all user tables in the current database. This includes all columns except computed columns; columns with the `NTEXT`, `TEXT`, `GEOMETRY`, `GEOGRAPHY`, or `IMAGE` data type; sparse columns; and columns that already have statistics or are the first column of an index. This function is meant for backward compatibility, and I don't recommend using it.

While a statistics object is created for a columnstore index, the values inside that index are null. Individual columns on a columnstore index can have the regular system-generated statistics created against them. When dealing with a columnstore index, if you find that you're still referencing the individual columns, you may find, in some situations, that creating a multicolumn statistic is useful. An example would look like this:

```
CREATE STATISTICS MultiColumnExample
ON dbo.bigProduct (ProductNumber,
                   Name);
```

With the exception of the individual column statistics and any that you create, there is no need to worry about the automatically created index statistic on a columnstore index.

If you partition a columnstore index (partitioning is not a performance enhancement tool, it's a data management tool), you'll need to change your statistics to be incremental using the following command to ensure that statistics updates are only by partition:

```
UPDATE STATISTICS dbo.bigProduct WITH RESAMPLE, INCREMENTAL=ON;
```

To update statistics manually, use one of the following options:

- `UPDATE STATISTICS`: You can use this option to update the statistics of individual or all index keys and nonindexed columns of a table or an indexed view.
- `sys.sp_updatestats`: Use this stored procedure to update statistics of all user tables in the current database. However, note that it can only sample the statistics, not use `FULLSCAN`, and it will update statistics when only a single action has been performed on that statistics. In short, this is a rather blunt instrument for maintaining statistics.

You may find that allowing the automatic updating of statistics is not quite adequate for your system. Scheduling `UPDATE STATISTICS` for the database during off-hours is an acceptable way to deal with this issue. `UPDATE STATISTICS` is the preferred mechanism because it offers a greater degree of flexibility and control. It's possible, because of the types of data inserted, that the sampling method for gathering the statistics, used because it's faster, may not gather the appropriate data. In these cases, you can force a `FULLSCAN` so that all the data is used to update the statistics just like what happens when the statistics are initially created. This can be a costly operation, so it's best to be

selective about which indexes receive this treatment and when it is run. In addition, if you do set sampling rates for your statistics rebuilds, including FULLSCAN, you should use PERSIST_SAMPLE_PERCENT to ensure that any automated processes that fire will use the same sampling rate.

Note In general, you should always use the default settings for automatic statistics. Consider modifying these settings only after identifying that the default settings appear to detract from performance.

Statistics Maintenance Status

You can verify the current settings for the autostats feature using the following:

- sys.databases
- DATABASEPROPERTYEX
- sp_autostats

Status of Auto Create Statistics

You can verify the current setting for auto create statistics by running a query against the sys.databases system table.

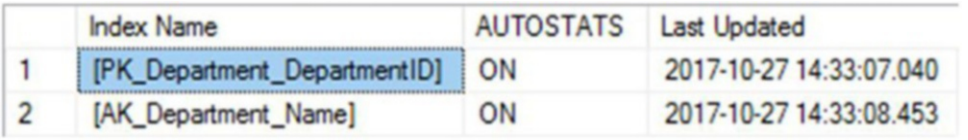
```
SELECT  is_auto_create_stats_on
FROM    sys.databases
WHERE   [name] = 'AdventureWorks2017';
```

A return value of 1 means enabled, and a value of 0 means disabled.

You can also verify the status of specific indexes using the sp_autostats system stored procedure, as shown in the following code. Supplying any table name to the stored procedure will provide the configuration value of auto create statistics for the current database under the Output section of the global statistics settings.

```
USE AdventureWorks2017;
EXEC sys.sp_autostats 'HumanResources.Department';
```

Figure 13-34 shows an excerpt of the preceding `sp_autostats` statement's output.



	Index Name	AUTOSTATS	Last Updated
1	[PK_Department_DepartmentID]	ON	2017-10-27 14:33:07.040
2	[AK_Department_Name]	ON	2017-10-27 14:33:08.453

Figure 13-34. *`sp_autostats` output*

A return value of `ON` means enabled, and a value of `OFF` means disabled. This stored procedure is more useful when verifying the status of auto update statistics, as explained earlier in this chapter.

You can also verify the current setting for auto update statistics, and auto update statistics asynchronously, in a similar manner to auto create statistics. Here's how to do it using the function `DATABASEPROPERTYEX`:

```
SELECT  DATABASEPROPERTYEX('AdventureWorks2017', 'IsAutoUpdateStatistics');
```

Here's how to do it using `sp_autostats`:

```
USE AdventureWorks2017;
EXEC sp_autostats
    'Sales.SalesOrderDetail';
```

Analyzing the Effectiveness of Statistics for a Query

For performance reasons, it is extremely important to maintain proper statistics on your database objects. Issues with statistics can be fairly common. You need to keep your eyes open to the possibility of problems with statistics while analyzing the performance of a query. If an issue with statistics does arise, then it can really take you for a ride. In fact, checking that the statistics are up-to-date at the beginning of a query-tuning session eliminates an easily fixed problem. In this section, you'll see what you can do should you find statistics to be missing or out-of-date.

While analyzing an execution plan for a query, look for the following points to ensure a cost-effective processing strategy:

- Indexes are available on the columns referred to in the filter and join criteria.
- In the case of a missing index, statistics should be available on the columns with no index. It may be preferable to have the index itself.
- Since outdated statistics are of no use and can even be misleading, it is important that the estimates used by the optimizer from the statistics are up-to-date.

You analyzed the use of a proper index in Chapter 9. In this section, you will analyze the effectiveness of statistics for a query.

Resolving a Missing Statistics Issue

To see how to identify and resolve a missing statistics issue, consider the following example. To more directly control the data, I'll use a test table instead of one of the AdventureWorks2017 tables. First disable both auto create statistics and auto update statistics using the `ALTER DATABASE` command.

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

Create a test table with a large number of rows and a nonclustered index.

```
IF EXISTS (   SELECT *
              FROM sys.objects
              WHERE object_id = OBJECT_ID(N'dbo.Test1'))
    DROP TABLE dbo.Test1;
GO

CREATE TABLE dbo.Test1 (C1 INT,
                        C2 INT,
                        C3 CHAR(50));
INSERT INTO dbo.Test1 (C1,
                       C2,
                       C3)
```

```
VALUES (51, 1, 'C3'),
       (52, 1, 'C3');

CREATE NONCLUSTERED INDEX iFirstIndex ON dbo.Test1 (C1, C2);

SELECT TOP 10000
       IDENTITY(INT, 1, 1) AS n
INTO #Nums
FROM master.dbo.syscolumns AS scl,
     master.dbo.syscolumns AS sC2;

INSERT INTO dbo.Test1 (C1,
                       C2,
                       C3)
SELECT n % 50,
       n,
       'C3'
FROM #Nums;
DROP TABLE #Nums;
```

Since the index is created on (C1, C2), the statistics on the index contain a histogram for the first column, C1, and density values for the prefixed column combinations (C1 and C1 * C2). There are no histograms or density values alone for column C2.

To understand how to identify missing statistics on a column with no index, execute the following SELECT statement. Since the auto create statistics feature is off, the optimizer won't be able to find the data distribution for the column C2 used in the WHERE clause. Before executing the query, ensure you have enabled Include Actual Execution Plan by clicking the query toolbar or hitting Ctrl+M.

```
SELECT  *
FROM    dbo.Test1
WHERE   C2 = 1;
```

The information on missing statistics is also provided by the graphical execution plan, as shown in Figure 13-35.

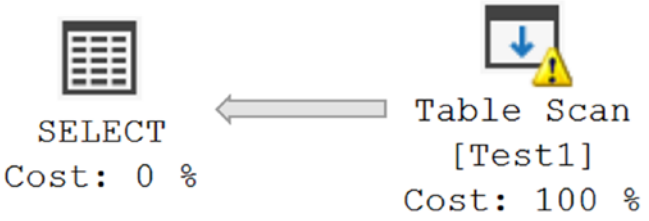


Figure 13-35. *Missing statistics indication in a graphical plan*

The graphical execution plan shows an operator with the yellow exclamation point. This indicates some problem with the operator in question. You can obtain a detailed description of the warning by right-clicking the Table Scan operator and then selecting Properties from the context menu. There's a warning section in the properties page that you can drill into, as shown in Figure 13-36.

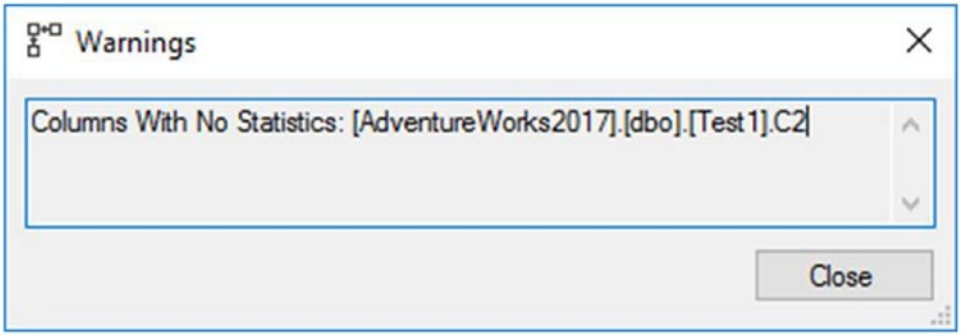


Figure 13-36. *Property values from the warning in the Index Scan operator*

Figure 13-36 shows that the statistics for the column are missing. This may prevent the optimizer from selecting the best processing strategy. The current cost of this query, as recorded by Extended Events is 100 reads and 850mc on average.

To resolve this missing statistics issue, you can create the statistics on column `Test1.C2` by using the `CREATE STATISTICS` statement.

```
CREATE STATISTICS Stats1 ON Test1(C2);
```

Before rerunning the query, be sure to clean out the procedure cache because this query will benefit from simple parameterization.

```
DECLARE @Planhandle VARBINARY(64);

SELECT @Planhandle = deqs.plan_handle
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
WHERE dest.text LIKE '%SELECT  *
FROM    dbo.Test1
WHERE   C2 = 1;%'

IF @Planhandle IS NOT NULL
BEGIN
    DBCC FREEPROCCACHE(@Planhandle);
END
GO
```

Caution When running the previous query on a production system, using the `LIKE '%...%'` wildcards can be inefficient. Looking for a specific string can be a more accurate way to remove a single query from the plan cache.

Figure 13-37 shows the resultant execution plan with statistics created on column `C2`.

```
Reads: 34
Duration: 4.3 ms.
```

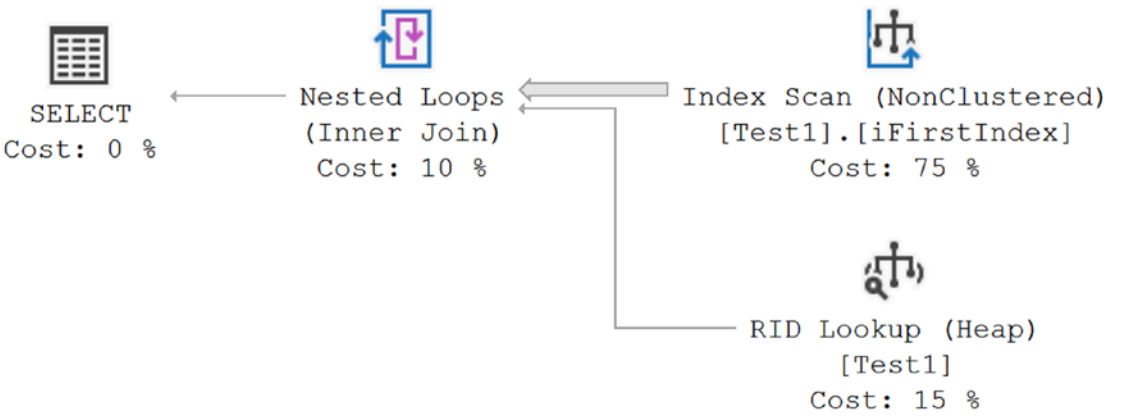


Figure 13-37. *Execution plan with statistics in place*

The query optimizer uses statistics on a noninitial column in a composite index to determine whether scanning the leaf level of the composite index to obtain the RID lookup information will be a more efficient processing strategy than scanning the whole table. In this case, creating statistics on column `C2` allows the optimizer to determine that instead of scanning the base table, it will be less costly to scan the composite index on (`C1`, `C2`) and bookmark lookup to the base table for the few matching rows. Consequently, the number of logical reads has decreased from 100 to 34, but the elapsed time has increased significantly because of the extra processing needed to join the data from two different operators.

Resolving an Outdated Statistics Issue

Sometimes outdated or incorrect statistics can be more damaging than missing statistics. Based on old statistics or a partial scan of changed data, the optimizer may decide upon a particular indexing strategy, which may be highly inappropriate for the current data distribution. Unfortunately, the execution plans don't show the same glaring warnings for outdated or incorrect statistics as they do for missing statistics. However, there is an extended event called `inaccurate_cardinality_estimate`. This is a debug event, which means its use could be somewhat problematic on a production system. I strongly caution you in its use, only when properly filtered and only for short periods of time, but I want to point it out. Instead, take advantage of Showplan Analysis detailed in Chapter 7.

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
```

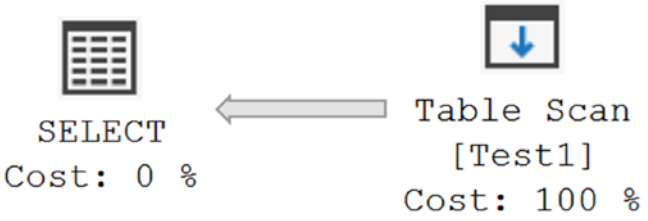


Figure 13-39. *Execution plan with outdated statistics*

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.

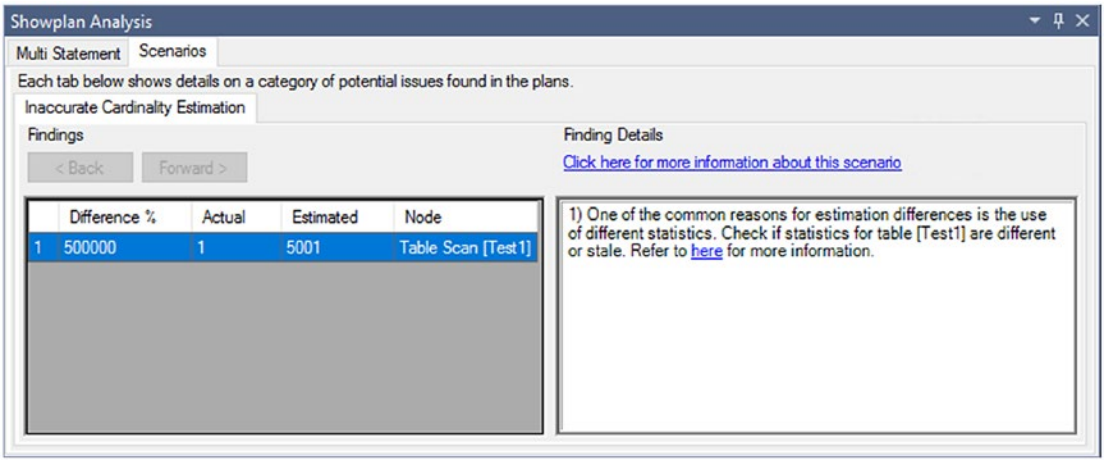


Figure 13-41. *Inaccurate Cardinality Estimation report showing the difference between actual and estimated*

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
```

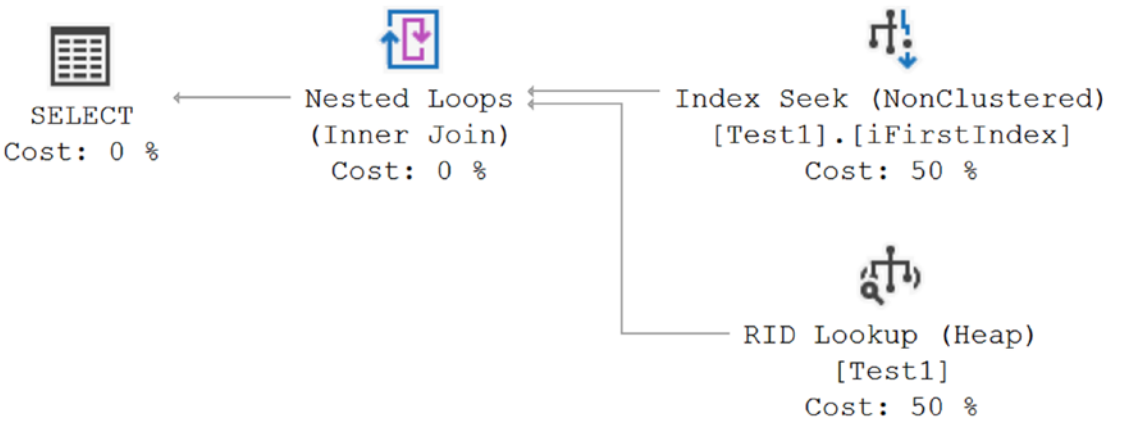


Figure 13-42. *Actual and estimated number of rows with up-to-date statistics*

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

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.

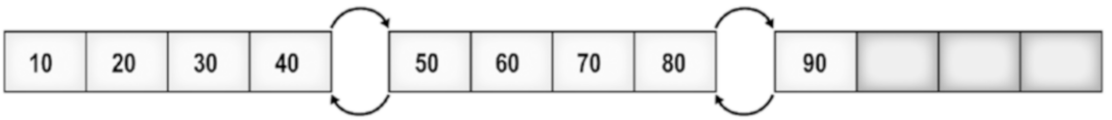


Figure 14-1. *Leaf pages layout*

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.

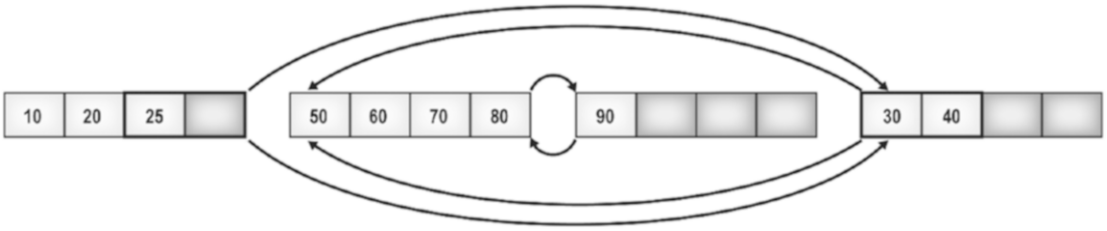


Figure 14-2. *Out-of-order leaf pages*

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.

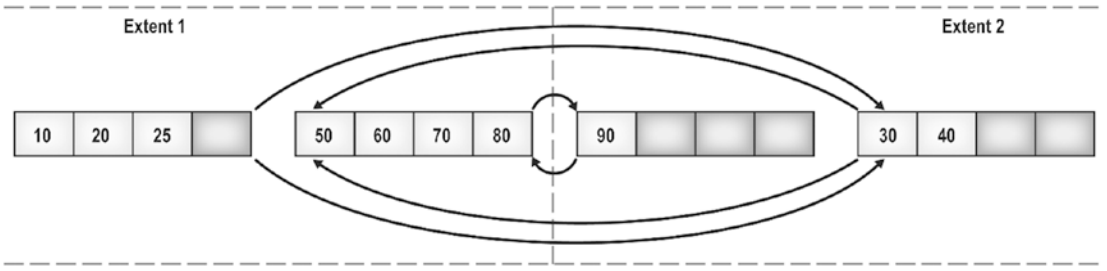


Figure 14-3. *Out-of-order leaf pages distributed across extents*

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.

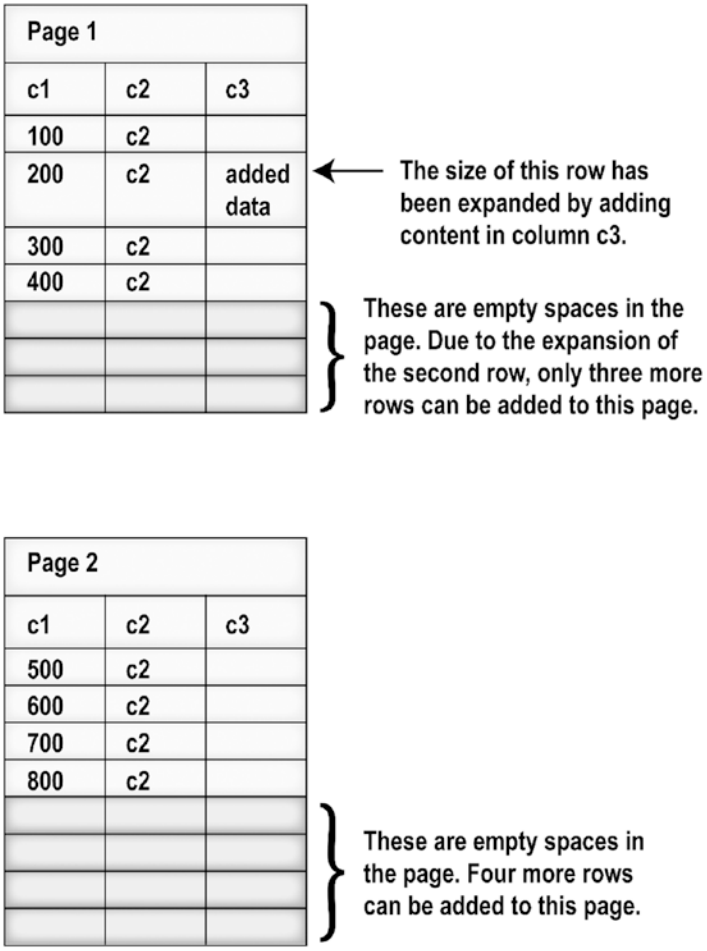


Figure 14-6. *Page split caused by an UPDATE statement*

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.

```
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
```

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.

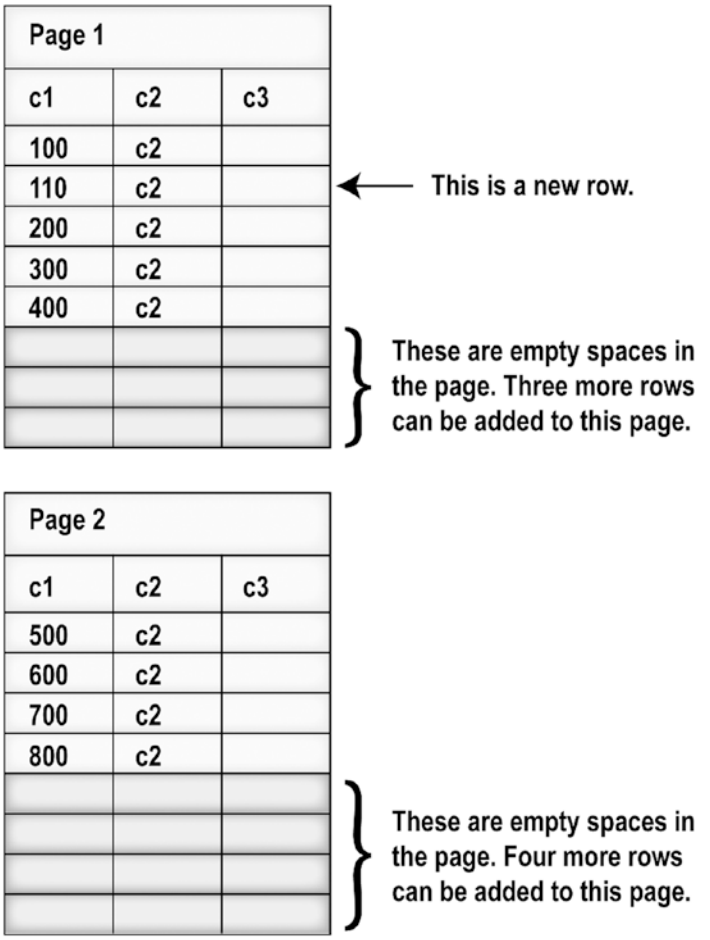


Figure 14-10. *Page split caused by an INSERT statement*

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.

```
DELETE dbo.bigTransactionHistory
WHERE Quantity = 13;
```

Now when we run the previous query, we can see the logical fragmentation of the columnstore index, as shown in Figure 14-11.

	TableName	IndexName	partition_number	row_group_id	state_description	total_rows	deleted_rows	PercentFull
1	bigTransactionHistory	cci_bigTransactionHistory	1	0	COMPRESSED	1048576	10544	98
2	bigTransactionHistory	cci_bigTransactionHistory	1	1	COMPRESSED	1048576	10467	99
3	bigTransactionHistory	cci_bigTransactionHistory	1	2	COMPRESSED	1048576	10388	99
4	bigTransactionHistory	cci_bigTransactionHistory	1	3	COMPRESSED	1048576	10468	99
5	bigTransactionHistory	cci_bigTransactionHistory	1	4	COMPRESSED	1048576	10273	99
6	bigTransactionHistory	cci_bigTransactionHistory	1	5	COMPRESSED	1048576	10541	98
7	bigTransactionHistory	cci_bigTransactionHistory	1	6	COMPRESSED	1048576	10361	99
8	bigTransactionHistory	cci_bigTransactionHistory	1	7	COMPRESSED	1048576	10606	98
9	bigTransactionHistory	cci_bigTransactionHistory	1	8	COMPRESSED	1048576	10590	98
10	bigTransactionHistory	cci_bigTransactionHistory	1	9	COMPRESSED	1048576	10530	98
11	bigTransactionHistory	cci_bigTransactionHistory	1	10	COMPRESSED	1048576	10429	99
12	bigTransactionHistory	cci_bigTransactionHistory	1	11	COMPRESSED	1048576	10578	98
13	bigTransactionHistory	cci_bigTransactionHistory	1	12	COMPRESSED	1048576	10568	98
14	bigTransactionHistory	cci_bigTransactionHistory	1	13	COMPRESSED	1048576	10428	99
15	bigTransactionHistory	cci_bigTransactionHistory	1	14	COMPRESSED	1048576	10472	99
16	bigTransactionHistory	cci_bigTransactionHistory	1	15	COMPRESSED	1048576	10400	99
17	bigTransactionHistory	cci_bigTransactionHistory	1	16	COMPRESSED	1048576	10608	98
18	bigTransactionHistory	cci_bigTransactionHistory	1	17	COMPRESSED	1048576	10369	99
19	bigTransactionHistory	cci_bigTransactionHistory	1	18	COMPRESSED	1048576	10355	99
20	bigTransactionHistory	cci_bigTransactionHistory	1	19	COMPRESSED	1048576	10472	99
21	bigTransactionHistory	cci_bigTransactionHistory	1	20	COMPRESSED	1048576	10423	99
22	bigTransactionHistory	cci_bigTransactionHistory	1	21	COMPRESSED	1048576	10489	98
23	bigTransactionHistory	cci_bigTransactionHistory	1	22	COMPRESSED	1048576	10683	98
24	bigTransactionHistory	cci_bigTransactionHistory	1	23	COMPRESSED	1048576	10503	98
25	bigTransactionHistory	cci_bigTransactionHistory	1	24	COMPRESSED	1048576	10580	98
26	bigTransactionHistory	cci_bigTransactionHistory	1	25	COMPRESSED	1048576	10480	99
27	bigTransactionHistory	cci_bigTransactionHistory	1	26	COMPRESSED	1048576	10372	99
28	bigTransactionHistory	cci_bigTransactionHistory	1	27	COMPRESSED	1048576	10499	98
29	bigTransactionHistory	cci_bigTransactionHistory	1	28	COMPRESSED	1048576	10459	99
30	bigTransactionHistory	cci_bigTransactionHistory	1	29	COMPRESSED	104086	1045	98
31	bigTransactionHistory	cci_bigTransactionHistory	1	30	COMPRESSED	750811	7466	99

Figure 14-11. *Fragmentation of a clustered columnstore index*

You can see that all the rowgroups were affected by the DELETE operation and are now fragmented between 98 percent and 99 percent.

Fragmentation Overhead

Fragmentation overhead primarily consists of the additional overhead caused by reading more pages from disk. Reading more pages from disk means reading more pages into memory. Both of these cause strain on the system because you're using more and more resources to deal with the fragmented storeage of the index. As I stated earlier in the opening Discussion on Fragmentation, for some systems, this may not be an issue. However, for some systems it is. We'll discuss the details of exactly where the load comes from in both rowstore and columnstore indexes in some detail.

Rowstore Overhead

Both internal and external fragmentation adversely affect data retrieval performance. External fragmentation causes a noncontiguous sequence of index pages on the disk, with new leaf pages far from the original leaf pages and with their physical ordering different from their logical ordering. Consequently, a range scan on an index will need more switches between the corresponding extents than ideally required, as explained earlier in the chapter. Also, a range scan on an index will be unable to benefit from read-ahead operations performed on the disk. If the pages are arranged contiguously, then a read-ahead operation can read pages in advance without much head movement.

For better performance, it is preferable to use sequential I/O, since this can read a whole extent (eight 8KB pages together) in a single disk I/O operation. By contrast, a noncontiguous layout of pages requires nonsequential or random I/O operations to retrieve the index pages from the disk, and a random I/O operation can read only 8KB of data in a single disk operation (this may be acceptable, however, if you are retrieving only one row). The increasing speed of hard drives, especially SSDs, has reduced the impact of this issue, but it's still there in some situations.

In the case of internal fragmentation, rows are distributed sparsely across a large number of pages, increasing the number of disk I/O operations required to read the index pages into memory and increasing the number of logical reads required to retrieve multiple index rows from memory. As mentioned earlier, even though it increases the cost of data retrieval, a little internal fragmentation can be beneficial because it allows you to perform `INSERT` and `UPDATE` queries without causing page splits. For queries that don't have to traverse a series of pages to retrieve the data, fragmentation can have minimal impact. Put another way, retrieving a single value from the index won't be impacted by the fragmentation; or, at most, it might have an additional level in the B-tree that it has to travel down.

To understand how fragmentation affects the performance of a query, create a test table with a clustered index and insert a highly fragmented data set in the table. Since an INSERT operation in between an ordered data set can cause a page split, you can easily create the fragmented data set by adding rows in the following order:

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT,
                        C2 INT,
                        C3 INT,
                        c4 CHAR(2000));

CREATE CLUSTERED INDEX i1 ON dbo.Test1 (C1);


WITH Nums
AS (SELECT TOP (10000)
           ROW_NUMBER() OVER (ORDER BY (SELECT 1)) AS n
    FROM master.sys.all_columns AS ac1
        CROSS JOIN master.sys.all_columns AS ac2)
INSERT INTO dbo.Test1 (C1,
                       C2,
                       C3,
                       c4)
SELECT n,
       n,
       n,
       'a'
FROM Nums;

WITH Nums
AS (SELECT 1 AS n
    UNION ALL
    SELECT n + 1
    FROM Nums
    WHERE n < 10000)
```

```
INSERT INTO dbo.Test1 (C1,
                       C2,
                       C3,
                       c4)
SELECT 10000 - n,
       n,
       n,
       'a'
FROM Nums
OPTION (MAXRECURSION 10000);
```

To determine the number of logical reads required to retrieve a small result set and a large result set from this fragmented table, execute the following two SELECT statements with an Extended Events session (in this case, sql_batch_completed is all that's needed), monitoring query performance:

```
--Reads 6 rows
SELECT *
FROM dbo.Test1
WHERE C1 BETWEEN 21
         AND     23;

--Reads all rows
SELECT *
FROM dbo.Test1
WHERE C1 BETWEEN 1
         AND     10000;
```

The number of logical reads performed by the individual queries is, respectively, as follows:

```
6 rows
Reads:8
Duration:2.6ms
All rows
Reads:2542
Duration:475ms
```

To evaluate how the fragmented data set affects the number of logical reads, rearrange the index leaf pages physically by rebuilding the clustered index.

```
ALTER INDEX i1 ON dbo.Test1 REBUILD;
```

With the index leaf pages rearranged in the proper order, rerun the query. The number of logical reads required by the preceding two `SELECT` statements reduces to 5 and 13, respectively.

```
6 rows
Reads:6
Duration:1ms
All rows
Reads:2536
Duration:497ms
```

Performance improved for the smaller data set but didn't change much for the larger data set because just dropping a couple of pages isn't likely to have that big of an impact. The cost overhead because of fragmentation usually increases in line with the number of rows retrieved because this involves reading a greater number of out-of-order pages. For *point queries* (queries retrieving only one row), fragmentation doesn't usually matter since the row is retrieved from one leaf page only, but this isn't always the case. Because of the internal structure of the index, fragmentation may increase the cost of even a point query.

Note The lesson from this section is that, for better query performance, it is important to analyze the amount of fragmentation in an index and rearrange it if required.

Columnstore Overhead

While you are not dealing with pages rearranged on disk with the logical fragmentation of a columnstore index, you are still going to see a performance impact. The deleted values are stored in a B-tree index associated with the row group. Any data retrieval must go through an additional outer join against this data. You can't see this in the execution plan because it's an internal process. You can, however, see it in the performance of the queries against fragmented columnstore indexes.

To demonstrate this, we'll start with a query that takes advantage of the columnstore index, shown here:

```
SELECT bth.Quantity,
       AVG(bth.ActualCost)
FROM dbo.bigTransactionHistory AS bth
WHERE bth.Quantity BETWEEN 8
                   AND     15
GROUP BY bth.Quantity;
```

If you run this query, on average you get performance metrics as follows:

```
Reads:20932
Duration:70ms
```

If we were to fragment the index, specifically within the range of information upon which we're querying, as follows:

```
DELETE dbo.bigTransactionHistory
WHERE Quantity BETWEEN 9
               AND     12;
```

then the performance metrics change, as shown here:

```
Reads:20390
Duration:79ms
```

Note that the reads have dropped since a smaller amount of data overall will be processed to arrive at the results. However, performance has degraded from 70ms to 79ms. This is because of the fragmentation of the index, which we can see has become worse in Figure 14-12.

	TableName	IndexName	partition_number	row_group_id	state_description	total_rows	deleted_rows	PercentFull
1	bigTransactionHistory	cci_bigTransactionHistory	1	0	COMPRESSED	1048576	52759	94
2	bigTransactionHistory	cci_bigTransactionHistory	1	1	COMPRESSED	1048576	52497	94
3	bigTransactionHistory	cci_bigTransactionHistory	1	2	COMPRESSED	1048576	52191	95
4	bigTransactionHistory	cci_bigTransactionHistory	1	3	COMPRESSED	1048576	52482	94
5	bigTransactionHistory	cci_bigTransactionHistory	1	4	COMPRESSED	1048576	52294	95
6	bigTransactionHistory	cci_bigTransactionHistory	1	5	COMPRESSED	1048576	52819	94
7	bigTransactionHistory	cci_bigTransactionHistory	1	6	COMPRESSED	1048576	52647	94
8	bigTransactionHistory	cci_bigTransactionHistory	1	7	COMPRESSED	1048576	52477	94
9	bigTransactionHistory	cci_bigTransactionHistory	1	8	COMPRESSED	1048576	52472	94
10	bigTransactionHistory	cci_bigTransactionHistory	1	9	COMPRESSED	1048576	52370	95
11	bigTransactionHistory	cci_bigTransactionHistory	1	10	COMPRESSED	1048576	52472	94
12	bigTransactionHistory	cci_bigTransactionHistory	1	11	COMPRESSED	1048576	52773	94
13	bigTransactionHistory	cci_bigTransactionHistory	1	12	COMPRESSED	1048576	52639	94
14	bigTransactionHistory	cci_bigTransactionHistory	1	13	COMPRESSED	1048576	52323	95
15	bigTransactionHistory	cci_bigTransactionHistory	1	14	COMPRESSED	1048576	52360	95
16	bigTransactionHistory	cci_bigTransactionHistory	1	15	COMPRESSED	1048576	52276	95
17	bigTransactionHistory	cci_bigTransactionHistory	1	16	COMPRESSED	1048576	52741	94
18	bigTransactionHistory	cci_bigTransactionHistory	1	17	COMPRESSED	1048576	52329	95
19	bigTransactionHistory	cci_bigTransactionHistory	1	18	COMPRESSED	1048576	52474	94
20	bigTransactionHistory	cci_bigTransactionHistory	1	19	COMPRESSED	1048576	52647	94
21	bigTransactionHistory	cci_bigTransactionHistory	1	20	COMPRESSED	1048576	52046	95
22	bigTransactionHistory	cci_bigTransactionHistory	1	21	COMPRESSED	1048576	52673	94
23	bigTransactionHistory	cci_bigTransactionHistory	1	22	COMPRESSED	1048576	52668	94
24	bigTransactionHistory	cci_bigTransactionHistory	1	23	COMPRESSED	1048576	52885	94
25	bigTransactionHistory	cci_bigTransactionHistory	1	24	COMPRESSED	1048576	52453	94
26	bigTransactionHistory	cci_bigTransactionHistory	1	25	COMPRESSED	1048576	52414	95
27	bigTransactionHistory	cci_bigTransactionHistory	1	26	COMPRESSED	1048576	52114	95
28	bigTransactionHistory	cci_bigTransactionHistory	1	27	COMPRESSED	1048576	52500	94
29	bigTransactionHistory	cci_bigTransactionHistory	1	28	COMPRESSED	1048576	52434	94
30	bigTransactionHistory	cci_bigTransactionHistory	1	29	COMPRESSED	104086	5230	94
31	bigTransactionHistory	cci_bigTransactionHistory	1	30	COMPRESSED	750811	37419	95

Figure 14-12. *Increased fragmentation of the clustered columnstore index*

Analyzing the Amount of Fragmentation

You've already seen how to determine the fragmentation of a columnstore index. We can do the same with rowstore indexes. You can analyze the fragmentation ratio of an index by using the `sys.dm_db_index_physical_stats` dynamic management function. For a table with a clustered index, the fragmentation of the clustered index is congruous with the fragmentation of the data pages since the leaf pages of the clustered index and data pages are the same. `sys.dm_db_index_physical_stats` also indicates the amount of fragmentation in a heap table (or a table with no clustered index). Since a heap table doesn't require any row ordering, the logical order of the pages isn't relevant for the heap table.

The output of `sys.dm_db_index_physical_stats` shows information on the pages and extents of an index (or a table). A row is returned for each level of the B-tree in the index. A single row for each allocation unit in a heap is returned. As explained earlier, in SQL Server, eight contiguous 8KB pages are grouped together in an extent that is 64KB in size. For small tables (much less than 64KB), the pages in an extent can belong to more than one index or table—these are called *mixed extents.* If there are lots of small tables in the database, mixed extents help SQL Server conserve disk space.

As a table (or an index) grows and requests more than eight pages, SQL Server creates an extent dedicated to the table (or index) and assigns the pages from this extent. Such an extent is called a *uniform extent*, and it serves up to eight page requests for the same table (or index). Uniform extents help SQL Server lay out the pages of a table (or an index) contiguously. They also reduce the number of page creation requests by an eighth, since a set of eight pages is created in the form of an extent. Information stored in a uniform extent can still be fragmented, but accessing an allocation of pages is going to be much more efficient. If you have mixed extents, pages shared between multiple objects, and fragmentation within those extents, accessing the information becomes even more problematic. But there is no defragmenting done on mixed extents.

To analyze the fragmentation of an index, let's re-create the table with the fragmented data set used in the "Fragmentation Overhead" section. You can obtain the fragmentation detail of the clustered index (Figure 14-13) by executing the query against the `sys.dm_db_index_physical_stats` dynamic view used earlier.

```
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;
```

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	74.8174817481748	7543	9999	50.017358537188	20000	2022.999

Figure 14-13. *Fragmented statistics*

The dynamic management function `sys.dm_db_index_physical_stats` scans the pages of an index to return the data. You can control the level of the scan, which affects the speed and the accuracy of the scan. To quickly check the fragmentation of an index, use the Limited option. You can obtain an increased accuracy with only a moderate decrease in speed by using the Sample option, as in the previous example, which scans 1 percent of the pages. For the most accuracy, use the Detailed scan, which hits all the pages in an index. Just understand that the Detailed scan can have a major performance impact depending on the size of the table and index in question. If the index has fewer than 10,000 pages and you select the Sample mode, then the Detailed mode is used instead. This means that despite the choice made in the earlier query, the Detailed scan mode was used. The default mode is Limited.

By defining the different parameters, you can get fragmentation information on different sets of data. By removing the `OBJECT_ID` function in the earlier query and supplying a `NULL` value, the query would return information on all indexes within the database. Don't get surprised by this and accidentally run a Detailed scan on all indexes. You can also specify the index you want information on or even the partition with a partitioned index.

The output from `sys.dm_db_index_physical_stats` includes 24 different columns. I selected the basic set of columns used to determine the fragmentation and size of an index. This output represents the following:

- `avg_fragmentation_in_percent`: This number represents the logical average fragmentation for indexes and heaps as a percentage. If the table is a heap and the mode is Sampled, then this value will be `NULL`. If average fragmentation is less than 10 to 20 percent and the table isn't massive, fragmentation is unlikely to be an issue. If the index is between 20 and 40 percent, fragmentation might be an issue, but it can generally be helped by defragmenting the index through an index reorganization (more information on index reorganization and index rebuild is available in the "Fragmentation Resolutions" section). Large-scale fragmentation, usually greater than 40 percent, may require an index rebuild. Your system may have different requirements than these general numbers.

- `fragment_count`: This number represents the number of fragments, or separated groups of pages, that make up the index. It's a useful number to understand how the index is distributed, especially when compared to the `pagecount` value. `fragmentcount` is `NULL` when the sampling mode is Sampled. A large fragment count is an additional indication of storage fragmentation.
- `page_count`: This number is a literal count of the number of index or data pages that make up the statistic. This number is a measure of size but can also help indicate fragmentation. If you know the size of the data or index, you can calculate how many rows can fit on a page. If you then correlate this to the number of rows in the table, you should get a number close to the `pagecount` value. If the `pagecount` value is considerably higher, you may be looking at a fragmentation issue. Refer to the `avg_fragmentation_in_percent` value for a precise measure.
- `avg_page_space_used_in_percent`: To get an idea of the amount of space allocated within the pages of the index, use this number. This value is `NULL` when the sampling mode is Limited.
- `recordcount`: Simply put, this is the number of records represented by the statistics. For indexes, this is the number of records within the current level of the B-tree as represented from the scanning mode. (Detailed scans will show all levels of the B-tree, not simply the leaf level.) For heaps, this number represents the records present, but this number may not correlate precisely to the number of rows in the table since a heap may have two records after an update and a page split.
- `avg_record_size_in_bytes`: This number simply represents a useful measure for the amount of data stored within the index or heap record.

Running `sys.dm_db_index_physical_stats` with a Detailed scan will return multiple rows for a given index. That is, multiple rows are displayed if that index spans more than one level. Multiple levels exist in an index when that index spans more than a single page. To see what this looks like and to observe some of the other columns of data present in the dynamic management function, run the query this way:

```
SELECT ddips.*
FROM sys.dm_db_index_physical_stats(DB_ID('AdventureWorks2017'),
                                    OBJECT_ID(N'dbo.Test1'),
                                    NULL,
                                    NULL,
                                    'Detailed') AS ddips;
```

To make the data readable, I've broken down the resulting data table into three pieces in a single graphic; see Figure 14-14.

	database_id	object_id	index_id	partition_number	index_type_desc	alloc_unit_type_desc
1	6	1076198884	1	1	CLUSTERED INDEX	IN_ROW_DATA
2	6	1076198884	1	1	CLUSTERED INDEX	IN_ROW_DATA
3	6	1076198884	1	1	CLUSTERED INDEX	IN_ROW_DATA

index_depth	index_level	avg_fragmentation_in_percent	fragment_count	avg_fragment_size_in_pages	page_count
3	0	74.8174817481748	7543	1.3255998939414	9999
3	1	96.969696969697	25	1.32	33
3	2	0	1	1	1

avg_page_space_used_in_percent	record_count	ghost_record_count	version_ghost_record_count	min_record_size_in_bytes
50.017358537188	20000	0	0	2019
59.8703978255498	9999	0	0	11
6.46157647640227	33	0	0	11

max_record_size_in_bytes	avg_record_size_in_bytes	forwarded_record_count	compressed_page_count	hobt_id
2027	2022.999	NULL	0	72057594073907200
14	13.999	NULL	0	72057594073907200
14	13.909	NULL	0	72057594073907200

columnstore_delete_buffer_state	columnstore_delete_buffer_state_desc
0	NOT VALID
0	NOT VALID
0	NOT VALID

Figure 14-14. *Detailed scan of fragmented index*

As you can see, three rows were returned, representing the leaf level of the index (`index_ level = 0`) and representing the first level of the B-tree (`index_level = 1`), which is the second row and the third level of the B-tree (`index_level` = 2). You can see the additional information offered by `sys.dm_db_index_physical_stats` that can provide more detailed analysis of your indexes. For example, you can see the minimum and maximum record sizes, as well as the index depth (the number of levels in the B-tree) and how many records are on each level. A lot of this information will be less useful for basic fragmentation analysis, which is why I chose to limit the number of columns in the samples as well as use the Sampled scan mode. The columnstore information you see is primarily nonclustered columnstore internals. No information about clustered columnstore is returned by `sys.dm_db_index_physical_stats`. Instead, as shown earlier, you would use `sys.dm_db_column_store_row_group_physical_stats`.

Analyzing the Fragmentation of a Small Table

Don't be overly concerned with the output of `sys.dm_db_index_physical_stats` for small tables. For a small table or index with fewer than eight pages, SQL Server uses mixed extents for the pages. For example, if a table (`SmallTable1` or its clustered index) contains only two pages, then SQL Server allocates the two pages from a mixed extent instead of dedicating an extent to the table. The mixed extent may contain pages of other small tables/indexes also, as shown in Figure 14-15.

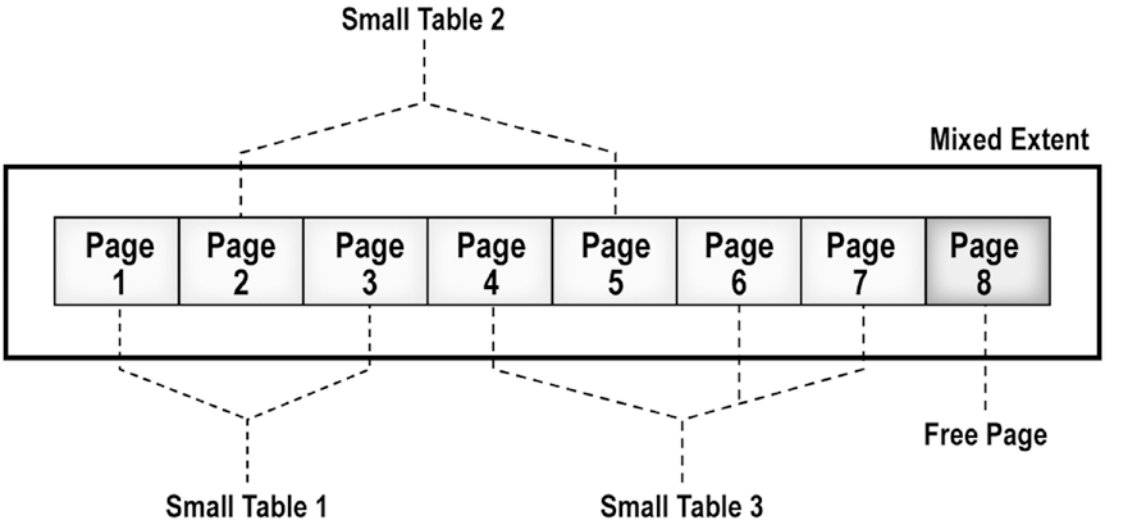


Figure 14-15. *Mixed extent*

The distribution of pages across multiple mixed extents may lead you to believe that there is a high amount of external fragmentation in the table or the index, when in fact this is by design in SQL Server and is therefore perfectly acceptable.

To understand how the fragmentation information of a small table or index may look, create a small table with a clustered index.

```
DROP TABLE IF EXISTS dbo.Test1;
GO

CREATE TABLE dbo.Test1 (C1 INT,
                        C2 INT,
                        C3 INT,
                        C4 CHAR(2000));

DECLARE @n INT = 1;

WHILE @n <= 28
BEGIN
    INSERT INTO dbo.Test1
    VALUES (@n, @n, @n, 'a');
    SET @n = @n + 1;
END

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

In the preceding table, with each INT taking 4 bytes, the average row size is 2,012 (=4 + 4 + 4 + 2,000) bytes. Therefore, a default 8KB page can contain up to four rows. After all 28 rows are added to the table, a clustered index is created to physically arrange the rows and reduce fragmentation to a minimum. With the minimum internal fragmentation, seven (= 28 / 4) pages are required for the clustered index (or the base table). Since the number of pages is not more than eight, SQL Server uses pages from mixed extents for the clustered index (or the base table). If the mixed extents used for the clustered index are not side by side, then the output of `sys.dm_db_index_physical_stats` may express a high amount of external fragmentation. But as a SQL user, you can't reduce the resultant external fragmentation. Figure 14-16 shows the output of `sys.dm_db_index_physical_stats`.

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	14.2857142857143	2	7	99.8517420311342	28	2019

Figure 14-16. *Fragmentation of a small clustered index*

From the output of `sys.dm_db_index_physical_stats`, you can analyze the fragmentation of the small clustered index (or the table) as follows:

- `avg_fragmentation_in_percent`: Although this index may cross to multiple extents, the fragmentation shown here is not an indication of external fragmentation because this index is being stored on mixed extents.
- `Avg_page_space_used_in_percent`: This shows that all or most of the data is stored well within the seven pages displayed in the `pagecount` field. This eliminates the possibility of logical fragmentation.
- `Fragment_count`: This shows that the data is fragmented and stored on more than one extent, but since it's less than eight pages long, SQL Server doesn't have much choice about where it stores the data.

In spite of the preceding misleading values, a small table (or index) with fewer than eight pages is simply unlikely to benefit from efforts to remove the fragmentation because it will be stored on mixed extents.

Once you determine that fragmentation in an index (or a table) needs to be dealt with, you need to decide which defragmentation technique to use. The factors affecting this decision, and the different techniques, are explained in the following section.

Fragmentation Resolutions

You can resolve fragmentation in an index by rearranging the index rows and pages so that their physical and logical orders match. To reduce external fragmentation, you can physically reorder the leaf pages of the index to follow the logical order of the index. On the columnstore index you're invoking the Tuple Mover, which will close the deltastores and put them into compressed segments, or you're doing that and forcing a reorganization of the data to achieve the best compression. You achieve all this through the following techniques:

- Dropping and re-creating the index
- Re-creating the index with the `DROP_EXISTING = ON` clause

- Executing the `ALTER INDEX REBUILD` statement on the index
- Executing the `ALTER INDEX REORGANIZE` statement on the index

Dropping and Re-creating the Index

One of the apparently easiest ways to remove fragmentation in an index is to drop the index and then re-create it. Dropping and re-creating the index reduces fragmentation the most since it allows SQL Server to use completely new pages for the index and populate them appropriately with the existing data. This avoids both internal and external fragmentation. Unfortunately, this method has a large number of serious shortcomings.

- *Blocking*: This technique of defragmentation adds a high amount of overhead on the system, and it causes blocking. Dropping and re-creating the index blocks all other requests on the table (or on any other index on the table). It can also be blocked by other requests against the table.
- *Missing index*: With the index dropped, and possibly being blocked and waiting to be re-created, queries against the table will not have the index available for use. This can lead to the poor performance that the index was intended to remedy.
- *Nonclustered indexes*: If the index being dropped is a clustered index, then all the nonclustered indexes on the table have to be rebuilt after the cluster is dropped. They then have to be rebuilt again after the cluster is re-created. This leads to further blocking and other problems such as statement recompiles (covered in detail in Chapter 19).
- *Unique constraints*: Indexes that are used to define a primary key or a unique constraint cannot be removed using the `DROP INDEX` statement. Also, both unique constraints and primary keys can be referred to by foreign key constraints. Prior to dropping the primary key, all foreign keys that reference the primary key would have to be removed first. Although this is possible, this is a risky and time-consuming method for defragmenting an index.

It is possible to use the ONLINE option for dropping a clustered index, which means the index is still readable while it is being dropped, but that saves you only from the previous blocking issue. For all these reasons, dropping and re-creating the index is not a recommended technique for a production database, especially at anything outside off-peak times.

Re-creating the Index with the DROP_EXISTING Clause

To avoid the overhead of rebuilding the nonclustered indexes twice while rebuilding a clustered index, use the DROP_EXISTING clause of the CREATE INDEX statement. This re-creates the clustered index in one atomic step, avoiding re-creating the nonclustered indexes since the clustered index key values used by the row locators remain the same. To rebuild a clustered key in one atomic step using the DROP_EXISTING clause, execute the CREATE INDEX statement as follows:

```
CREATE UNIQUE CLUSTERED INDEX FirstIndex
ON dbo.Test1
(
    C1
)
WITH (DROP_EXISTING = ON);
```

You can use the DROP_EXISTING clause for both clustered and nonclustered indexes and even to convert a nonclustered index to a clustered index. However, you can't use it to convert a clustered index to a nonclustered index.

The drawbacks of this defragmentation technique are as follows:

- *Blocking*: Similar to the DROP and CREATE methods, this technique also causes and faces blocking from other queries accessing the table (or any index on the table).
- *Index with constraints*: Unlike the first method, the CREATE INDEX statement with DROP_EXISTING can be used to re-create indexes with constraints. If the constraint is a primary key or the unique constraint is associated with a foreign key, then failing to include the UNIQUE keyword in the CREATE statement will result in an error like this:

Msg 1907, Level 16, State 1, Line 1 Cannot recreate index 'PK_Name'. The new index definition does not match the constraint being enforced by the existing index.

- *Table with multiple fragmented indexes*: As table data fragments, the indexes often become fragmented as well. If this defragmentation technique is used, then all the indexes on the table have to be identified and rebuilt individually.

You can avoid the last two limitations associated with this technique by using `ALTER INDEX REBUILD`, as explained next.

Executing the ALTER INDEX REBUILD Statement

`ALTER INDEX REBUILD` rebuilds an index in one atomic step, just like `CREATE INDEX` with the `DROP_EXISTING` clause. Since `ALTER INDEX REBUILD` also rebuilds the index physically, it allows SQL Server to assign fresh pages to reduce both internal and external fragmentation to a minimum. But unlike `CREATE INDEX` with the `DROP_EXISTING` clause, it allows an index (supporting either the `PRIMARY KEY` or `UNIQUE` constraint) to be rebuilt dynamically without dropping and re-creating the constraints.

In a columnstore index, the REBUILD statement will, in an offline fashion, completely rebuild the columnstore, invoking the Tuple Mover to remove the deltastore but also rearranging the data to ensure maximum effective compression. With rowstore indexes, the preferred mechanism for dealing with index fragmentation is the REBUILD. For columnstore indexes, the preferred method is the REORGANIZE statement, covered in detail in the next section.

To understand the use of `ALTER INDEX REBUILD` to defragment a rowstore index, consider the fragmented table used in the "Fragmentation Overhead" and "Analyzing the Amount of Fragmentation" sections. This table is repeated here:

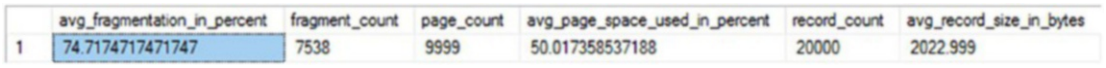
```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT,
                        C2 INT,
                        C3 INT,
                        c4 CHAR(2000));
```

```
CREATE CLUSTERED INDEX i1 ON dbo.Test1 (C1);

WITH Nums
AS (SELECT TOP (10000)
           ROW_NUMBER() OVER (ORDER BY (SELECT 1)) AS n
    FROM master.sys.all_columns AS ac1
        CROSS JOIN master.sys.all_columns AS ac2)
INSERT INTO dbo.Test1 (C1,
                       C2,
                       C3,
                       c4)
SELECT n,
       n,
       n,
       'a'
FROM Nums;

WITH Nums
AS (SELECT 1 AS n
    UNION ALL
    SELECT n + 1
    FROM Nums
    WHERE n < 10000)
INSERT INTO dbo.Test1 (C1,
                       C2,
                       C3,
                       c4)
SELECT 10000 - n,
       n,
       n,
       'a'
FROM Nums
OPTION (MAXRECURSION 10000);
```

If you take a look at the current fragmentation, you can see that it is both internally and externally fragmented (Figure 14-17).



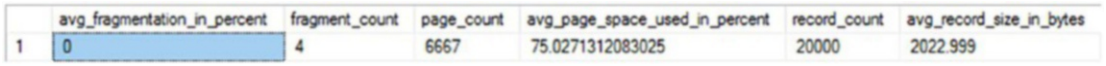
	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	74.7174717471747	7538	9999	50.017358537188	20000	2022.999

Figure 14-17. *Internal and external fragmentation*

You can defragment the clustered index (or the table) by using the `ALTER INDEX REBUILD` statement.

```
ALTER INDEX i1 ON dbo.Test1 REBUILD;
```

Figure 14-18 shows the resultant output of the standard `SELECT` statement against `sys.dm_db_index_physical_stats`.



	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	0	4	6667	75.0271312083025	20000	2022.999

Figure 14-18. *Fragmentation resolved by ALTER INDEX REBUILD*

Compare the preceding results of the query in Figure 14-18 with the earlier results in Figure 14-18. You can see that both internal and external fragmentation have been reduced efficiently. Here's an analysis of the output:

- *Internal fragmentation*: The table has 20,000 rows with an average row size (2,022.999 bytes) that allows a maximum of four rows per page. If the rows are highly defragmented to reduce the internal fragmentation to a minimum, then there should be about 6,000 data pages in the table (or leaf pages in the clustered index). You can observe the following in the preceding output:
  - *Number of leaf (or data) pages*: `pagecount` = 6667
  - *Amount of information in a page*: `avg_page_space_used_in_percent` = 75.02 percent
- *External fragmentation*: A large number of extents are required to hold the 6,667 pages. For a minimum of external fragmentation, there should not be any gap between the extents, and all pages should be physically arranged in the logical order of the index. The preceding output illustrates the number of out-of-order pages = `avg_ fragmentation_in_percent` = 0 percent. That is an effective defragmentation of this index. With fewer extents aligned with each other, access will be faster.