be using the ONLINE rebuild option as well. The ONLINE option radically reduces the blocking associated with the rebuild operation. To rebuild an index that is both ONLINE and RESUMABLE, you must specify all this in the command.

```
ALTER INDEX i1 ON dbo.Test1 REBUILD WITH (ONLINE=ON, RESUMABLE=ON);
```

This will run the index rebuild operation until it is completed or until you issue the following command:

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

This will pause the ONLINE rebuild operation, and the table and indexes in question will remain accessible without any blocking. To restart the operation, use this:

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

## Executing the ALTER INDEX REORGANIZE Statement

For a rowstore index, ALTER INDEX REORGANIZE reduces the fragmentation of an index without rebuilding the index. It reduces external fragmentation by rearranging the existing leaf pages of the index in the logical order of the index key. It compacts the rows within the pages, reducing internal fragmentation, and discards the resultant empty pages. This technique doesn't use any new pages for defragmentation.

For a columnstore index, ALTER INDEX REORGANIZE will ensure that the deltastore within the columnstore index gets cleaned out and that all the logical deletes are taken care of. It does this while keeping the index online and accessible. This will ensure that the index is defragmented. Further, you have the option of forcing the compression of all the row groups. This will function similarly to running ALTER INDEX REBUILD, but it continues to keep the index online during the operation, unlike the REBUILD process. Because of this, ALTER INDEX REORGANIZE is preferred for columnstore indexes.

To avoid the blocking overhead associated with ALTER INDEX REBUILD, this technique uses a nonatomic online approach. As it proceeds through its steps, it requests a small number of locks for a short period. Once each step is done, it releases the locks and proceeds to the next step. While trying to access a page, if it finds that the page is being used, it skips that page and never returns to the page again. This allows other queries to run on the table along with the ALTER INDEX REORGANIZE operation. Also, if this operation is stopped intermediately, then all the defragmentation steps performed up to then are preserved.

For a rowstore index, since ALTER INDEX REORGANIZE doesn't use any new pages to reorder the index and it skips the locked pages, the amount of defragmentation provided by this approach is usually less than that of ALTER INDEX REBUILD. To observe the relative effectiveness of ALTER INDEX REORGANIZE compared to ALTER INDEX REBUILD, rebuild the test table used in the previous section on ALTER INDEX REBUILD.

Rebuild the fragmented rowstore table using the script from before. To reduce the fragmentation of the clustered rowstore index, use ALTER INDEX REORGANIZE as follows:

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

Figure 14-19 shows the resultant output from 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 | 0.239377618192699 | 80 | 6684 | 74.836236718557 | 20000 | 2022.999 |

***Figure 14-19.*** *Results of ALTER INDEX REORGANIZE*

From the output, you can see that ALTER INDEX REORGANIZE doesn't reduce fragmentation as effectively as ALTER INDEX REBUILD, as shown in the previous section. For a highly fragmented index, the ALTER INDEX REORGANIZE operation can take much longer than rebuilding the index. Also, if an index spans multiple files, ALTER INDEX REORGANIZE doesn't migrate pages between the files. However, the main benefit of using ALTER INDEX REORGANIZE is that it allows other queries to access the table (or the indexes) simultaneously.

To see the results of defragmentation of a columnstore index, let's use the already fragmented columnstore index from the "Columnstore Overhead" section earlier in the chapter.

```
ALTER INDEX ClusteredColumnStoreTest ON dbo.bigTransactionHistory
REORGANIZE;
```

The results of the REORGANIZE statement are visible in Figure 14-20.

| | 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 | TOMBSTONE | 104086 | 5230 | 94 |
| 31 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 30 | TOMBSTONE | 750811 | 37419 | 95 |
| 32 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 31 | COMPRESSED | 812248 | 0 | 100 |

***Figure 14-20.*** *Results of REORGANIZE without compression on columnstore index*

You'll notice that most of the rowgroups are still somewhat fragmented. Two of the row groups (29,30) haven't been changed from `COMPRESSED` to `TOMBSTONE`. This means those rowgroups will be removed in the background at some later date. In short, only a few of the row groups were merged, and almost none of the deleted rows was dealt with. This is because the `REORGANIZE` command will clean up the deleted data only when more than 10 percent of the data in a rowgroup has been deleted. Let's remove more data from the table to bring some of the rowgroups to more than 10 percent deleted.

```
DELETE dbo.bigTransactionHistory
WHERE Quantity BETWEEN 8
                   AND     17;
```

Now when we look at the fragmentation results, we see a lot more activity, as shown in Figure 14-21.

| | TableName | IndexName | partition_number | row_group_id | state_description | total_rows | deleted_rows | PercentFull |
|---|---|---|---|---|---|---|---|---|
| 1 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 0 | TOMBSTONE | 1048576 | 104902 | 89 |
| 2 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 1 | TOMBSTONE | 1048576 | 104961 | 89 |
| 3 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 2 | COMPRESSED | 1048576 | 104147 | 90 |
| 4 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 3 | TOMBSTONE | 1048576 | 105212 | 89 |
| 5 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 4 | TOMBSTONE | 1048576 | 105101 | 89 |
| 6 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 5 | COMPRESSED | 1048576 | 104770 | 90 |
| 7 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 6 | TOMBSTONE | 1048576 | 104905 | 89 |
| 8 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 7 | TOMBSTONE | 1048576 | 105097 | 89 |
| 9 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 8 | COMPRESSED | 1048576 | 104170 | 90 |
| 10 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 9 | TOMBSTONE | 1048576 | 105264 | 89 |
| 11 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 10 | COMPRESSED | 1048576 | 104479 | 90 |
| 12 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 11 | COMPRESSED | 1048576 | 104813 | 90 |
| 13 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 12 | TOMBSTONE | 1048576 | 105276 | 89 |
| 14 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 13 | TOMBSTONE | 1048576 | 105195 | 89 |
| 15 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 14 | TOMBSTONE | 1048576 | 104944 | 89 |
| 16 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 15 | TOMBSTONE | 1048576 | 105181 | 89 |
| 17 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 16 | TOMBSTONE | 1048576 | 104969 | 89 |
| 18 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 17 | TOMBSTONE | 1048576 | 105001 | 89 |
| 19 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 18 | TOMBSTONE | 1048576 | 105264 | 89 |
| 20 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 19 | COMPRESSED | 1048576 | 104274 | 90 |
| 21 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 20 | COMPRESSED | 1048576 | 104847 | 90 |
| 22 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 21 | COMPRESSED | 1048576 | 104635 | 90 |
| 23 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 22 | TOMBSTONE | 1048576 | 104966 | 89 |
| 24 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 23 | COMPRESSED | 1048576 | 104850 | 90 |
| 25 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 24 | TOMBSTONE | 1048576 | 105189 | 89 |
| 26 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 25 | TOMBSTONE | 1048576 | 105050 | 89 |
| 27 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 26 | COMPRESSED | 1048576 | 104592 | 90 |
| 28 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 27 | TOMBSTONE | 1048576 | 104965 | 89 |
| 29 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 28 | COMPRESSED | 878208 | 87553 | 90 |
| 30 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 29 | COMPRESSED | 1025265 | 102585 | 89 |
| 31 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 30 | COMPRESSED | 943674 | 0 | 100 |
| 32 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 31 | COMPRESSED | 943671 | 0 | 100 |
| 33 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 32 | COMPRESSED | 943632 | 0 | 100 |
| 34 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 33 | COMPRESSED | 943615 | 0 | 100 |
| 35 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 34 | COMPRESSED | 943611 | 0 | 100 |
| 36 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 35 | COMPRESSED | 943610 | 0 | 100 |
| 37 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 36 | COMPRESSED | 943607 | 0 | 100 |
| 38 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 37 | COMPRESSED | 943575 | 0 | 100 |
| 39 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 38 | COMPRESSED | 943526 | 0 | 100 |
| 40 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 39 | COMPRESSED | 943479 | 0 | 100 |
| 41 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 40 | COMPRESSED | 943475 | 0 | 100 |
| 42 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 41 | COMPRESSED | 943395 | 0 | 100 |
| 43 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 42 | COMPRESSED | 943387 | 0 | 100 |
| 44 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 43 | COMPRESSED | 943381 | 0 | 100 |
| 45 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 44 | COMPRESSED | 943364 | 0 | 100 |
| 46 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 45 | COMPRESSED | 943312 | 0 | 100 |
| 47 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 46 | COMPRESSED | 943312 | 0 | 100 |
| 48 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 47 | COMPRESSED | 943300 | 0 | 100 |

***Figure 14-21.*** *REORGANIZE without compression against a more fragmented index*

You can see that many more rowgroups have been marked as TOMBSTONE and that all the new pages have zero deleted rows. You'll note that the row_group_id values have been generated for the row groups that have been compressed. The old row_group_id won't be reused. If you look at the table after the cleanup is complete, you'll see only the COMPRESSED row groups, bringing the total down to 30 because of the removed data, but you'll see gaps.

If we were to rerun the REORGANIZE command but include the row group, the command would look like this:

```
ALTER INDEX cci_bigTransactionHistory
ON dbo.bigTransactionHistory
REORGANIZE
WITH (COMPRESS_ALL_ROW_GROUPS = ON);
```

The command COMPRESS_ALL_ROW_GROUPS will ensure that any OPEN or CLOSED rowgroups in the deltastore will get moved into the columnstore going through the compression and everything else associated with a columnstore index.

Before running this, though, let's delete a little more data to push the rowgroups that are at 10 percent fragmentation over the top.

```
DELETE dbo.bigTransactionHistory
WHERE Quantity BETWEEN 6
               AND     8;
```

The results shown in Figure 14-22 include removing the TOMBSTONE, as well as reorganizing the index completely.

| | TableName | IndexName | partition_number | row_group_id | state_description | total_rows | deleted_rows | PercentFull |
|---|---|---|---|---|---|---|---|---|
| 1 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 2 | TOMBSTONE | 1048576 | 125061 | 88 |
| 2 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 5 | TOMBSTONE | 1048576 | 125925 | 87 |
| 3 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 8 | TOMBSTONE | 1048576 | 125388 | 88 |
| 4 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 10 | TOMBSTONE | 1048576 | 125386 | 88 |
| 5 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 11 | TOMBSTONE | 1048576 | 125754 | 88 |
| 6 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 19 | TOMBSTONE | 1048576 | 125134 | 88 |
| 7 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 20 | TOMBSTONE | 1048576 | 125572 | 88 |
| 8 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 21 | TOMBSTONE | 1048576 | 125648 | 88 |
| 9 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 23 | TOMBSTONE | 1048576 | 125789 | 88 |
| 10 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 26 | TOMBSTONE | 1048576 | 125558 | 88 |
| 11 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 28 | TOMBSTONE | 878208 | 105032 | 88 |
| 12 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 29 | TOMBSTONE | 1025265 | 123191 | 87 |
| 13 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 30 | COMPRESSED | 943674 | 21049 | 97 |
| 14 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 31 | COMPRESSED | 943671 | 20823 | 97 |
| 15 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 32 | COMPRESSED | 943632 | 20967 | 97 |
| 16 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 33 | COMPRESSED | 943615 | 21016 | 97 |
| 17 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 34 | COMPRESSED | 943611 | 21007 | 97 |
| 18 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 35 | COMPRESSED | 943610 | 21167 | 97 |
| 19 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 36 | COMPRESSED | 943607 | 20897 | 97 |
| 20 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 37 | COMPRESSED | 943575 | 21229 | 97 |
| 21 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 38 | COMPRESSED | 943526 | 20900 | 97 |
| 22 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 39 | COMPRESSED | 943479 | 20752 | 97 |
| 23 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 40 | COMPRESSED | 943475 | 21333 | 97 |
| 24 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 41 | COMPRESSED | 943395 | 21005 | 97 |
| 25 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 42 | COMPRESSED | 943387 | 20963 | 97 |
| 26 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 43 | COMPRESSED | 943381 | 20811 | 97 |
| 27 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 44 | COMPRESSED | 943364 | 20995 | 97 |
| 28 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 45 | COMPRESSED | 943312 | 21074 | 97 |
| 29 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 46 | COMPRESSED | 943312 | 21218 | 97 |
| 30 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 47 | COMPRESSED | 943300 | 20891 | 97 |
| 31 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 48 | COMPRESSED | 923515 | 0 | 100 |
| 32 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 49 | COMPRESSED | 923442 | 0 | 100 |
| 33 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 50 | COMPRESSED | 923190 | 0 | 100 |
| 34 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 51 | COMPRESSED | 923188 | 0 | 100 |
| 35 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 52 | COMPRESSED | 923018 | 0 | 100 |
| 36 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 53 | COMPRESSED | 923004 | 0 | 100 |
| 37 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 54 | COMPRESSED | 922928 | 0 | 100 |
| 38 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 55 | COMPRESSED | 922822 | 0 | 100 |
| 39 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 56 | COMPRESSED | 922787 | 0 | 100 |
| 40 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 57 | COMPRESSED | 922651 | 0 | 100 |
| 41 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 58 | COMPRESSED | 902074 | 0 | 100 |
| 42 | bigTransactionHistory | cci_bigTransactionHistory | 1 | 59 | COMPRESSED | 773176 | 0 | 100 |

***Figure 14-22.*** *Compression and defragmentation for the columnstore index*

Effectively, the order of the row groups doesn't really matter. As they get defragmented, they get moved from their original location within the index to a new location later.

If you don't want to deal with the 10 percent limitation, you can use the `REBUILD` option on the columnstore index, but you will have to deal with the fact that you're taking the index offline during that process.

Table 14-1 summarizes the characteristics of these four defragmentation techniques on a rowstore index.

***Table 14-1.*** *Characteristics of Rowstore Defragmentation Techniques*

| **Characteristics/ Issues** | **Drop and Create Index** | **Create Index with `DROP_ EXISTING`** | **`ALTER INDEX REBUILD`** | **`ALTER INDEX REORGANIZE`** |
|---|---|---|---|---|
| Rebuild nonclustered indexes on clustered index fragmentation | Twice | No | No | No |
| Missing indexes | Yes | No | No | No |
| Defragment index with constraints | Highly complex | Moderately complex | Easy | Easy |
| Defragment multiple indexes together | No | No | Yes | Yes |
| Concurrency with others | Low | Low | Medium, depending on concurrent user activity | High |
| Intermediate cancellation | Dangerous with no transaction | Progress lost | Progress lost | Progress preserved |
| Degree of defragmentation | High | High | High | Moderate to low |
| Apply new fill factor | Yes | Yes | Yes | No |
| Statistics are updated | Yes | Yes | Yes | No |

You can also reduce internal fragmentation by compressing more rows within a page, reducing free spaces within the pages. The maximum amount of compression that can be done within the leaf pages of an index is controlled by the fill factor, as you will see next.

When dealing with large databases and the indexes associated, it may become necessary to split up the tables and the indexes across disks using partitioning. Indexes on partitions can also become fragmented as the data within the partition changes.

When dealing with a portioned index, you will need to determine whether you want to either reorganize or rebuild (with `REORGANIZE` or `REBUILD`, respectively) one, some, or all partitions as part of the `ALTER INDEX` command. Partitioned indexes cannot be rebuilt online. Keep in mind that doing anything that affects all partitions is likely to be a costly operation.

If compression is specified on an index, even on a partitioned index, you must be sure to set the compression while performing the `ALTER INDEX` operation to what it was before; if you don't, it will be lost, and you'll have to rebuild the index. This is especially important for nonclustered indexes, which will not inherit the compression setting from the table.

## Defragmentation and Partitions

If you have massive databases, a standard mechanism for effectively managing the data is to break it up into partitions. While partitions can, in some rare cases, help with performance, they are foremost for managing data. But, one of the issues with indexes and partitions is that if you rebuild the index, it's unavailable during the rebuild. This means that with partitions, which are on massive indexes, you can expect to have a major portion of your data offline during the rebuild. SQL Server 2012 introduced the ability to do an online rebuild. If you had a partitioned index, it would look like this:

```
ALTER INDEX i1 ON dbo.Test1
REBUILD PARTITION = ALL
WITH (ONLINE = ON);
```

This can rebuild the entire partition and do it as an online operation, meaning it keeps the index largely available while it does the rebuild. But, for some partitions, this is a massive undertaking that will probably result in excessive load on the server and the need for a lot more tempdb storage. SQL Server 2014 introduced new functionality that lets you designate individual partitions.

```
ALTER INDEX i1 ON dbo.Test1
REBUILD PARTITION = 1
WITH (ONLINE = ON);
```

This reduces the overhead of the rebuild operation while still keeping the index mostly available during the rebuild. I do emphasize that it is "mostly" online because there is still some degree of locking and contention that will occur during the rebuild. It's not a completely free operation. It's just radically improved over the alternative.

Talking about the locking involved with index rebuild operations in partitions, you also have one other new piece of functionality introduced in SQL Server 2014. You can now modify the lock priority used during the rebuild operation by again adjusting the `REBUILD` command.

```
ALTER INDEX i1
ON dbo.Test1
REBUILD PARTITION = 1
WITH (ONLINE = ON (WAIT_AT_LOW_PRIORITY (MAX_DURATION = 20,
                                         ABORT_AFTER_WAIT = SELF)));
```

What this does is set the duration that the rebuild operation is willing to wait, in minutes. Then, it allows you to determine which processes get aborted in order to clear the system for the index rebuild. You can have it stop itself or the blocking process. The most interesting thing is that the waiting process is set to low priority, so it's not using a lot of system resources, and any transactions that come in won't be blocked by this process.

## Significance of the Fill Factor

On rowstore indexes, the internal fragmentation of an index is reduced by getting more rows per leaf page in an index. Getting more rows within a leaf page reduces the total number of pages required for the index and in turn decreases disk I/O and the logical reads required to retrieve a range of index rows. On the other hand, if the index key values are highly transactional, then having fully used index pages will cause page splits. Therefore, for a transactional table, a good balance between maximizing the number of rows in a page and avoiding page splits is required.

SQL Server allows you to control the amount of free space within the leaf pages of the index by using the *fill factor*. If you know that there will be enough `INSERT` queries on the table or `UPDATE` queries on the index key columns, then you can pre-add free space to the index leaf page using the fill factor to minimize page splits. If the table is read-only, you

can create the index with a high fill factor to reduce the number of index pages. It's a good idea to have some free space, though, when dealing with inserts against an IDENTITY column (or any index key that contains ordered data that will tend to create a hot page).

The default fill factor is 0, which means the leaf pages are packed to 100 percent, although some free space is left in the branch nodes of the B-tree structure. The fill factor for an index is applied only when the index is created. As keys are inserted and updated, the density of rows in the index eventually stabilizes within a narrow range. As you saw in the previous chapter's sections on page splits caused by UPDATE and INSERT, when a page split occurs, generally half the original page is moved to a new page, which happens irrespective of the fill factor used during the index creation.

To understand the significance of the fill factor, let's use a small test table with 24 rows.

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT,
                        C2 CHAR(999));

WITH Nums
AS (SELECT 1 AS n
    UNION ALL
    SELECT n + 1
    FROM Nums
    WHERE n < 24)
INSERT INTO dbo.Test1 (C1,
                       C2)
SELECT n * 100,
       'a'
FROM Nums;
```

Increase the maximum number of rows in the leaf (or data) page by creating a clustered index with the default fill factor.

```
CREATE CLUSTERED INDEX FillIndex ON Test1(C1);
```

Since the average row size is 1,010 bytes, a clustered index leaf page (or table data page) can contain a maximum of eight rows. Therefore, at least three leaf pages are required for the 24 rows. You can confirm this in the sys.dm_db_index_physical_stats output shown in Figure 14-23.

| | avg_fragmentation_in_percent | fragment_count | page_count | avg_page_space_used_in_percent | record_count | avg_record_size_in_bytes |
|---|---|---|---|---|---|---|
| 1 | 33.3333333333333 | 2 | 3 | 100 | 24 | 1010 |

***Figure 14-23.*** *Fill factor set to default value of 0*

Note that `avg_page_space_used_in_percent` is 100 percent since the default fill factor allows the maximum number of rows to be compressed in a page. Since a page cannot contain a part row to fill the page fully, `avg_page_space_used_in_percent` will be often a little less than 100 percent, even with the default fill factor.

To reduce the initial frequency of page splits caused by `INSERT` and `UPDATE` operations, create some free space within the leaf (or data) pages by re-creating the clustered index with a fill factor as follows:

```
ALTER INDEX FillIndex ON dbo.Test1 REBUILD
WITH  (FILLFACTOR= 75);
```

Because each page has a total space for eight rows, a fill factor of 75 percent will allow six rows per page. Thus, for 24 rows, the number of leaf pages should increase to four, as in the `sys.dm_db_index_physical_stats` output shown in Figure 14-24.

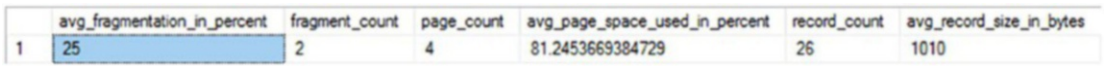
| | avg_fragmentation_in_percent | fragment_count | page_count | avg_page_space_used_in_percent | record_count | avg_record_size_in_bytes |
|---|---|---|---|---|---|---|
| 1 | 25 | 2 | 4 | 74.9938225846306 | 24 | 1010 |

***Figure 14-24.*** *Fill factor set to 75*

Note that `avg_page_space_used_in_percent` is about 75 percent, as set by the fill factor. This allows two more rows to be inserted in each page without causing a page split. You can confirm this by adding two rows to the first set of six rows (`C1` = 100 – 600, contained in the first page).

```
INSERT  INTO dbo.Test1
VALUES  (110, 'a'),  --25th row
        (120, 'a') ;  --26th row
```

Figure 14-25 shows the current fragmentation.



| | avg_fragmentation_in_percent | fragment_count | page_count | avg_page_space_used_in_percent | record_count | avg_record_size_in_bytes |
|---|---|---|---|---|---|---|
| 1 | 25 | 2 | 4 | 81.2453669384729 | 26 | 1010 |

***Figure 14-25.*** *Fragmentation after new records*

From the output, you can see that the addition of the two rows has not added any pages to the index. Accordingly, `avg_page_space_used_in_percent` increased from 74.99 percent to 81.25 percent. With the addition of two rows to the set of the first six rows, the first page should be completely full (eight rows). Any further addition of rows within the range of the first eight rows should cause a page split and thereby increase the number of index pages to five.

```
INSERT  INTO dbo.Test1
VALUES  (130, 'a') ;  --27th row
```

Now `sys.dm_db_index_physical_stats` displays the difference in Figure 14-26.

| | avg_fragmentation_in_percent | fragment_count | page_count | avg_page_space_used_in_percent | record_count | avg_record_size_in_bytes |
|---|---|---|---|---|---|---|
| 1 | 40 | 3 | 5 | 67.4919693600198 | 27 | 1010 |

***Figure 14-26.*** *Number of pages goes up*

Note that even though the fill factor for the index is 75 percent, `Avg. Page Density (full)` has decreased to 67.49 percent, which can be computed as follows:

```
Avg. Page Density (full)
= Average rows per page / Maximum rows per page
= (27 / 5) / 8
= 67.5%
```

From the preceding example, you can see that the fill factor is applied when the index is created. But later, as the data is modified, it has no significance. Irrespective of the fill factor, whenever a page splits, the rows of the original page are distributed between two pages, and `avg_page_space_used_in_percent` settles accordingly. Therefore, if you use a nondefault fill factor, you should ensure that the fill factor is reapplied regularly to maintain its effect.

You can reapply a fill factor by re-creating the index or by using `ALTER INDEX REORGANIZE` or `ALTER INDEX REBUILD`, as was shown. `ALTER INDEX REORGANIZE` takes the fill factor specified during the index creation into account. `ALTER INDEX REBUILD` also takes the original fill factor into account, but it allows a new fill factor to be specified, if required.

Without periodic maintenance of the fill factor, for both default and nondefault fill factor settings, `avg_page_space_used_in_percent` for an index (or a table) eventually settles within a narrow range.

An argument can be made that rather than attempt to defragment indexes over and over again, with all the overhead that implies, you could be better off settling on a fill factor that allows for a fairly standard set of distribution across the pages in your indexes. Some people do use this method, sacrificing some read performance and disk space to avoid page splits and the associated issues in which they result. Testing on your own systems to both find the right fill factor and determine whether that method works will be necessary.

## Automatic Maintenance

In a database with a great deal of transactions, tables and indexes become fragmented over time (assuming you're not using the fill factor method just mentioned). Thus, to improve performance, you should check the fragmentation of the tables and indexes regularly, and you should defragment the ones with a high amount of fragmentation. You also may need to take into account the workload and defragment indexes as dictated by the load as well as the fragmentation level of the index. You can do this analysis for a database by following these steps:

1. Identify all user tables in the current database to analyze fragmentation.
2. Determine fragmentation of every user table and index.
3. Determine user tables and indexes that require defragmentation by taking into account the following considerations:
    - A high level of fragmentation where `avg_fragmentation_in_percent` is greater than 20 percent
    - Not a very small table/index—that is, `pagecount` is greater than 8
4. Defragment tables and indexes with high fragmentation.

For a fully functional script that includes a large degree of capability, I strongly recommend using the Minion Reindex application located at `http://bit.ly/2EGsmYU` or Ola Hollengren's scripts at `http://bit.ly/JijaNI`.

In addition to those scripts, you can use the maintenance plans built into SQL Server. However, I don't recommend them because you surrender a lot of control for a little bit of ease of use. You'll be much happier with the results you get from one of the sets of scripts recommended earlier.

# Summary

As you learned in this chapter, in a highly transactional database, page splits caused by `INSERT` and `UPDATE` statements may fragment the tables and indexes, increasing the cost of data retrieval. You can avoid these page splits by maintaining free spaces within the pages using the fill factor. Since the fill factor is applied only during index creation, you should reapply it at regular intervals to maintain its effectiveness. Data manipulation of columnstore indexes also leads to fragmentation and performance degradation. You can determine the amount of fragmentation in an index (or a table) using `sys.dm_db_index_physical_stats` for a rowstore index or using `sys.column_store_row_groups` for a columnstore index. Upon determining a high amount of fragmentation, you can use either `ALTER INDEX REBUILD` or `ALTER INDEX REORGANIZE`, depending on the required amount of defragmentation, the database concurrency, and whether you are dealing with a rowstore or columnstore index.

Defragmentation rearranges the data so that its physical order on the disk matches its logical order in the table/index, thus improving the performance of queries. However, unless the optimizer decides upon an effective execution plan for the query, query performance even after defragmentation can remain poor. Therefore, it is important to have the optimizer use efficient techniques to generate cost-effective execution plans.

In the next chapter, I explain execution plan generation and the techniques the optimizer uses to decide upon an effective execution plan.

CHAPTER 15

# Execution Plan Generation

The performance of any query depends on the effectiveness of the execution plan decided upon by the optimizer, as you learned in previous chapters. Because the overall time required to execute a query is the sum of the time required to generate the execution plan plus the time required to execute the query based on this execution plan, it is important that the cost of generating the execution plan itself is low or that a plan gets reused from cache, avoiding that cost altogether. The cost incurred when generating the execution plan depends on the process of generating the execution plan, the process of caching the plan, and the reusability of the plan from the plan cache. In this chapter, you will learn how an execution plan is generated.

In this chapter, I cover the following topics:

- Execution plan generation and caching
- The SQL Server components used to generate an execution plan
- Strategies to optimize the cost of execution plan generation
- Factors affecting parallel plan generation

## Execution Plan Generation

SQL Server uses a cost-based optimization technique to determine the processing strategy of a query. The optimizer considers both the metadata of the database objects, such as unique constraints or index size, and the current distribution statistics of the columns referred to in the query when deciding which index and join strategies should be used.

The cost-based optimization allows a database developer to concentrate on implementing a business rule, rather than on the exact syntax of the query. At the same time, the process of determining the query-processing strategy remains quite complex and can consume a fair amount of resources. SQL Server uses a number of techniques to optimize resource consumption.




G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_15

- Syntax-based optimization of the query
- Trivial plan match to avoid in-depth query optimization for simple queries
- Index and join strategies based on current distribution statistics
- Query optimization in stepped phases to control the cost of optimization
- Execution plan caching to avoid the unnecessary regeneration of query plans

The following techniques are performed in order, as shown in Figure 15-1.

1. Parsing
2. Binding
3. Query optimization
4. Execution plan generation, caching, and hash plan generation
5. Query execution

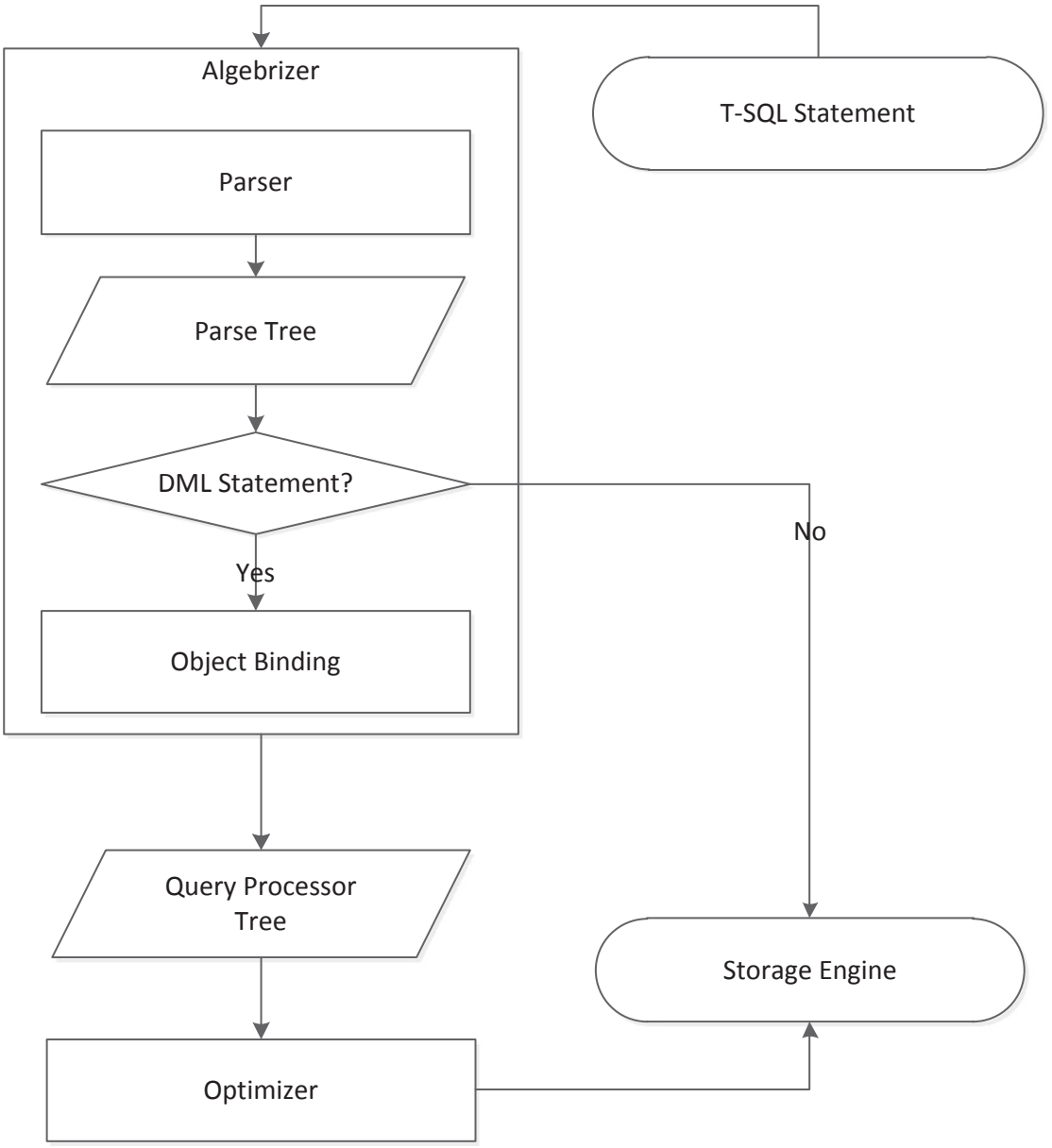



***Figure 15-1.*** *SQL Server techniques to optimize query execution*

Let's take a look at these steps in more detail.

## Parser

When a query is submitted, SQL Server passes it to the algebrizer within the *relational engine*. (This relational engine is one of the two main parts of SQL Server data retrieval and manipulation, with the other being the *storage engine*, which is responsible for data access, modifications, and caching.) The relational engine takes care of parsing, name and type resolution, and optimization. It also executes a query as per the query execution plan and requests data from the storage engine.

The first part of the algebrizer process is the parser. The parser checks an incoming query, validating it for the correct syntax. The query is terminated if a syntax error is detected. If multiple queries are submitted together as a batch as follows (note the error in syntax), then the parser checks the complete batch together for syntax and cancels the complete batch when it detects a syntax error. (Note that more than one syntax error may appear in a batch, but the parser goes no further than the first one.)

```
CREATE TABLE dbo.Test1 (c1 INT);
INSERT  INTO dbo.Test1
VALUES  (1);
CEILEKT * FROM dbo.t1; --Error:  I meant,  SELECT * FROM t1
```

On validating a query for correct syntax, the parser generates an internal data structure called a *parse tree* for the algebrizer. The parser and algebrizer taken together are called *query compilation.*

## Binding

The parse tree generated by the parser is passed to the next part of the algebrizer for processing. The algebrizer now resolves all the names of the different objects, meaning the tables, the columns, and so on, that are being referenced in the T-SQL in a process called *binding*. It also identifies all the various data types being processed. It even checks for the location of aggregates (such as `GROUP BY` and `MAX`). The output of all these verifications and resolutions is a binary set of data called a *query processor tree.*

To see this part of the algebrizer in action, if the following batch query is submitted, then the first three statements before the error statement are executed, and the errant statement and the one after it are cancelled.

```
IF (SELECT OBJECT_ID('dbo.Test1')) IS NOT NULL
    DROP TABLE dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT);
INSERT INTO dbo.Test1
VALUES (1);
SELECT 'Before Error',
       C1
FROM dbo.Test1 AS t;
SELECT 'error',
       c1
FROM dbo.no_Test1;
--Error:  Table doesn't exist
SELECT 'after error' AS c1
FROM dbo.Test1 AS t;
```

If a query contains an implicit data conversion, then the normalization process adds an appropriate step to the query tree. The process also performs some syntax-based transformation. For example, if the following query is submitted, then the syntax-based optimization transforms the syntax of the query, as shown in the T-SQL in Figure 15-2 taken from the SELECT operator properties in the execution plan, where BETWEEN becomes >= and <=.

```
SELECT soh.AccountNumber,
       soh.OrderDate,
       soh.PurchaseOrderNumber,
       soh.SalesOrderNumber
FROM Sales.SalesOrderHeader AS soh
WHERE soh.SalesOrderID BETWEEN 62500
                       AND     62550;
```

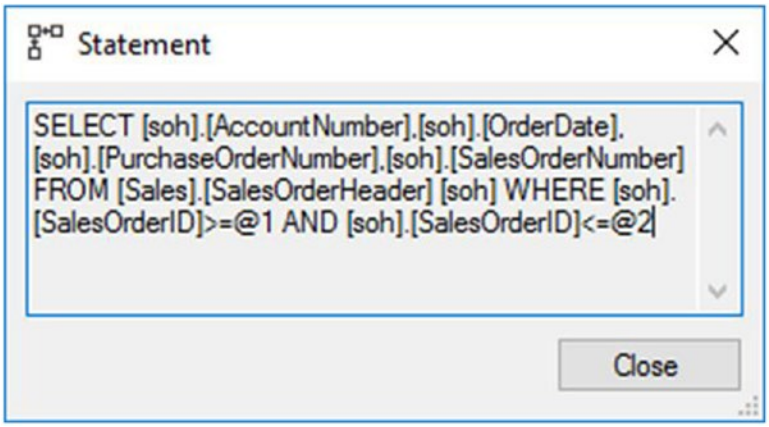



***Figure 15-2.*** *Syntax-based optimization*

You can also see some evidence of parameterization, discussed in more detail later in this chapter. The execution plan generated from the query looks like Figure 15-3.


***Figure 15-3.*** *Execution plan with a warning*

You should also note the warning indicator on the `SELECT` operator. Looking at the properties for this operator, you can see that `SalesOrderID` is actually getting converted as part of the process and the optimizer is warning you.

```
Type conversion in expression (CONVERT(nvarchar(23),[soh].
[SalesOrderID],0)) may affect "CardinalityEstimate" in query plan choice
```

I left this example in, with the warning, to illustrate a couple of points. First, warnings can be unclear. In this case, the warning is coming from the calculated column, `SalesOrderNumber`. It's doing a conversion of the `SalesOrderID` to a string and adding a value to it. In the way it does it, the optimizer recognizes that this could be problematic, so it gives you a warning. But, you're not referencing the column in any kind of filtering fashion such as the `WHERE` clause, `JOIN`s, or `HAVING`. Because of that, you can safely ignore the warning. I also left it in because it illustrates just fine that AdventureWorks is a good example database because it has the same types of poor choices that are sometimes in databases in the real world too.