```
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.

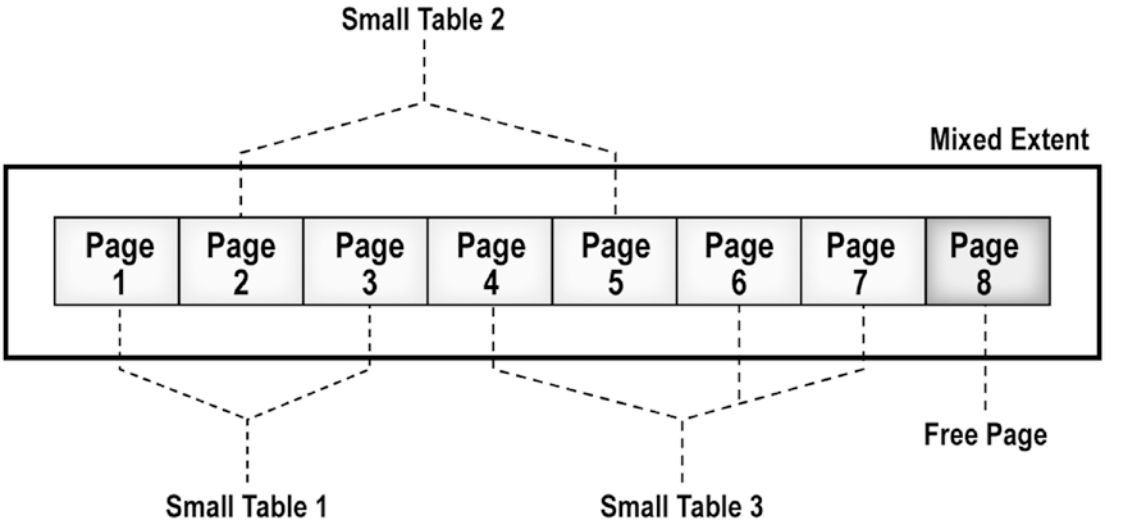


Figure 14-15. *Mixed extent*

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:

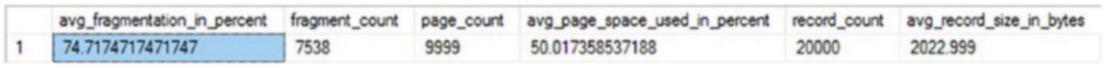
```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT,
                        C2 INT,
                        C3 INT,
                        c4 CHAR(2000));
```

```
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).



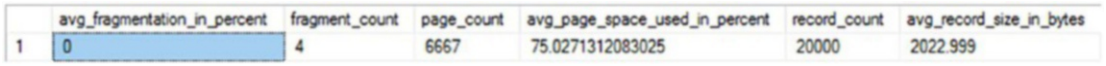
	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	74.7174717471747	7538	9999	50.017358537188	20000	2022.999

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.

Rebuilding an index in SQL Server 2005 and newer will also compact the large object (LOB) pages. You can choose not to by setting a value of `LOB_COMPACTION = OFF`. If you aren't worried about storage but you are concerned about how long your index reorganization is taking, this might be advisable to turn off.

When you use the `PAD_INDEX` setting while creating an index, it determines how much free space to leave on the index intermediate pages, which can help you deal with page splits. This is taken into account during the index rebuild, and the new pages will be set back to the original values you determined at the index creation unless you specify otherwise. I've almost never seen this make a major difference on most systems. You'll need to test on your system to determine whether it can help.

If you don't specify otherwise, the default behavior is to defragment all indexes across all partitions. If you want to control the process, you just need to specify which partition you want to rebuild when.

As shown previously, the `ALTER INDEX REBUILD` technique effectively reduces fragmentation. You can also use it to rebuild *all* the indexes of a table in one statement.

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

Although this is the most effective defragmentation technique, it does have some overhead and limitations.

- *Blocking*: Similar to the previous two index-rebuilding techniques, `ALTER INDEX REBUILD` introduces blocking in the system. It blocks all other queries trying to access the table (or any index on the table). It can also be blocked by those queries. You can reduce this using `ONLINE INDEX REBUILD`.
- *Transaction rollback*: Since `ALTER INDEX REBUILD` is fully atomic in action, if it is stopped before completion, then all the defragmentation actions performed up to that time are lost. You can run `ALTER INDEX REBUILD` using the `ONLINE` keyword, which will reduce the locking mechanisms, but it will increase the time involved in rebuilding the index.

Introduced in Azure SQL Database and available in SQL Server 2017, you now have the capacity to restart an index rebuild operation. You can restart a failed index rebuild, or you can pause the rebuild operation only to restart it later. To do this, you have to