For this index, you can see that the page compression was able to move the index from 106 pages to 25, of which 25 were compressed. The row type compression in this instance made a difference in the number of pages in the index but was not nearly as dramatic as that of the page compression.

To see that compression works for you without any modification to code, run the following query:

```
SELECT  a.City,
        a.PostalCode
FROM    Person.Address AS a
WHERE   a.City = 'Newton'
        AND a.PostalCode = 'V2M1N7';
```

The optimizer chose, on my system, to use the `IX_Comp_Page_Test` index. Even if I forced it to use the `IXTest` index thusly, the performance was identical, although one extra page was read in the second query:

```
SELECT  a.City,
        a.PostalCode
FROM    Person.Address AS a WITH (INDEX = IX_Test)
WHERE   a.City = 'Newton'
        AND a.PostalCode = 'V2M1N7';
```

So, although one index is taking up radically less room on approximately one-quarter as many pages, it's done at no cost in performance.

Compression has a series of impacts on other processes within SQL Server, so further understanding of the possible impacts as well as the possible benefits should be explored thoroughly prior to implementation. In most cases, the cost to the CPU is completely outweighed by the benefits everywhere else, but you should test and monitor your system.

Clean up the indexes after you finish testing.

```
DROP INDEX Person.Address.IX_Test;
DROP INDEX Person.Address.IX_Comp_Test;
DROP INDEX Person.Address.IX_Comp_Page_Test;
```

Columnstore Indexes

Introduced in SQL Server 2012, the columnstore index is used to index information by columns rather than by rows. This is especially useful when working within data warehousing systems where large amounts of data have to be aggregated and accessed quickly. The information stored within a columnstore index is grouped on each column, and these groupings are stored individually. This makes aggregations on different sets of columns extremely fast since the columnstore index can be accessed rather than accessing large numbers of rows in order to aggregate the information. Further, you get more speed because the storage is column oriented, so you'll be touching storage only for the columns you're interested in, not the entire row of columns. Finally, you'll see some performance enhancements from columnstore because the columnar data is stored compressed. The columnstore comes in two types, similar to regular indexes: a clustered columnstore and a nonclustered columnstore. Prior to SQL Server 2016, the nonclustered column store cannot be updated. You must drop it and then re-create it (or, if you're using partitioning, you can switch in and out different partitions). From SQL Server 2016 onward, you can use a nonclustered columnstore inside your transactional database to enable real-time analytic queries. A clustered column store was introduced in SQL Server 2014 and is available there and only in the Enterprise version for production machines. In SQL Server 2016 and SQL Server 2017, the columnstore is available in all editions. There are a number of limits on using columnstore indexes.

You can't use certain data types such as `binary`, `text`, `varchar(max) (supported in SQL Server 2017)`, `uniqueidentifier` (in SQL Server 2012, this data type works in SQL Server 2014 and greater), `clr` data types, or `xml`.

- You can't create a columnstore index on a sparse column.
- A table on which you want to create a clustered columnstore can't have any constraints including primary key or foreign key constraints.

For the complete list of restrictions, refer to SQL Server Books Online.

Columnstores are primarily meant for use within data warehouses and therefore work best when dealing with the associated styles of storage such as star schemas. Because of how the data is stored within the columnstore index, you'll see columnstores used frequently when dealing with partitioned data. The way a columnstore index is designed, it functions optimally when dealing with large data sets of at least 100,000 rows. In the AdventureWorks2017 database, none of the tables as configured is sufficiently large to really put the columnstore to work. To have enough data, I'm going to use Adam Machanic's script, `make_big_adventure.sql`, to create a couple of large tables, `dbo.bigTransactionHistory` and `dbo.bigProduct`. The script can be downloaded at `http://bit.ly/2mNBIhg`.

Columnstore Index Storage

The real beauty of the columnstore indexes is that with a clustered columnstore and a nonclustered columnstore, you can tailor the behavior of the storage within your system to the purposes of that system without sacrificing other query behavior. If your system is a data warehouse with large fact tables, you can use the clustered columnstore to define your data storage since the vast majority of the queries will benefit from that clustered columnstore. However, if you have an OLTP system on which you occasionally need to run analysis style queries, you can use the nonclustered columnstore in addition to your regular clustered and nonclustered indexes, also called *rowstore indexes*.

The following are the benefits of the columnstore index:

- Enhanced performance in data warehouse and analytic work loads
- Excellent data compression
- Reduced I/O
- Mode data that fits in memory

To understand the columnstore more completely, I should define a few terms.

- *Rowgroup*: A group of rows compressed and stored in a column-wise fashion.
- *Segment*: Also called a column segment, a column of data compressed and stored on disk. Each rowgroup has a column segment for every column in the table.
- *Dictionary*: Encoding for some data types that defines the segment. These can be global, for all segments, or local, used for one segment.

The columnstore data is not stored in a B-tree as the rowstore indexes are. Instead, the data is pivoted and aggregated on each column within the table. The information is also broken into subsets called *rowgroups*. Each rowgroup consists of up to 1,048,576 rows. When the data is loaded in a batch into a columnstore, it is automatically broken into rowgroups if the number of rows exceeds 100,000. As data gets updated in columnstore indexes, changes are stored in what is called the *deltastore*. This is actually a B-tree index controlled behind the scenes by the SQL Server engine. Added rows are accumulated in the deltastore until there are 102,400 of them, and then they will be pivoted and compressed into the rowgroups. The process that does this is called the *tuple mover*. Deletes of rows from columnstores depend on where the row is at. A row in the deltastore is simply removed. A row that is already compressed into a rowgroup goes through a logical delete. Another B-tree index, again controlled out of sight, manages a list of identifiers for the rows removed. An update works similarly, consisting of a delete (logical or actual, depending on location) and an insert into the deltastore.

If you're doing your loading in small batches, with lots of updates, you will be dealing with the deltastore. This is extremely likely in the event that you're using a nonclustered columnstore index on a rowstore table. By and large the deltastore manages itself. However, it's not a bad idea to, when possible, rebuild the columnstore index to clear out the logically deleted rows and get compressed rowgroups. You can do this using the `ALTER INDEX REORGANIZE` command. We'll cover that in detail in Chapter 14.

The pivoted, grouped, and compressed storage of the columnstore lends itself to incredible performance enhancements when dealing with grouped data. However, it's much slower and more problematic when doing the kind of single-row or range lookups that are needed for OLTP-style queries.

The behavior of the clustered and nonclustered columnstore indexes is basically the same. The difference is that the clustered columnstore, like the clustered rowstore index, is storing the data. The nonclustered columnstore, on the other hand, must have the data stored and managed elsewhere in a rowstore index.

Columnstore Index Behavior

Take this query as an example:

```
SELECT bp.Name AS ProductName,
       COUNT(bth.ProductID),
       SUM(bth.Quantity),
       AVG(bth.ActualCost)
FROM dbo.bigProduct AS bp
    JOIN dbo.bigTransactionHistory AS bth
        ON bth.ProductID = bp.ProductID
GROUP BY bp.Name;
```

If you run this query against the tables as they are currently configured, you'll see an execution plan that looks like Figure 9-14.

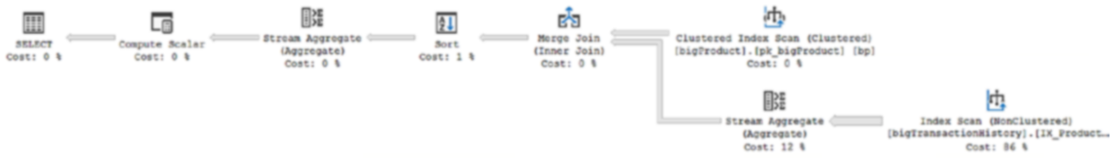


Figure 9-14. *Multiple aggregations for a GROUP BY query*

The reads and execution time for the query are as follows:

```
Table 'Worktable'. Scan count 0, logical reads 0
Table 'bigTransactionHistory'. Scan count 1, logical reads 131819
Table 'bigProduct'. Scan count 1, logical reads 601
CPU time = 16 ms,  elapsed time = 13356 ms.
```

There are a large number of reads, and this query uses quite a bit of CPU and is not terribly fast to execute. We have two types of columnstore indexes to choose from. If you want to just add a nonclustered columnstore index to an existing table, it's possible. We could migrate the data here to a clustered columnstore, but the behavior of the query is the same. For simplicity in the example then, we'll just use the nonclustered columnstore. When you create the nonclustered columnstore index, you can pick the columns to avoid any that might not be supported by the columnstore index.

```
CREATE NONCLUSTERED COLUMNSTORE INDEX ix_csTest
ON dbo.bigTransactionHistory
(
    ProductID,
    Quantity,
    ActualCost
);
```

With the nonclustered columnstore index in place, the optimizer now has the option of using that index to satisfy the previous query. Just like all other indexes available to the optimizer, costs are associated with the columnstore index, so it may or may not be chosen to satisfy the requirements for any given query against the table. In this case, if you rerun the original aggregate query, you can see that the optimizer determined that the costs associated with using the columnstore index were beneficial to the query. The execution plan now looks like Figure 9-15.

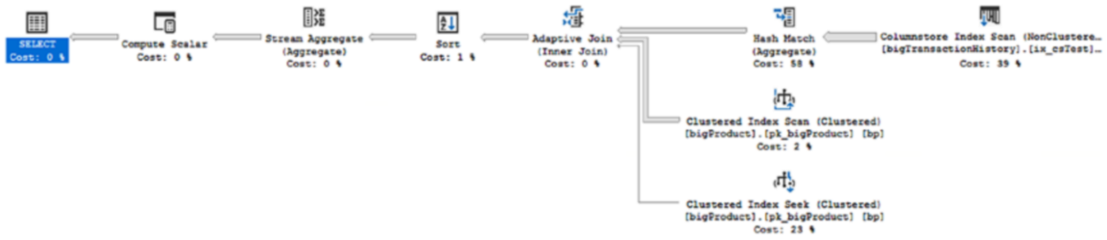

SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Stream Aggregate
(Aggregate)
Cost: 0 %
Sort
Cost: 1 %
Adaptive Join
(Inner Join)
Cost: 0 %
Hash Match
(Aggregate)
Cost: 58 %
Columnstore Index Scan (NonClustere…
[bigTransactionHistory].[ix_csTest]…
Cost: 39 %
Clustered Index Scan (Clustered)
[bigProduct].[pk_bigProduct] [bp]
Cost: 2 %
Clustered Index Seek (Clustered)
[bigProduct].[pk_bigProduct] [bp]
Cost: 23 %

Figure 9-15. *The columnstore index is used instead of the clustered index*

As you can see, there are a number of differences in the plan. There's a lot to unpack here, but before we do, let's take a look at the reads and execution time. The results are identical: 24,975 rows on my system. The real differences are seen in the reads and execution times for the query.

```
Table 'bigTransactionHistory'. Scan count 4, logical reads 0
Table 'bigTransactionHistory'. Segment reads 31, segment skipped 0.
Table 'bigProduct'. Scan count 3, logical reads 620
Table 'Worktable'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
CPU time = 1922 ms,  elapsed time = 1554 ms.
```

The radical reduction in the number of reads required to retrieve the data and the marginal increase in speed are all the result of being able to reference information that is indexed by column instead of by row. We went from 13.3 seconds to 1.5 seconds on the execution time. That's the kind of massive performance enhancements you can look forward to.

Let's unpack the execution plan a little because this is the first really complex plan we've seen. The first thing to note is that the optimizer chose to make this a parallel plan. You can see that in the operators that have a yellow symbol attached like the `Columnstore Index Scan` operator in Figure 9-16.

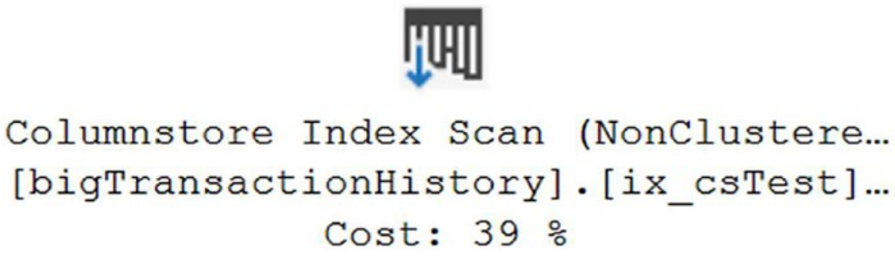

Columnstore Index Scan (NonClustere…
[bigTransactionHistory].[ix_csTest]…
Cost: 39 %

Figure 9-16. *A Columnstore Index Scan operator in parallel execution*

There's a new processing method for dealing with data called *batch mode*. Currently, only queries that contain columnstore indexes have batch mode processing, but Microsoft has already announced that this will change. Batch mode deals with rows in batches within the operations of a plan. This is a huge advantage. Row mode processing means that each row goes through a negotiation process as it moves between operators in the plan: 10,000 rows, 10,000 negotiations. That is very intensive. Batch mode moves rows in batches instead of individually. The batches are approximately evenly distributed up to 1,000 rows per batch (although this varies). That means instead of 10,000 negotiations, there are only 10 to move the 10,000 rows. That is a gigantic performance benefit. Further, batch mode takes advantage of multiple processors to help speed up execution. To determine the execution mode of the operators in a plan, look to the properties of that operator. Figure 9-17 shows the appropriate property for the `Columnstore Index Scan`.

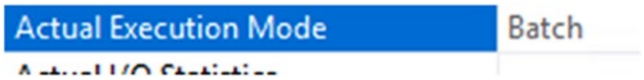

Actual Execution Mode Batch

Figure 9-17. *Actual execution mode*

Batch mode processing is the preferred method when dealing with columnstore indexes because it is generally much faster than the alternative, row mode. Prior to SQL Server 2017, it generally required a parallel execution plan before a query would enter batch mode processing. However, SQL Server 2017 allows for batch mode processing in nonparallel execution plans.

There is a limited set of operations, documented in SQL Server Books Online, that result in batch mode processing, but when working with those operations on a system with enough processors, you will see yet another substantial performance enhancement.

Columnstore indexes don't require you to have the columns in a particular order, unlike clustered and nonclustered indexes. Also, unlike these other indexes, you should place multiple columns within a columnstore index so that you get benefits across those columns. Put another way, if you anticipate that you'll need to query the column at some point, add it proactively to the columnstore index definition. But if you're retrieving large numbers of columns from a columnstore index, you might see some performance degradation.

Another aspect of columnstore indexes that enhances performance is segment elimination. Each segment shows the minimum and maximum values within the segment (either with actual values or with a reference to a dictionary). If a segment won't contain a given value, it's just skipped. This becomes especially relevant when you're combining partitioning with columnstore indexes. Then, even if you don't get partition elimination, the segment elimination will effectively skip a partition if none of the data in segments contained in that partition matches the criteria we're filtering on.

There's an additional behavior of columnstore indexes visible in the execution plan in Figure 9-15. Introduced in SQL Server 2017 and in Azure SQL Database is the batch mode adaptive join. Let's look at an expanded view of a subset of the plan in Figure 9-18.

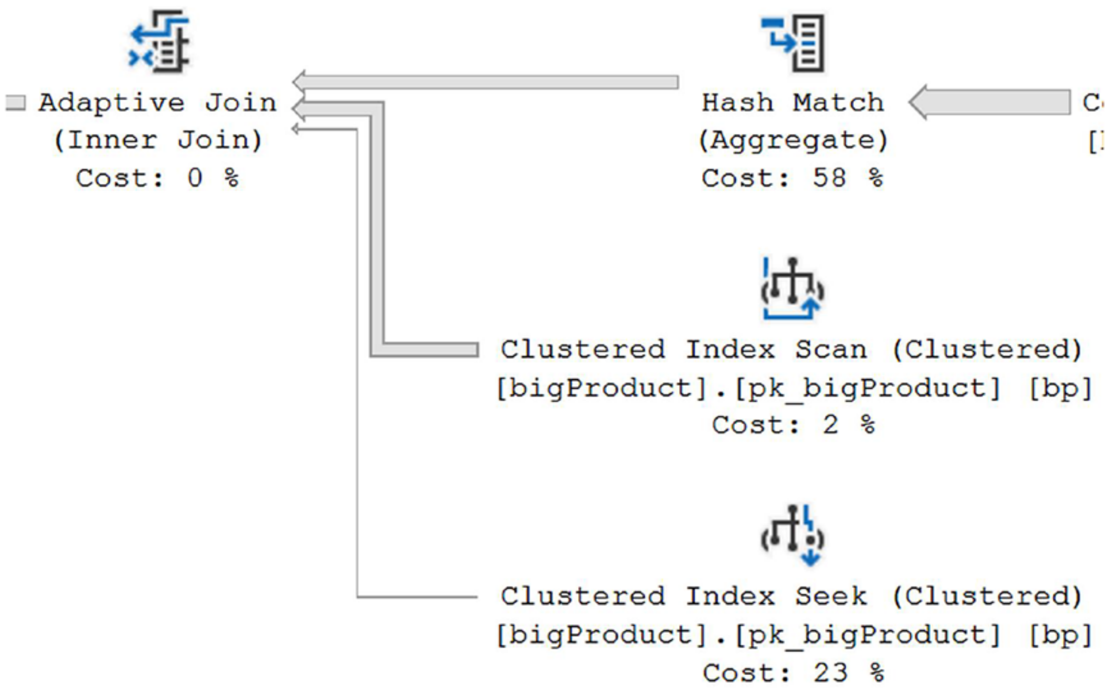
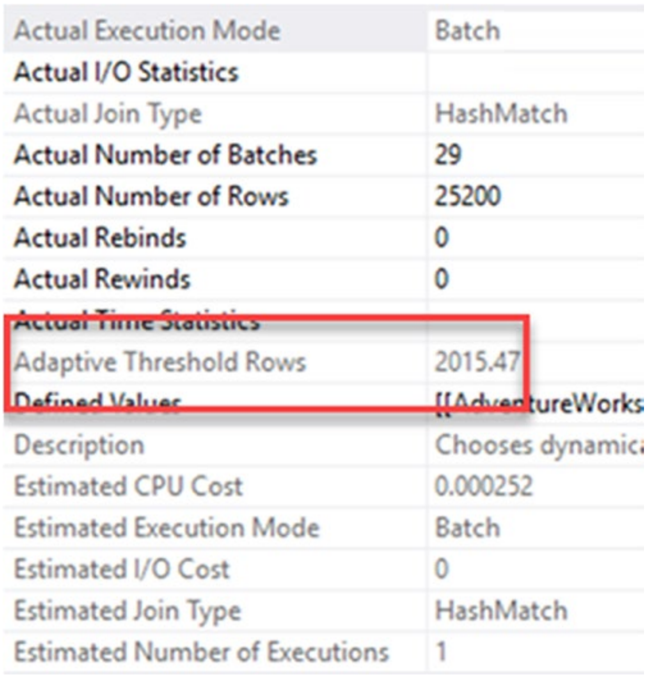

Adaptive Join
(Inner Join)
Cost: 0 %
Hash Match
(Aggregate)
Cost: 58 %
C
[
Clustered Index Scan (Clustered)
[bigProduct].[pk_bigProduct] [bp]
Cost: 2 %
Clustered Index Seek (Clustered)
[bigProduct].[pk_bigProduct] [bp]
Cost: 23 %

Figure 9-18. *Adaptive join and its attendant behavior*

Because selecting the wrong join type can so severely hurt performance, a new style of join has been added, the adaptive join. The adaptive join will create two possible branches for a given execution plan. You can see the two branches in Figure 9-18 as `Clustered Index Scan` and `Clustered IndexSeek`, both against the `pk_bigProduct` index. The adaptive join can decide, while executing, to use either a hash join or a nested loop join. It does this by loading data into an adaptive buffer, managed internally; we can't see it. If the row threshold is not reached, that buffer becomes the outer row driver for the loops join. Otherwise, a hash table gets built to do a normal hash join. Once the table is created, though, it can determine based on row counts which join type is better. After the adaptive join picks the type of join it intends to use, it will then go down one of the two branches. The top branch is for the hash match join, and the bottom is for a loops join. The information for the determination of a given join type is stored with the execution plan within the properties, as shown in Figure 9-19.



Actual Execution Mode	Batch
Actual I/O Statistics	
Actual Join Type	HashMatch
Actual Number of Batches	29
Actual Number of Rows	25200
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Adaptive Threshold Rows	2015.47
Defined Values	[[AdventureWorks
Description	Chooses dynamic
Estimated CPU Cost	0.000252
Estimated Execution Mode	Batch
Estimated I/O Cost	0
Estimated Join Type	HashMatch
Estimated Number of Executions	1

Figure 9-19. *A subset of the adaptive join properties*

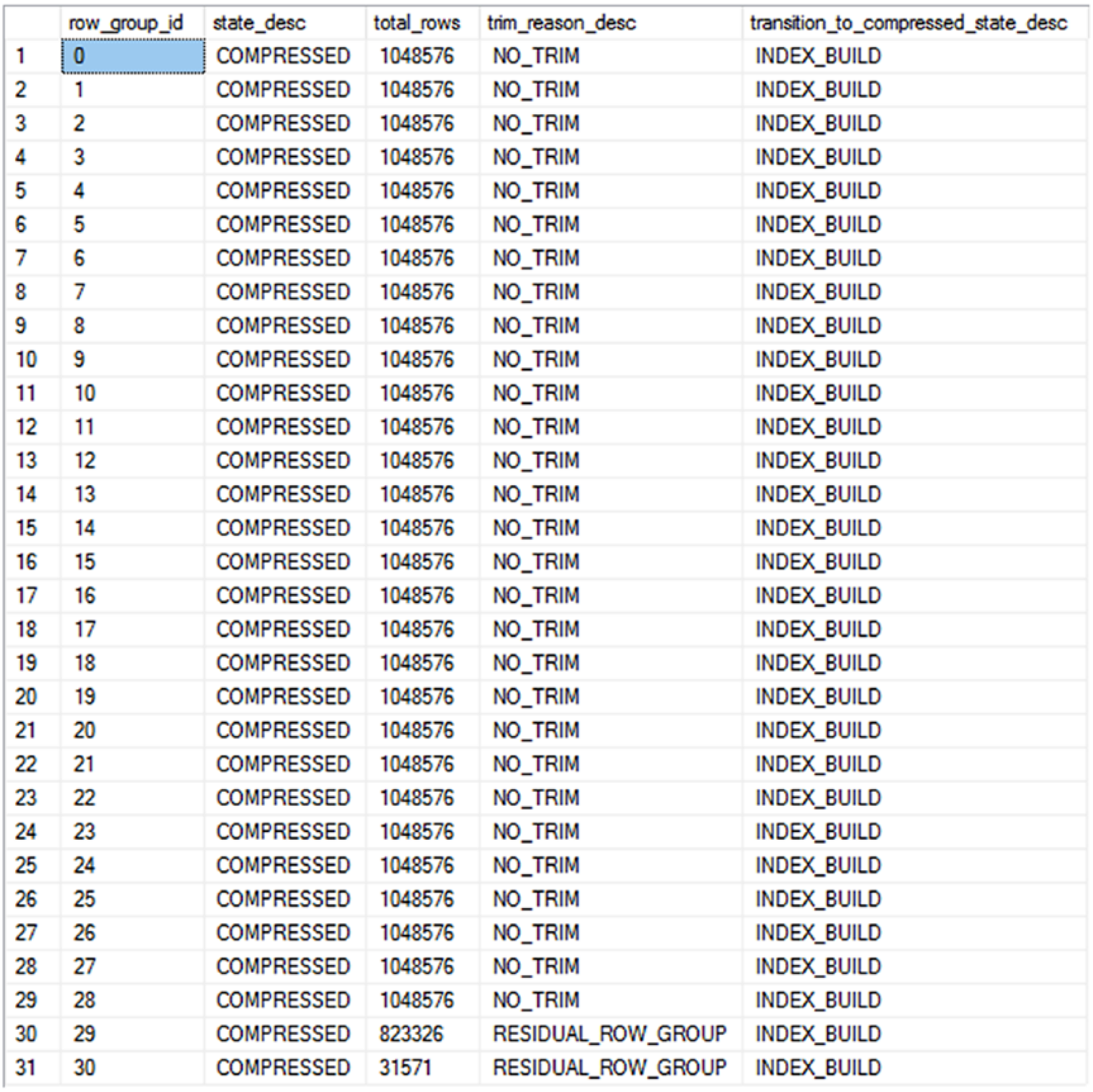
About midway down the properties shown in Figure 9-19 is the Adaptive Threshold Rows property. When the number of rows in the hash table is at or below this value, the adaptive join will use the loops join. Above the same value, the adaptive join will use the hash match join. You can also see properties for the estimated and actual join type used, so you can see how the behavior of a given query changes as the data it accesses also changes.

You can also see from this join that you can mix and match querying between columnstore and row store tables at will. The same basic rules always apply.

There are a number of DMOs you can use to look at the status of your columnstore indexes. One that's immediately useful is sys.dm_db_column_store_row_group_physical_stats. It shows the status of the row groups, and it's easy to query it.

```
SELECT ddcsrgps.row_group_id,
       ddcsrgps.state_desc,
       ddcsrgps.total_rows,
       ddcsrgps.trim_reason_desc,
       ddcsrgps.transition_to_compressed_state_desc
FROM sys.dm_db_column_store_row_group_physical_stats AS ddcsrgps
WHERE ddcsrgps.object_id = OBJECT_ID('dbo.bigTransactionHistory')
ORDER BY ddcsrgps.row_group_id DESC;
```

The output of the columnstore index from dbo.bigTransationHistory looks like Figure 9-20.



	row_group_id	state_desc	total_rows	trim_reason_desc	transition_to_compressed_state_desc
1	0	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
2	1	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
3	2	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
4	3	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
5	4	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
6	5	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
7	6	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
8	7	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
9	8	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
10	9	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
11	10	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
12	11	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
13	12	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
14	13	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
15	14	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
16	15	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
17	16	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
18	17	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
19	18	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
20	19	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
21	20	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
22	21	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
23	22	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
24	23	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
25	24	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
26	25	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
27	26	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
28	27	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
29	28	COMPRESSED	1048576	NO_TRIM	INDEX_BUILD
30	29	COMPRESSED	823326	RESIDUAL_ROW_GROUP	INDEX_BUILD
31	30	COMPRESSED	31571	RESIDUAL_ROW_GROUP	INDEX_BUILD

Figure 9-20. *Output of sys.dm_db_column_store_row_group_physical_stats*

You can now see how the rows were loaded and grouped in the index, whether or not there is compression and how the rows were moved by looking at the `transition_to_compressed_state_desc`.

I'm going to leave the tables and the columnstore index in place for later examples in the book.

Recommendations

First, you should always focus on picking the correct clustered index for the data in question. Generally, an OLTP system will benefit the most from rowstore, B-tree indexes. Equally generally, a data warehouse, reporting, or analysis system will benefit the most from columnstore indexes. There are likely to be exceptions in either direction, but that should be the essential guide.

Because you can add rowstore indexes to a clustered columnstore and you can add a nonclustered columnstore to rowstore tables, you can deal with exceptional behavior in either situation. A columnstore is ideal for tables with large numbers of rows. Smaller tables may still gain some benefits but may not. Test on your system to know for sure.

When dealing with columnstore indexes, you should generally follow these rules:

- Load the data into the columnstore in either a single transaction, if possible, or, if not, in batches that are greater than 102,400 to take advantage of the compressed rowgroups.
- Minimize small-scale updates to data within a clustered columnstore to avoid the overhead of dealing with the deltastore.
- Plan to have an index rebuild periodically based on data movement for both clustered and nonclustered columnstores to eliminate deleted data completely from the rowgroups and to move modified data from the deltastore into the rowgroups.
- Maintain the statistics on your columnstore indexes similar to how you do the same on your rowstore indexes. While they are not visible in the same way as rowstore indexes, they still must be maintained.

Special Index Types

As special data types and storage mechanisms are introduced to SQL Server by Microsoft, methods for indexing these special storage types are also developed. Explaining all the details possible for each of these special index types is outside the scope of the book. In the following sections, I introduce the basic concepts of each index type to facilitate the possibility of their use in tuning your queries.

Full-Text

You can store large amounts of text in SQL Server by using the MAX value in the VARCHAR, NVARCHAR, CHAR, and NCHAR fields. A normal clustered or nonclustered index against these large fields would be unsupportable because a single value can far exceed the page size within an index. So, a different mechanism of indexing text is to use the full-text engine, which must be running to work with full-text indexes. You can also build a full-text index on VARBINARY data.

You need to have one column on the table that is unique. The best candidates for performance are integers: INT or BIGINT. This column is then used along with the word to identify which row within the table it belongs to, as well as its location within the field. SQL Server allows for incremental changes, either change tracking or time-based, to the full-text indexes as well as complete rebuilds.

SQL Server 2012 introduced another method for working with text called *semantic search*. It uses phrases from documents to identify relationships between different sets of text stored within the database.

Spatial

Introduced in SQL Server 2008 is the ability to store spatial data. This data can be either a geometry type or the very complex geographical type, literally identifying a point on the earth. To say the least, indexing this type of data is complicated. SQL Server stores these indexes in a flat B-tree, similar to regular indexes, except that it is also a hierarchy of four grids linked together. Each of the grids can be given a density of low, medium, or high, outlining how big each grid is. There are mechanisms to support indexing of the spatial data types so that different types of queries, such as finding when one object is within the boundaries or near another object, can benefit from performance increases inherent in indexing.

A spatial index can be created only against a column of type geometry or geography. It has to be on a base table, it must have no indexed views, and the table must have a primary key. You can create up to 249 spatial indexes on any given column on a table. Different indexes are used to define different types of index behavior. More information is available in the book *Pro Spatial with SQL Server 2012* by Alastair Aitchison (Apress, 2012).

XML

Introduced as a data type in SQL Server 2005, XML can be stored not as text but as well-formed XML data within SQL Server. This data can be queried using the XQuery language as supported by SQL Server. To enhance the performance capabilities, a special set of indexes has been defined. An XML column can have one primary and several secondary indexes. The primary XML shreds the properties, attributes, and elements of the XML data and stores it as an internal table. There must be a primary key on the table, and that primary key must be clustered in order to create an XML index. After the XML index is created, the secondary indexes can be created. These indexes have types `Path`, `Value`, and `Property`, depending on how you query the XML. For more details, check out *Expert Performance Indexing in SQL Server* by Jason Strate and Grant Fritchey (Apress, 2015).

Additional Characteristics of Indexes

Other index properties can affect performance, positively and negatively. A few of these behaviors are explored here.

Different Column Sort Order

SQL Server supports creating a composite index with a different sort order for the different columns of the index. Suppose you want an index with the first column sorted in ascending order and the second column sorted in descending order to eliminate a sort operation, which can be quite costly. You could achieve this as follows:

```
CREATE NONCLUSTERED INDEX i1 ON t1(c1 ASC,  c2 DESC);
```

Index on Computed Columns

You can create an index on a computed column, as long as the expression defined for the computed column meets certain restrictions, such as that it references columns only from the table containing the computed column and it is deterministic.

Index on BIT Data Type Columns

SQL Server allows you to create an index on columns with the BIT data type. The ability to create an index on a BIT data type column by itself is not a big advantage since such a column can have only two unique values, except for the rare circumstance where the vast majority of the data is one value and only a few rows are the other. As mentioned previously, columns with such low selectivity (number of unique values) are not usually good candidates for indexing. However, this feature comes into its own when you consider covering indexes. Because covering indexes require including all the columns in the index, the ability to add the BIT data type column to an index key allows covering indexes to have such a column, if required (outside of the columns that would be part of the INCLUDE operator).

CREATE INDEX Statement Processed As a Query

The CREATE INDEX operation is integrated into the query processor. The optimizer can use existing indexes to reduce scan cost and sort while creating an index.

Take, for example, the Person.Address table. A nonclustered index exists on a number of columns: AddressLine1, AddressLine2, City, StateProvinceId, and PostalCode. If you needed to run queries against the City column with the existing index, you'll get a scan of that index. Now create a new index like this:

```
CREATE NONCLUSTERED INDEX IX_Test
ON Person.Address(City);
```

You can see in Figure 9-21 that, instead of scanning the table, the optimizer chose to scan the index to create the new index because the column needed for the new index was contained within the other nonclustered index.

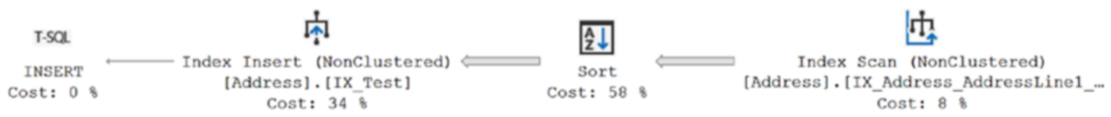

T-SQL
INSERT
Cost: 0 %
Index Insert (NonClustered)
[Address].[IX_Test]
Cost: 34 %
Sort
Cost: 58 %
Index Scan (NonClustered)
[Address].[IX_Address_AddressLine1_...
Cost: 8 %

Figure 9-21. *Execution plan for CREATE INDEX*

Be sure to drop the index when you're done.

```
DROP INDEX IX_Test ON Person.Address;
```

Parallel Index Creation

SQL Server supports parallel plans for a CREATE INDEX statement, as supported in other SQL queries. On a multiprocessor machine, index creation won't be restricted to a single processor but will benefit from the multiple processors. You can control the number of processors to be used in a CREATE INDEX statement with the max degree of parallelism configuration parameter of SQL Server. The default value for this parameter is 0, as you can see by executing the sp_configure stored procedure (after setting show advanced options).

```
EXEC sp_configure
    'max degree of parallelism' ;
```

The default value of 0 means that SQL Server can use all the available CPUs in the system for the parallel execution of a T-SQL statement. On a system with four processors, the maximum degree of parallelism can be set to 2 by executing spconfigure.

```
EXEC sp_configure
    'max degree of parallelism',
    2 ;
RECONFIGURE WITH OVERRIDE ;
```

This allows SQL Server to use up to two CPUs for the parallel execution of a T-SQL statement. This configuration setting takes effect immediately, without a server restart.

The query hint MAXDOP can be used for the CREATE INDEX statement. Also, be aware that the parallel CREATE INDEX feature is available only in SQL Server Enterprise editions.

Online Index Creation

The default creation of an index is done as an offline operation. This means exclusive locks are placed on the table, restricting user access while the index is created. It is possible to create the indexes as an online operation. This allows users to continue to access the data while the index is being created. This comes at the cost of increasing the amount of time and resources it takes to create the index. Introduced in SQL Server 2012, indexes with varchar(MAX), nvarchar(MAX), and nbinary(MAX) can actually be rebuilt online. Online index operations are available only in SQL Server Enterprise editions.

Considering the Database Engine Tuning Advisor

A simple approach to indexing is to use the Database Engine Tuning Advisor tool provided by SQL Server. This tool is a usage-based tool that looks at a particular workload and works with the query optimizer to determine the costs associated with various index combinations. Based on the tool's analysis, you can add or drop indexes as appropriate.

Note I will cover the Database Engine Tuning Advisor tool in more depth in Chapter 10.

Summary

In this chapter, you learned that there are a number of additional functions in and around indexes that expand on the behavior defined the preceding chapter.

In the next chapter, you will learn more about the Database Engine Tuning Advisor, the SQL Server–provided tool that can help you determine the correct indexes in a database for a given SQL workload.

CHAPTER 10

Database Engine Tuning Advisor

SQL Server's performance frequently depends upon having the proper indexes on the database tables. However, as the workload and data change over time, the existing indexes may not be entirely appropriate, and new indexes may be required. The task of deciding upon the correct indexes is complicated by the fact that an index change that benefits one set of queries may be detrimental to another set of queries.

To help you through this process, SQL Server provides a tool called the Database Engine Tuning Advisor. This tool can help identify an optimal set of indexes and statistics for a given workload without requiring an expert understanding of the database schema, workload, or SQL Server internals. It can also recommend tuning options for a small set of problem queries. In addition to the tool's benefits, I cover its limitations in this chapter because it is a tool that can cause more harm than good if used without deliberate intent.

In this chapter, I cover the following topics:

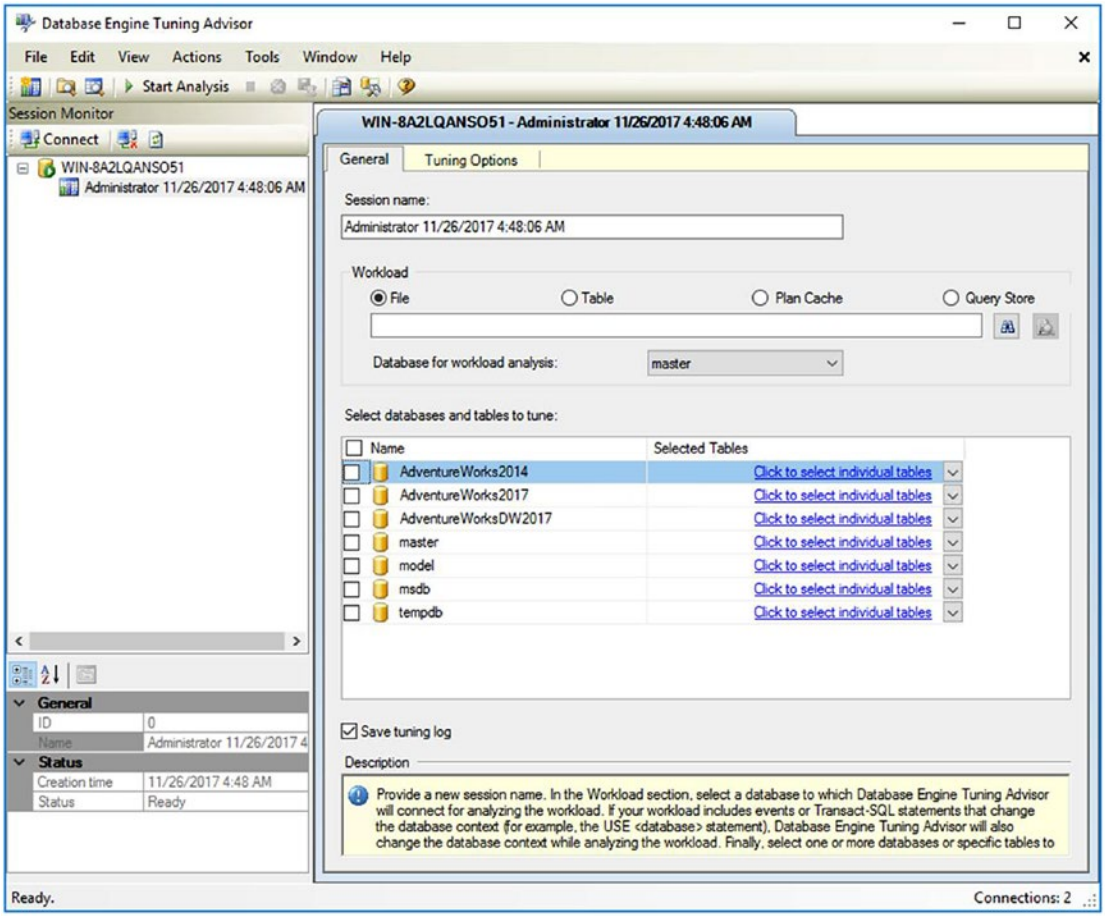
- How the Database Engine Tuning Advisor works
- How to use the Database Engine Tuning Advisor on a set of problematic queries for index recommendations, including how to define traces
- The limitations of the Database Engine Tuning Advisor

Database Engine Tuning Advisor Mechanisms

You can run the Database Engine Tuning Advisor directly by selecting Microsoft SQL Server 2017 ➤ SQL Server 2017 Database Engine Tuning Advisor. You can also run it from the command prompt (`dta.exe`), from SQL Profiler (Tools ➤ Database Engine

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_10

Tuning Advisor), from a query in Management Studio (highlight the required query and select Query ➤ Analyze Query in the Database Engine Tuning Advisor), or from Management Studio (select Tools ➤ Database Engine Tuning Advisor). Once the tool is open and you're connected to a server, you should see a window like the one in Figure 10-1. I'll run through the options to define and run an analysis in this section and then follow up in the next section with some detailed examples.


Database Engine Tuning Advisor
File
Edit
View
Actions
Tools
Window
Help
Start Analysis
Session Monitor
Connect
WIN-8A2LQANSO51
Administrator 11/26/2017 4:48:06 AM
WIN-8A2LQANSO51 - Administrator 11/26/2017 4:48:06 AM
General
Tuning Options

Session name: Administrator 11/26/2017 4:48:06 AM

Workload: [✓ File] Table Plan Cache Query Store

Database for workload analysis: master

Select databases and tables to tune:

	Name	Selected Tables
☐	AdventureWorks2014	Click to select individual tables
☐	AdventureWorks2017	Click to select individual tables
☐	AdventureWorksDW2017	Click to select individual tables
☐	master	Click to select individual tables
☐	model	Click to select individual tables
☐	msdb	Click to select individual tables
☐	tempdb	Click to select individual tables

☑ Save tuning log

Description

Provide a new session name. In the Workload section, select a database to which Database Engine Tuning Advisor will connect for analyzing the workload. If your workload includes events or Transact-SQL statements that change the database context (for example, the USE <database> statement), Database Engine Tuning Advisor will also change the database context while analyzing the workload. Finally, select one or more databases or specific tables to

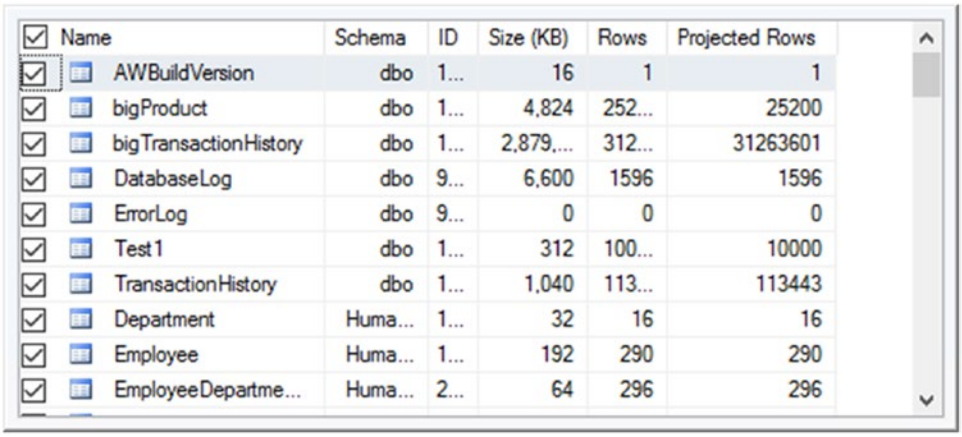
General	
ID	0
Name	Administrator 11/26/2017 4
Status	
Creation time	11/26/2017 4:48 AM
Status	Ready

Ready. Connections: 2

Figure 10-1. *Selecting the server and database in the Database Engine Tuning Advisor*

The Database Engine Tuning Advisor is already connected to a server. From here, you begin to outline the workload and the objects you want to tune. Creating a session name is necessary to label the session for documentation purposes. Then you need to pick a workload. The workload can come from a trace file or a table, from queries that exist in the plan cache, or from queries in the Query Store (the Query Store will be covered in detail in Chapter 11). Finally, you need to browse to the appropriate location. The workload is defined depending on how you launched the Database Engine Tuning Advisor. If you launched it from a query window, you would see a Query radio button, and the File and Table radio buttons would be disabled. You also have to define the Database for Workload Analysis setting and finally select a database to tune.

When you select a database, you can also select individual tables to be tuned by clicking the drop-down box on the right side of the screen; you'll see a list of tables like those in Figure 10-2.



✓	Name	Schema	ID	Size (KB)	Rows	Projected Rows
✓	AWBuildVersion	dbo	1...	16	1	1
✓	bigProduct	dbo	1...	4,824	252...	25200
✓	bigTransactionHistory	dbo	1...	2,879,...	312...	31263601
✓	DatabaseLog	dbo	9...	6,600	1596	1596
✓	ErrorLog	dbo	9...	0	0	0
✓	Test1	dbo	1...	312	100...	10000
✓	TransactionHistory	dbo	1...	1,040	113...	113443
✓	Department	Huma...	1...	32	16	16
✓	Employee	Huma...	1...	192	290	290
✓	EmployeeDepartme...	Huma...	2...	64	296	296

Figure 10-2. *Clicking the boxes defines individual tables for tuning in the Database Engine Tuning Advisor*

Once you define the workload, you need to select the Tuning Options tab, which is shown in Figure 10-3.