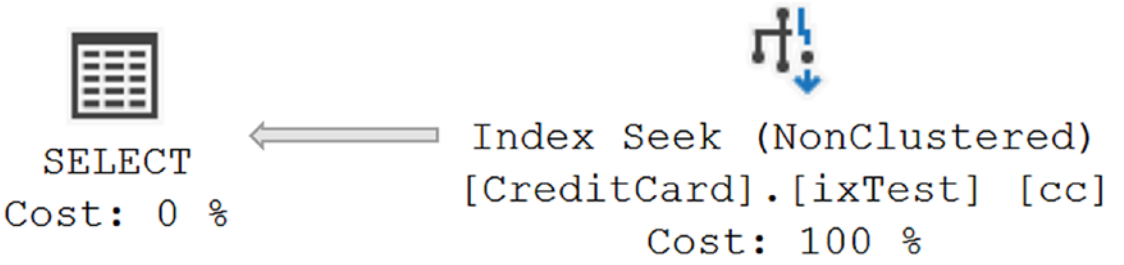


Figure 8-23. *Execution plan with a nonclustered index*

In this case, the `SELECT` statement doesn't include any column that requires a jump from the nonclustered index page to the data page of the table, which is what usually makes a nonclustered index costlier than a clustered index for a large result set and/or sorted output. This kind of nonclustered index is called a *covering index*.

It's also worth noting that I experimented with which column to put into the leading edge of the index, `ExpMonth` or `ExpYear`. After testing each, the reads associated with having `ExpMonth` first were 37 as compared to the 12 with `ExpYear`. That results from having to look through fewer pages with the year filtering first rather than the month. Remember to validate your choices when creating indexes with thorough testing.

Clean up the index after the testing is done.

```
DROP INDEX Sales.CreditCard.ixTest;
```

Summary

In this chapter, you learned that indexing is an effective method for reducing the number of logical reads and disk I/O for a query. Although an index may add overhead to action queries, even action queries such as `UPDATE` and `DELETE` can benefit from an index.

To decide the index key columns for a particular query, evaluate the `WHERE` clause and the join criteria of the query. Factors such as column selectivity, width, data type, and column order are important in deciding the columns in an index key. Since an index is mainly useful in retrieving a small number of rows, the selectivity of an indexed column should be very high. It is important to note that nonclustered indexes contain the value of a clustered index key as their row locator because this behavior greatly influences the selection of an index type.

In the next chapter, you will learn more about other functionality and other types of indexes available to help you tune your queries.

CHAPTER 9

Index Analysis

In the previous chapter I introduced the concepts surrounding B-tree indexes. This chapter takes that information and adds more functionality and more indexes. There's a lot of interesting interaction between indexes that you can take advantage of. There are also a number of settings that affect the behavior of indexes that I didn't address in the preceding chapter. I'll show you methods to squeeze even more performance out of your system. Most importantly, we dig into the details of the columnstore indexes and the radical improvement in performance that they can provide for analytical queries.

In this chapter, I cover the following topics:

- Advanced indexing techniques
- Columnstore indexes
- Special index types
- Additional characteristics of indexes

Advanced Indexing Techniques

Here are a few of the more advanced indexing techniques that you can consider:

- *Covering indexes*: These were introduced in Chapter 8.
- *Index intersections*: Use multiple nonclustered indexes to satisfy all the column requirements (from a table) for a query.
- *Index joins*: Use the index intersection and covering index techniques to avoid hitting the base table.
- *Filtered indexes*: To be able to index fields with odd data distributions or sparse columns, you can apply a filter to an index so that it indexes only some data.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_9

- *Indexed views*: These materialize the output of a view on disk.
- *Index compression*: The storage of indexes can be compressed through SQL Server, putting more rows of data on a page and improving performance.

I cover these topics in more detail in the following sections.

Covering Indexes

A *covering index* is a nonclustered index built upon all the columns required to satisfy a SQL query without going to the heap or the clustered index. If a query encounters an index and does not need to refer to the underlying structures at all, then the index can be considered a covering index.

For example, in the following `SELECT` statement, irrespective of where the columns are used within the statement, all the columns (`StateProvinceId` and `PostalCode`) should be included in the nonclustered index to cover the query fully:

```
SELECT  a.PostalCode
FROM    Person.Address AS a
WHERE   a.StateProvinceID = 42;
```

Then all the required data for the query can be obtained from the nonclustered index page, without accessing the data page. This helps SQL Server save logical and physical reads. If you run the query, you'll get the following I/O and execution time as well as the execution plan in Figure 9-1:

```
Table 'Address'. Scan count 1, logical reads 19
CPU time = 0 ms,  elapsed time = 0 ms.
```

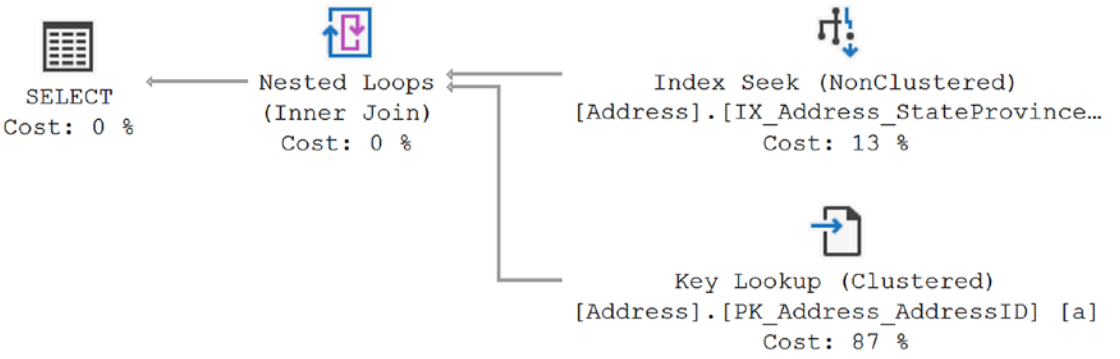


Figure 9-1. *Query without a covering index*

Here you have a classic lookup with the `Key Lookup` operator pulling the `PostalCode` data from the clustered index and joining it with the `Index Seek` operator against the `IX_Address_StateProvinceId` index.

Although you can re-create the index with both key columns, another way to make an index a covering index is to use the new `INCLUDE` operator. This stores data with the index without changing the structure of the index. I'll cover the details of why to use the `INCLUDE` operator a little later. Use the following to re-create the index:

```
CREATE NONCLUSTERED INDEX IX_Address_StateProvinceID
ON Person.Address
(
    StateProvinceID ASC
)
INCLUDE
(
    PostalCode
)
WITH (DROP_EXISTING = ON);
```

If you rerun the query, the execution plan (Figure 9-2), I/O, and execution time change. (Also, it's worth noting, 0ms is not the correct execution time. Using an Extended Events session, which records execution time in microseconds (µs), it's 177µs.

```
Table 'Address'. Scan count 1, logical reads 2
CPU time = 0 ms,  elapsed time = 0 ms.
```

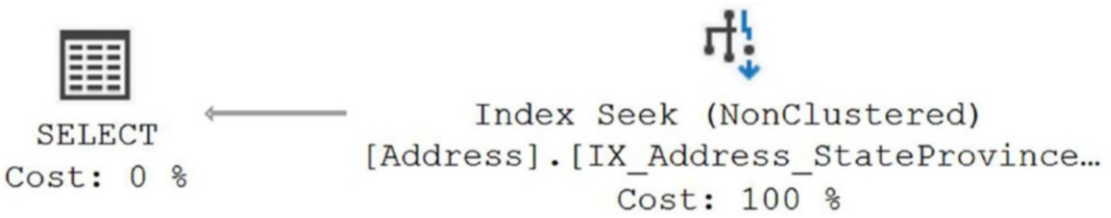


Figure 9-2. *Query with a covering index*

The reads have dropped from 19 to 2, and the execution plan is just about as simple as possible; it's a single Index Seek operation against the new and improved index, which is now covering. A covering index is a useful technique for reducing the number of logical reads of a query. Adding columns using the INCLUDE statement makes this functionality easier to achieve without adding to the number of columns in an index or the size of the index key since the included columns are stored only at the leaf level of the index.

The INCLUDE is best used in the following cases:

- You don't want to increase the size of the index keys, but you still want to make the index a covering index.
- You have a data type that cannot be an index key column but can be added to the nonclustered index through the INCLUDE command.
- You've already exceeded the maximum number of key columns for an index (although this is a problem best avoided).

Before continuing, put the index back into its original format.

```
CREATE NONCLUSTERED INDEX IX_Address_StateProvinceID
ON Person.Address
(
    StateProvinceID ASC
)
WITH (DROP_EXISTING = ON);
```

A Pseudoclustered Index

The covering index physically organizes the data of all the indexed columns in a sequential order. Thus, from a disk I/O perspective, a covering index that doesn't use included columns becomes a clustered index for all queries satisfied completely by the columns in the covering index. If the result set of a query requires a sorted output, then the covering index can be used to physically maintain the column data in the same order as required by the result set—it can then be used in the same way as a clustered index for sorted output. As shown in the previous example, covering indexes can give better performance than clustered indexes for queries requesting a range of rows and/or sorted output. The included columns are not part of the key and therefore wouldn't offer the same benefits for ordering as the key columns of the index.

Recommendations

To take advantage of covering indexes, be careful with the column list in SELECT statements to move only the data you need to (thus the standard prohibition against SELECT *). It's also a good idea to use as few columns as possible to keep the index key size small for the covering indexes. Add columns using the INCLUDE statement in places where it makes sense. Since a covering index includes all the columns used in a query, it has a tendency to be very wide, increasing the maintenance cost of the covering indexes. You must balance the maintenance cost with the performance gain that the covering index brings. If the number of bytes from all the columns in the index is small compared to the number of bytes in a single data row of that table and you are certain the query taking advantage of the covered index will be executed frequently, then it may be beneficial to use a covering index.

Tip Covering indexes can also help resolve blocking and deadlocks, as you will see in Chapters 20 and 21.

Before building a lot of covering indexes, consider how SQL Server can effectively and automatically create covering indexes for queries on the fly using index intersection.

Index Intersections

If a table has multiple indexes, then SQL Server can use multiple indexes to execute a query. SQL Server can take advantage of multiple indexes, selecting small subsets of data based on each index and then performing an intersection of the two subsets (that is, returning only those rows that meet all the criteria). SQL Server can exploit multiple indexes on a table and then employ a join algorithm to obtain the *index intersection* between the two subsets.

In the following SELECT statement, for the WHERE clause columns, the table has a nonclustered index on the SalesPersonID column, but it has no index on the OrderDate column:

```
--SELECT * is intentionally used in this query
SELECT soh.*
FROM Sales.SalesOrderHeader AS soh
```

```
WHERE soh.SalesPersonID = 276
      AND soh.OrderDate
      BETWEEN '4/1/2005' AND '7/1/2005';
```

Figure 9-3 shows the execution plan for this query.

Figure 9-3. *Execution plan with no index on the OrderDate column*

As you can see, the optimizer didn't use the nonclustered index on the SalesPersonID column. Since the value of the OrderDate column is also required, the optimizer chose the clustered index to fetch the value of all the referred columns. The I/O and time for retrieving this data was as follows:

```
Table 'SalesOrderHeader'. Scan count 1, logical reads 689
CPU time = 0 ms,  elapsed time = 3 ms.
```

To improve the performance of the query, the OrderDate column can be added to the nonclustered index on the SalesPersonId column or defined as an included column on the same index. But in this real-world scenario, you may have to consider the following while modifying an existing index:

- It may not be permissible to modify an existing index for various reasons.
- The existing nonclustered index key may be already quite wide.
- The cost of other queries using the existing index will be affected by the modification.

In such cases, you can create a new nonclustered index on the OrderDate column.

```
CREATE NONCLUSTERED INDEX IX_Test
ON Sales.SalesOrderHeader (OrderDate);
```

Run your SELECT command again.
Figure 9-4 shows the resultant execution plan of the SELECT statement.

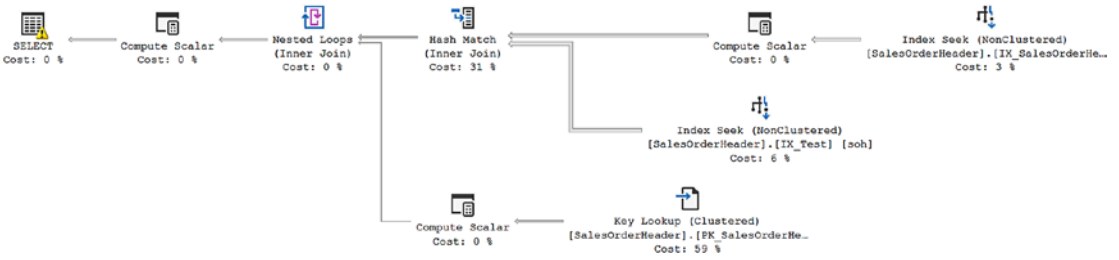


Figure 9-4. *Execution plan with an index on the OrderDate column*

As you can see, SQL Server exploited both the nonclustered indexes as index seeks (rather than scans) and then employed an intersection algorithm to obtain the index intersection of the two subsets. This is represented by the `Hash Join`. It then did a `Key Lookup` from the resulting data set to retrieve the rest of the data not included in the indexes. But, the complexity of the plan suggests that performance might be worse. Checking the statistics I/O and time, you can see that in fact you did get a good performance improvement:

```
Table 'SalesOrderHeader'. Scan count 2, logical reads 10
Table 'Workfile'. Scan count 0, logical reads 0,
Table 'Worktable'. Scan count 0, logical reads 0
CPU time = 0 ms,  elapsed time = 2 ms.
```

The reads dropped from 689 to 10 even though the plan used three different access points within the table and had to create storage for processing the `Hash Join`. The execution time also dropped (3,333µs to 2,279µs in Extended Events). You can also see there are additional operations occurring within the plan, such as the `Key Lookup`, that you might be able to eliminate with further adjustments to the indexes. However, it's worth noting, since you're returning all the columns through the `SELECT *` command, that you can't effectively eliminate the `Key Lookup` by using `INCLUDE` columns, so you may also need to adjust the query.

To improve the performance of a query, SQL Server can use multiple indexes on a table, although it is somewhat rare since it requires good statistics and precise indexes for the specific query. Generally, I try to use smaller, narrower keys on my indexes instead of wide keys. SQL Server can use indexes together frequently, and even when it doesn't, performance is better with narrow indexes. While creating a covering index, identify the existing nonclustered indexes that include most of the columns required by the covering index. You may already have two existing nonclustered indexes that jointly

serve all the columns required by the covering index. If it is possible, rearrange the column order of the existing nonclustered indexes appropriately, allowing the optimizer to consider an index intersection between the two nonclustered indexes.

At times, it is possible that you may have to create a separate nonclustered index for the following reasons:

- Reordering the columns in one of the existing indexes is not allowed.
- Some of the columns required by the covering index may not be included in the existing nonclustered indexes.
- The total number of columns in the existing nonclustered indexes may be more than the number of columns required by the covering index.

In such cases, you can create a nonclustered index on the remaining columns. If the combined column order of the new index and an existing nonclustered index meets the requirement of the covering index, the optimizer may be able to use index intersection. While identifying the columns and their order for the new index, try to maximize their benefit by keeping an eye on other queries, too.

Don't count on frequently getting index intersection to work. It's dependent on the choices made internally by the optimizer. However, there's nothing wrong with striving in this direction when creating your indexes.

Drop the index that was created for the tests.

```
DROP INDEX Sales.SalesOrderHeader.IX_Test;
```

Index Joins

The *index join* is a variation of index intersection, where the covering index technique is applied to the index intersection. If no single index covers a query but multiple indexes together can cover the query, SQL Server can use an index join to satisfy the query fully without going to the base table.

Let's look at this indexing technique at work. Make a slight modification to the query from the "Index Intersections" section like this:

```
SELECT soh.SalesPersonID,
       soh.OrderDate
FROM Sales.SalesOrderHeader AS soh
```

```
WHERE soh.SalesPersonID = 276
      AND soh.OrderDate
      BETWEEN '4/1/2013' AND '7/1/2013';
```

The execution plan for this query is shown in Figure 9-5, and the reads are as follows:

```
Table  'SalesOrderHeader'.  Scan count 1,  logical reads 689
CPU time = 0 ms,  elapsed time = 2 ms. (2345 us)
```

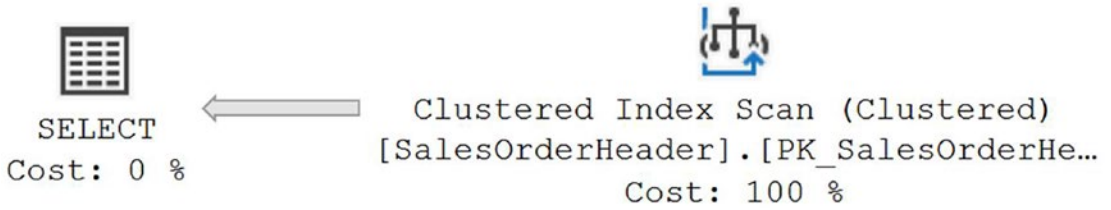


Figure 9-5. *Execution plan with no index join*

As shown in Figure 9-5, the optimizer didn't use the existing nonclustered index on the `SalesPersonID` column. Since the query requires the value of the `OrderDate` column also, the optimizer selected the clustered index to retrieve values for all the columns referred to in the query. If an index is created on the `OrderDate` column like this:

```
CREATE NONCLUSTERED INDEX IX_Test
ON Sales.SalesOrderHeader (OrderDate ASC);
```

and the query is rerun, then Figure 9-6 shows the result, and you can see the reads here:

```
Table 'Workfile'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
Table 'SalesOrderHeader'. Scan count 2, logical reads 10
CPU time = 0 ms,  elapsed time = 1 ms (1657 us).
```

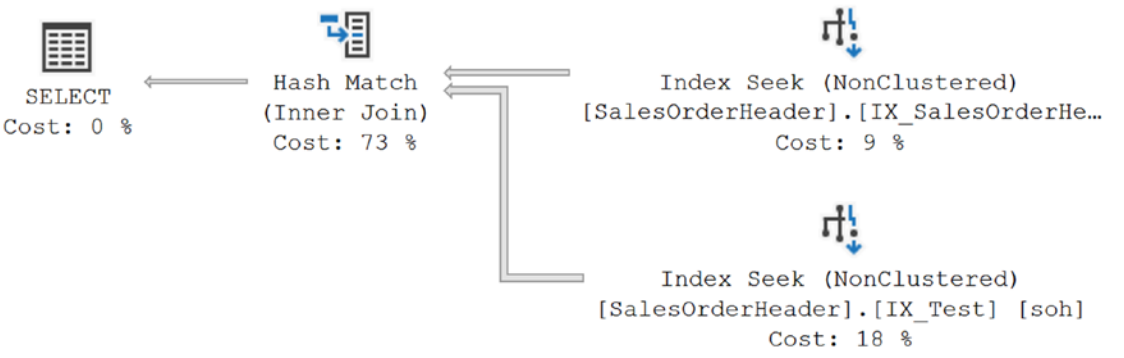


Figure 9-6. *Execution plan with an index join*

The combination of the two indexes acts like a covering index, reducing the reads against the table from 689 to 10 because it's using two Index Seek operations joined together instead of a clustered index scan.

But what if the WHERE clause didn't result in both indexes being used? Instead, you know that both indexes exist and that a seek against each would work like the previous query, so you choose to use an index hint.

```
SELECT soh.SalesPersonID,
       soh.OrderDate
FROM Sales.SalesOrderHeader AS soh
             WITH (INDEX(IX_Test, IX_SalesOrderHeader_SalesPersonID))
WHERE soh.OrderDate
BETWEEN '4/1/2013' AND '7/1/2013';
```

The results of this new query are shown in Figure 9-7, and the I/O is as follows:

```
Table 'Workfile'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
Table 'SalesOrderHeader'. Scan count 2, logical reads 64
CPU time = 0 ms,  elapsed time = 68 ms.
```

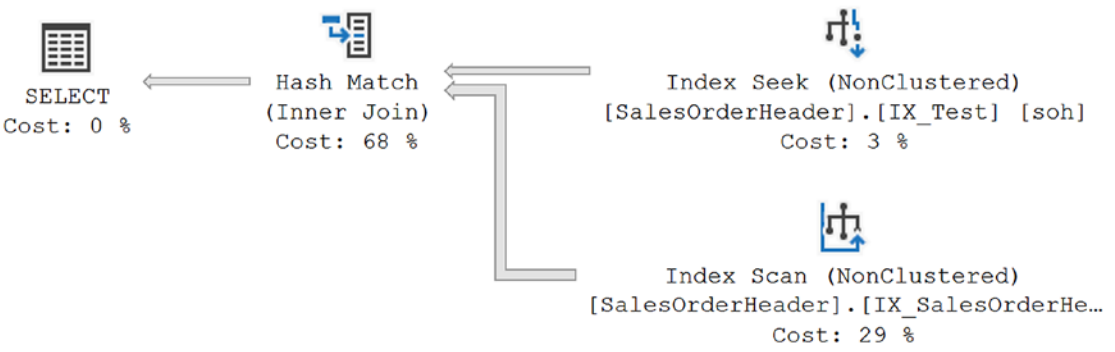


Figure 9-7. *Execution plan with index join through a hint*

The reads have clearly increased as has the execution time. Most of the time, the optimizer makes good choices when it comes to indexes and execution plans. Although query hints are available to allow you to take control from the optimizer, this control can cause as many problems as it solves. In attempting to force an index join as a performance benefit, instead the forced selection of indexes slowed down the execution of the query.

Remove the test index before continuing.

```
DROP INDEX Sales.SalesOrderHeader.IX_Test;
```

Note While generating a query execution plan, the SQL Server optimizer goes through the optimization phases not only to determine the type of index and join strategy to be used but also to evaluate the advanced indexing techniques such as index intersection and index join. Therefore, in some cases, instead of creating wide covering indexes, consider creating multiple narrow indexes. SQL Server can use them together to serve as a covering index yet use them separately where required. But you will need to test to be sure which works better in your situation—wider indexes or index intersections and joins.

Filtered Indexes

A filtered index is a nonclustered index that uses a filter, basically a `WHERE` clause, to ideally create a highly selective set of keys against a column or columns that may not have good selectivity otherwise. For example, a column with a large number of `NULL` values may be stored as a sparse column to reduce the overhead of those `NULL` values. Adding a filtered index using the column will allow you to have an index available on the data that is not `NULL`. The best way to understand this is to see it in action.

The `Sales.SalesOrderHeader` table has more than 30,000 rows. Of those rows, 27,000+ have a `null` value in the `PurchaseOrderNumber` column and the `SalesPersonId` column. If you wanted to get a simple list of purchase order numbers, the query might look like this:

```
SELECT soh.PurchaseOrderNumber,
       soh.OrderDate,
       soh.ShipDate,
       soh.SalesPersonID
FROM Sales.SalesOrderHeader AS soh
WHERE PurchaseOrderNumber LIKE 'PO5%'
      AND soh.SalesPersonID IS NOT NULL;
```

Running the query results in, as you might expect, a clustered index scan, and the following I/O and execution time, as shown in Figure 9-8:

```
Table 'SalesOrderHeader'. Scan count 1, logical reads 689
CPU time = 0 ms,  elapsed time = 52 ms.
```

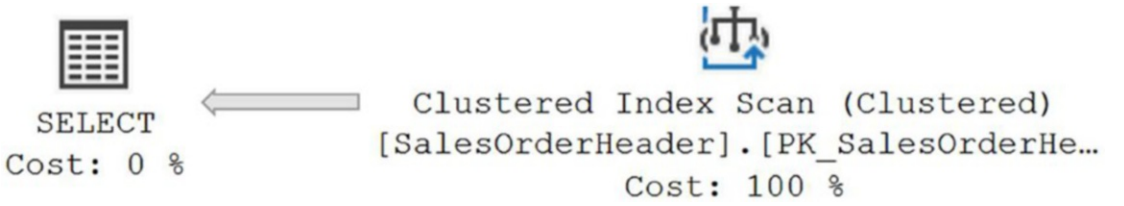


Figure 9-8. *Execution plan without an index*

To fix this, it is possible to create an index and include some of the columns from the query to make this a covering index.

```
CREATE NONCLUSTERED INDEX IX_Test
ON Sales.SalesOrderHeader
(
    PurchaseOrderNumber,
    SalesPersonID
)
INCLUDE
(
    OrderDate,
    ShipDate
);
```

When you rerun the query, the performance improvement is fairly radical (see Figure 9-9 and the I/O and time in the following result).

```
Table 'SalesOrderHeader'. Scan count 1, logical reads 5
CPU time = 0 ms,  elapsed time = 40 ms.
```

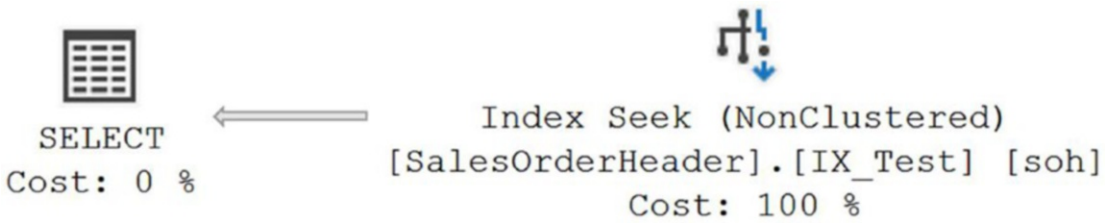


Figure 9-9. *Execution plan with a covering index*

As you can see, the covering index dropped the reads from 689 to 5 and the time from 52ms to 40ms. Normally, this would be considered a decent improvement and may be adequate for the system. Assume for a moment that this query has to be called frequently. Now, every bit of speed you can wring from it will pay dividends. Knowing that so much of the data in the indexed columns is `null`, you can adjust the index so that it filters out the `null` values, which aren't used by the index anyway, reducing the size of the tree and therefore the amount of searching required.

```
CREATE NONCLUSTERED INDEX IX_Test
ON Sales.SalesOrderHeader
(
    PurchaseOrderNumber,
    SalesPersonID
)
INCLUDE
(
    OrderDate,
    ShipDate
)
WHERE PurchaseOrderNumber IS NOT NULL
      AND SalesPersonID IS NOT NULL
WITH (DROP_EXISTING = ON);
```

The final run of the query resulted in the following performance metrics:

```
Table 'SalesOrderHeader'. Scan count 1, logical reads 4
CPU time = 0 ms,    elapsed time = 38 ms.
```

The execution plan is going to look identical, with an `Index Seek`. To see the differences between the plan for the covering index and the plan for the filtered, covering index, we can use SSMS to compare the plans. Save the first plan as a file (right-click the plan and select Save Execution Plan As), and then, from the second plan, right-click inside the plan and select Compare Plan. You'll then see something similar to Figure 9-10.

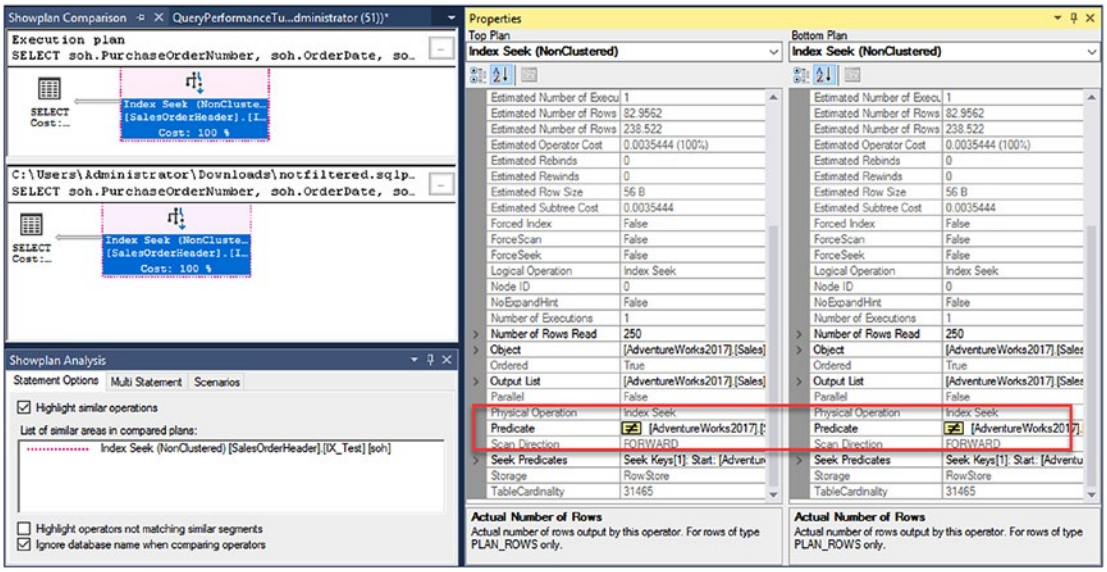


Figure 9-10. *Comparison of the two plans*

There are almost no direct indicators of differences in the execution plans. In the properties to the right, I've highlighted the one big difference. While the queries were identical, because of the index that filters out all null values, the predicate gets changed to remove `IS NOT NULL` because it's no longer needed. This is part of a process within the optimizer called *simplification*.

Although in terms of sheer numbers reducing the reads from 5 to 4 isn't much, it is a 20 percent reduction in the I/O cost of the query, and if this query were running hundreds or even thousands of times in a minute, like some queries do, that 20 percent reduction would be a great payoff indeed. Another visible evidence of the payoff is in the execution time, which dropped again from 40ms to 38ms.

Filtered indexes improve performance in many ways.

- Improving the efficiency of queries by reducing the size of the index
- Reducing storage costs by making smaller indexes
- Cutting down on the costs of index maintenance because of the reduced size

But, everything does come with a cost. You may see issues with parameterized queries not matching the filtered index, therefore preventing its use. Statistics are not updated based on the filtering criteria but rather on the entire table just like a regular index. Like with any of the suggestions in this book, test in your environment to ensure that filtered indexes are helpful.

One of the first places suggested for their use is just like the previous example, eliminating `NULL` values from the index. You can also isolate frequently accessed sets of data with a special index so that the queries against that data perform much faster. You can use the `WHERE` clause to filter data in a fashion similar to creating an indexed view (covered in more detail in the "Indexed Views" section) without the data maintenance headaches associated with indexed views by creating a filtered index that is a covering index, just like the earlier example.

Filtered indexes require a specific set of ANSI settings when they are accessed or created.

- ON: `ANSI_NULLS`, `ANSI_PADDING`, `ANSI_WARNINGS`, `ARITHABORT`, `CONCAT_NULL_YIELDS_NULL`, `QUOTED_IDENTIFIER`
- OFF: `NUMERIC_ROUNDABORT`

When completed, drop the testing index.

```
DROP INDEX Sales.SalesOrderHeader.IX_Test;
```

Indexed Views

A database view in SQL Server does not store any data. A view is simply a `SELECT` statement that is stored. You create a view using the `CREATE VIEW` statement. You can write queries against a view exactly as if it were a table. When a view gets queried, the optimizer receives the full definition of the `SELECT` statement and uses that as the basis for optimizing the query against the view. Through the optimization process, some or all of the definition of the `SELECT` statement may be used to satisfy the query against the view. What degree of simplification occurs here is determined by a combination of the `SELECT` statement itself and the query against that `SELECT` statement.

A database view can be materialized on the disk by creating a unique clustered index on the view. Such a view is referred to as an *indexed view* or a *materialized view*. After a unique clustered index is created on the view, the view's result set is materialized immediately and persisted in physical storage in the database, saving the overhead of

performing costly operations during query execution. After the view is materialized, multiple nonclustered indexes can be created on the indexed view. Effectively, this turns a view (again, just a query) into a real table with defined storage.

Benefit

You can use an indexed view to increase the performance of a query in the following ways:

- Aggregations can be precomputed and stored in the indexed view to minimize expensive computations during query execution.
- Tables can be prejoined, and the resulting data set can be materialized.
- Combinations of joins or aggregations can be materialized.

Overhead

Indexed views can produce major overhead on an OLTP database. Some of the overheads of indexed views are as follows:

- Any change in the base tables has to be reflected in the indexed view by executing the view's `SELECT` statement.
- Any changes to a base table on which an indexed view is defined may initiate one or more changes in the nonclustered indexes of the indexed view. The clustered index will also have to be changed if the clustering key is updated.
- The indexed view adds to the ongoing maintenance overhead of the database.
- Additional storage is required in the database.

The restrictions on creating an indexed view include the following:

- The first index on the view must be a unique clustered index.
- Nonclustered indexes on an indexed view can be created only after the unique clustered index is created.
- The view definition must be *deterministic*—that is, it is able to return only one possible result for a given query. (A list of deterministic and nondeterministic functions is provided in SQL Server Books Online.)

- The indexed view must reference only base tables in the same database, not other views.
- The indexed view may contain float columns. However, such columns cannot be included in the clustered index key.
- The indexed view must be schema bound to the tables referred to in the view to prevent modifications of the table schema (frequently a major problem).
- There are several restrictions on the syntax of the view definition. (A list of the syntax limitations on the view definition is provided in SQL Server Books Online.)
- The list of `SET` options that must be fixed are as follows:
  - `ON`: `ARITHABORT`, `CONCAT_NULL_YIELDS_NULL`, `QUOTED_IDENTIFIER`, `ANSI_NULLS`, `ANSI_ PADDING`, and `ANSI_WARNING`
  - `OFF`: `NUMERIC_ROUNDABORT`

Note If the query connection settings don't match these ANSI standard settings, you may see errors on the insert/update/delete of tables that are used within the indexed view.

Usage Scenarios

Reporting systems benefit the most from indexed views. OLTP systems with frequent writes may not be able to take advantage of the indexed views because of the increased maintenance cost associated with updating both the view and the underlying base tables within a single transaction. The net performance improvement provided by an indexed view is the difference between the total query execution savings offered by the view and the cost of storing and maintaining the view.

If you are using the Enterprise edition of SQL Server, an indexed view need not be referenced explicitly in the query for the query optimizer to use it during query execution. This allows existing applications to benefit from the newly created indexed views without changing those applications. Otherwise, you would need to directly reference it within your T-SQL code on editions of SQL Server other than Enterprise. The

query optimizer considers indexed views only for queries with nontrivial cost. You may also find that the new columnstore index will work better for you than indexed views, especially when you're running aggregation or analysis queries against the data. I'll cover the columnstore index later in this chapter.

Let's see how indexed views work with the following example. Consider the following three queries:

```
SELECT  p.[Name] AS ProductName,
        SUM(pod.OrderQty) AS OrderOty,
        SUM(pod.ReceivedQty) AS ReceivedOty,
        SUM(pod.RejectedQty) AS RejectedOty
FROM    Purchasing.PurchaseOrderDetail AS pod
        JOIN Production.Product AS p
        ON p.ProductID = pod.ProductID
GROUP BY p.[Name];

SELECT  p.[Name] AS ProductName,
        SUM(pod.OrderQty) AS OrderOty,
        SUM(pod.ReceivedQty) AS ReceivedOty,
        SUM(pod.RejectedQty) AS RejectedOty
FROM    Purchasing.PurchaseOrderDetail AS pod
        JOIN Production.Product AS p
        ON p.ProductID = pod.ProductID
GROUP BY p.[Name]
HAVING  (SUM(pod.RejectedQty) / SUM(pod.ReceivedQty)) > .08;

SELECT  p.[Name] AS ProductName,
        SUM(pod.OrderQty) AS OrderQty,
        SUM(pod.ReceivedQty) AS ReceivedQty,
        SUM(pod.RejectedQty) AS RejectedQty
FROM    Purchasing.PurchaseOrderDetail AS pod
        JOIN Production.Product AS p
        ON p.ProductID = pod.ProductID
WHERE   p.[Name] LIKE 'Chain%'
GROUP BY p.[Name];
```

All three queries use the aggregation function SUM on columns of the PurchaseOrderDetail table. Therefore, you can create an indexed view to precompute these aggregations and minimize the cost of these complex computations during query execution.

Here are the number of logical reads performed by these queries to access the appropriate tables:

```
Table 'Workfile'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
Table 'Product'. Scan count 1, logical reads 6
Table 'PurchaseOrderDetail'. Scan count 1, logical reads 66
CPU time = 0 ms,  elapsed time = 31 ms.

Table 'Workfile'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
Table 'Product'. Scan count 1, logical reads 6
Table 'PurchaseOrderDetail'. Scan count 1, logical reads 66
CPU time = 0 ms,  elapsed time = 16 ms.

Table 'PurchaseOrderDetail'. Scan count 5, logical reads 894
Table 'Product'. Scan count 1, logical reads 2
CPU time = 0 ms,  elapsed time = 1 ms.
```

I'll use the following script to create an indexed view to precompute the costly computations and join the tables:

```
CREATE OR ALTER VIEW Purchasing.IndexedView
WITH SCHEMABINDING
AS
SELECT pod.ProductID,
       SUM(pod.OrderQty) AS OrderQty,
       SUM(pod.ReceivedQty) AS ReceivedQty,
       SUM(pod.RejectedQty) AS RejectedQty,
       COUNT_BIG(*) AS Count
FROM Purchasing.PurchaseOrderDetail AS pod
GROUP BY pod.ProductID;
GO
```

```
CREATE UNIQUE CLUSTERED INDEX iv
ON Purchasing.IndexedView (ProductID);
GO
```

Certain constructs such as AVG are disallowed. (For the complete list of disallowed constructs, refer to SQL Server Books Online.) If aggregates are included in the view, like in this one, you must include COUNT_BIG by default.

The indexed view materializes the output of the aggregate functions on the disk. This eliminates the need for computing the aggregate functions during the execution of a query interested in the aggregate outputs. For example, the third query requests the sum of ReceivedQty and RejectedQty for certain products from the PurchaseOrderDetail table. Because these values are materialized in the indexed view for every product in the PurchaseOrderDetail table, you can fetch these preaggregated values using the following SELECT statement on the indexed view:

```
SELECT  iv.ProductID,
        iv.ReceivedQty,
        iv.RejectedQty
FROM    Purchasing.IndexedView AS iv;
```

As shown in the execution plan in Figure 9-11, the SELECT statement retrieves the values directly from the indexed view without accessing the base table (PurchaseOrderDetail).

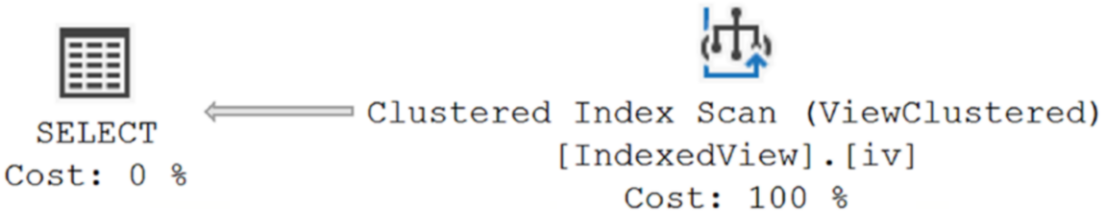


Figure 9-11. *Execution plan with an indexed view*

The indexed view benefits not only the queries based on the view directly but also other queries that may be interested in the materialized data. For example, with the indexed view in place, the three queries on `PurchaseOrderDetail` benefit without being rewritten (see the execution plan in Figure 9-12 for the execution plan from the first query), and the number of logical reads decreases, as shown here:

```
Table 'Product'. Scan count 1, logical reads 13
Table 'IndexedView'. Scan count 1, logical reads 4
CPU time = 0 ms,  elapsed time = 53 ms.

Table 'Product'. Scan count 1, logical reads 13
Table 'IndexedView'. Scan count 1, logical reads 4
CPU time = 0 ms,  elapsed time = 1 ms.

Table 'IndexedView'. Scan count 0, logical reads 10
Table 'Product'. Scan count 1, logical reads 2
CPU time = 0 ms,  elapsed time = 0 ms. (214 us)
```

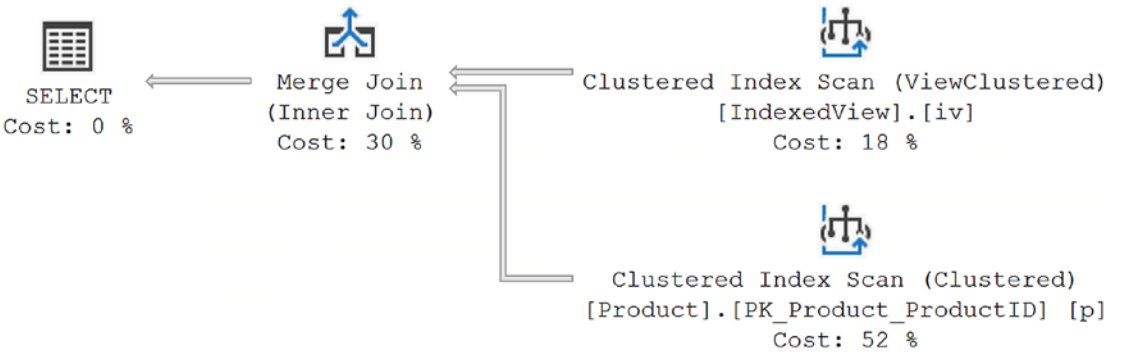


Figure 9-12. *Execution plan with the indexed view automatically used*

Even though the queries are not modified to refer to the new indexed view, the optimizer still uses the indexed view to improve performance. Thus, even existing queries in the database application can benefit from new indexed views without any modifications to the queries. If you do need different aggregations than what the indexed view offers, you'll be out of luck. Here again the columnstore index shines.

Make sure to clean up.

```
DROP VIEW Purchasing.IndexedView;
```

Index Compression

Data and index compression were introduced in SQL Server 2008 (available in the Enterprise and Developer editions, currently in all editions). *Compressing* an index means getting more key information onto a single page. This can lead to significant performance improvements because fewer pages and fewer index levels are needed to store the index. There will be overhead in the CPU as the key values in the index are compressed and decompressed, so this may not be a solution for all indexes. Memory benefits also because the compressed pages are stored in memory in a compressed state.

By default, an index will be not be compressed. You have to explicitly call for the index to be compressed when you create the index. There are two types of compression: row- and page-level compression. *Row-level compression* identifies columns that can be compressed (for details, look in Books Online) and compresses the storage of that column and does this for every row. *Page-level compression* is actually using row-level compression and then adding additional compression on top to reduce storage size for the nonrow elements stored on a page. Nonleaf pages in an index receive no compression under the page type. To see index compression in action, consider the following index:

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

This index was created earlier in the chapter. If you were to re-create it as defined here, this creates a row type of compression on an index with the same two columns as the first test index `IX_Test`.

```
CREATE NONCLUSTERED INDEX IX_Comp_Test
ON Person.Address
(
    City,
    PostalCode
)
WITH (DATA_COMPRESSION = ROW);
```

Create one more index.

```
CREATE NONCLUSTERED INDEX IX_Comp_Page_Test
ON Person.Address
(
    City,
    PostalCode
)
WITH (DATA_COMPRESSION = PAGE);
```

To examine the indexes being stored, modify the original query against `sys.dm_db_index_physical_stats` to add another column, `compressed_page_count`.

```
SELECT i.name,
       i.type_desc,
       s.page_count,
       s.record_count,
       s.index_level,
       s.compressed_page_count
FROM sys.indexes AS i
    JOIN sys.dm_db_index_physical_stats(DB_ID(N'AdventureWorks2017'),
                                        OBJECT_ID(N'Person.Address'),
                                        NULL,
                                        NULL,
                                        'DETAILED') AS s
        ON i.index_id = s.index_id
WHERE i.object_id = OBJECT_ID(N'Person.Address');
```

Running the query, you get the results in Figure 9-13.

	name	type_desc	page_count	record_count	index_level	compressed_page_count
12	IX_Comp_Test	NONCLUSTERED	63	19614	0	0
13	IX_Comp_Test	NONCLUSTERED	1	63	1	0
14	IX_Comp_Page_Test	NONCLUSTERED	25	19614	0	25
15	IX_Comp_Page_Test	NONCLUSTERED	1	25	1	0
16	IX_Test	NONCLUSTERED	106	19614	0	0
17	IX_Test	NONCLUSTERED	1	106	1	0

Figure 9-13. *sys.dm_db_index_physical_stats output about compressed indexes*

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.

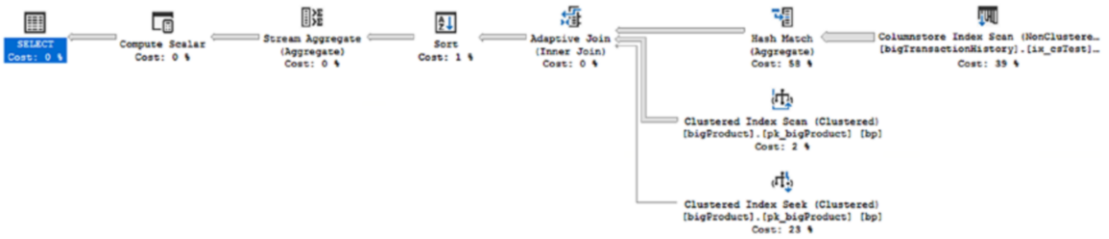


Figure 9-15. *The columnstore index is used instead of the clustered index*

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.

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`.

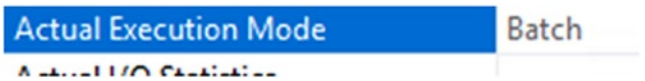


Figure 9-17. *Actual execution mode*

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.

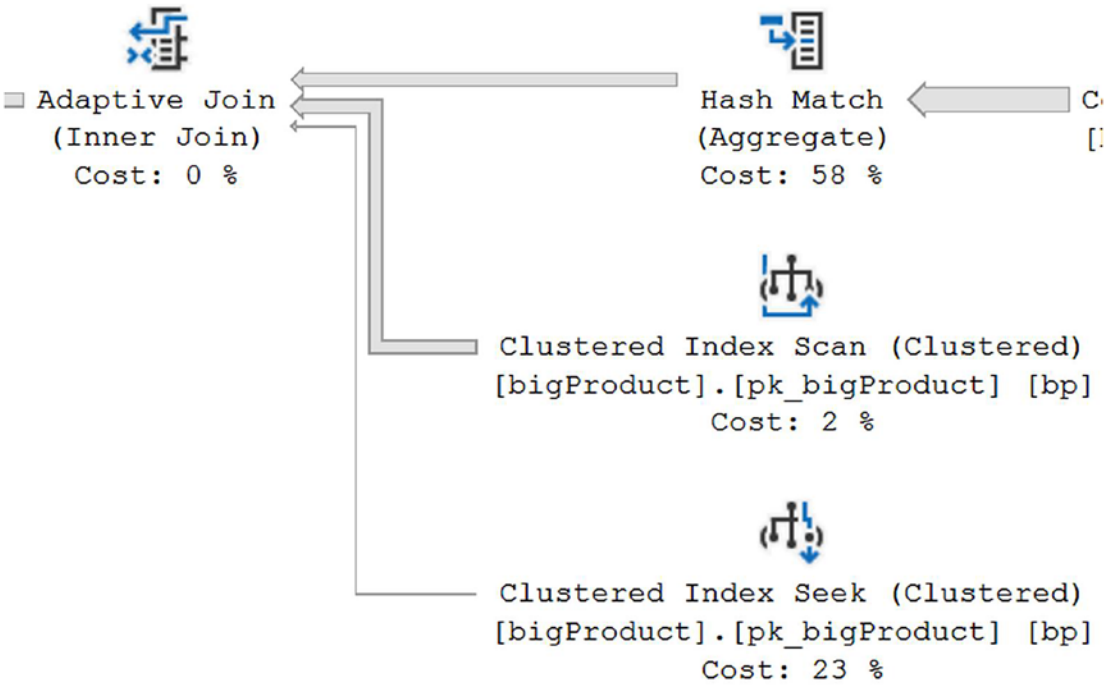


Figure 9-18. *Adaptive join and its attendant behavior*

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.

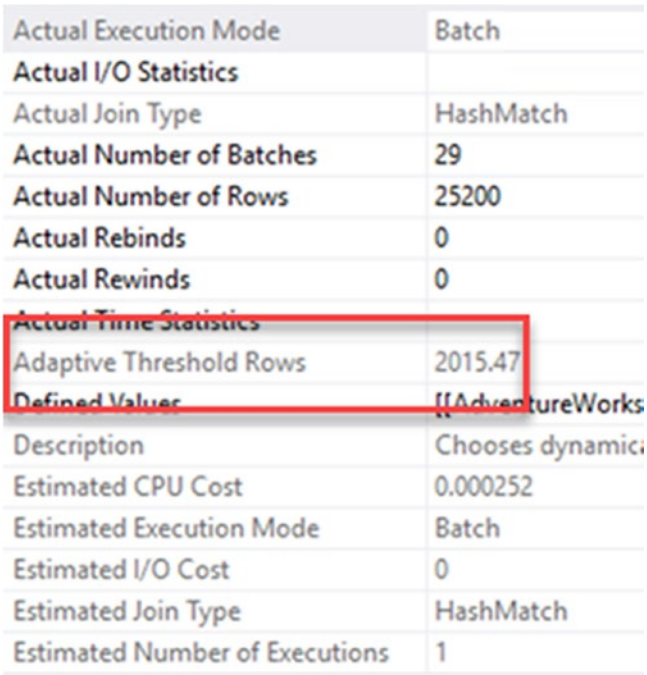


Figure 9-19. *A subset of the adaptive join properties*

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.

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.

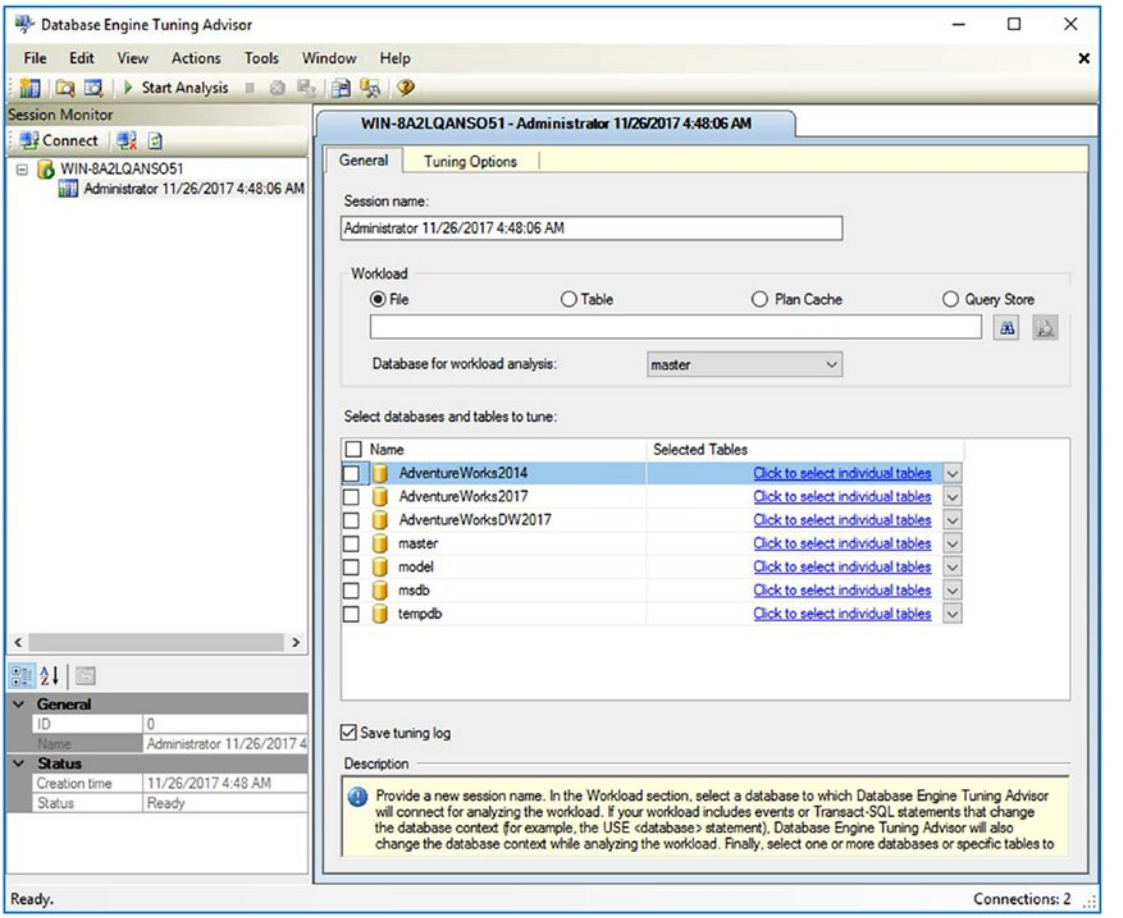
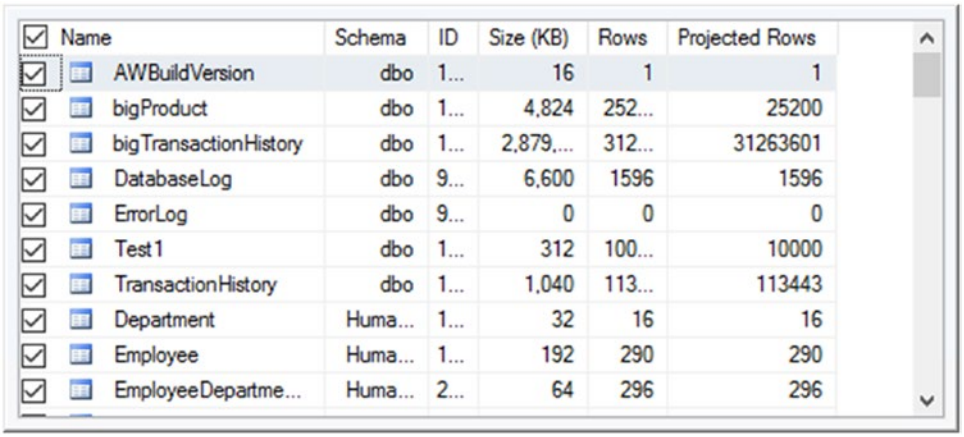


Figure 10-1. *Selecting the server and database in the Database Engine Tuning Advisor*

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.

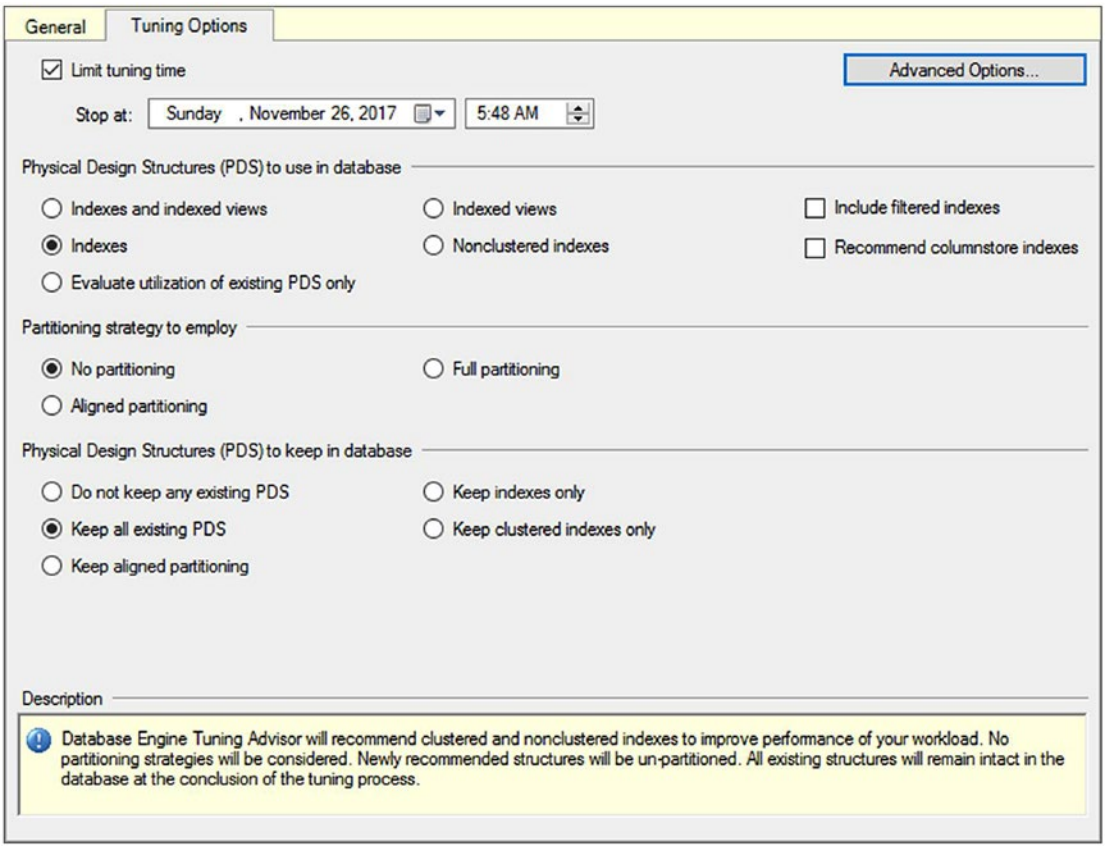


Figure 10-3. *Defining options in the Database Engine Tuning Advisor*

You define the length of time you want the Database Engine Tuning Advisor to run by selecting Limit Tuning Time and then defining a date and time for the tuning to stop. The longer the Database Engine Tuning Advisor runs, the better recommendations it should make. You pick the type of physical design structures to be considered for creation by the Database Engine Tuning Advisor, and you can also set the partitioning strategy so that the Tuning Advisor knows whether it should consider partitioning the tables and indexes as part of the analysis. Just remember, partitioning is foremost a data management tool, not a performance tuning mechanism. Partitioning may not necessarily be a desirable outcome if your data and structures don't warrant it. Finally, you can define the physical design structures that you want left alone within the database. Changing these options will narrow or widen the choices that the Database Engine Tuning Advisor can make to improve performance. You can optionally include filtered indexes, and the Database Engine Tuning Advisor can recommend columnstore indexes.

You can click the Advanced Options button to see even more options, as shown in Figure 10-4.

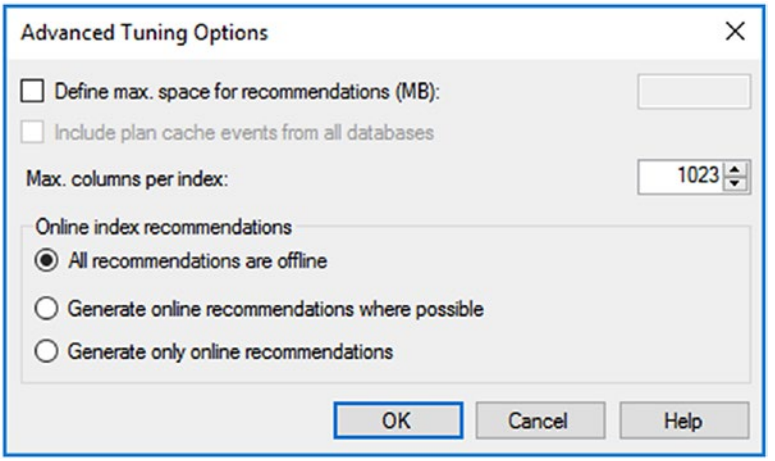


Figure 10-4. *Advanced Tuning Options dialog box*

This dialog box allows you to limit the space of the recommendations and the number of columns that can be included in an index. You decide whether you want to include plan cache events from every database on the system. Finally, you can define whether the new indexes or changes in indexes are done as an online or offline index operation.

Once you've appropriately defined all of these settings, you can start the Database Engine Tuning Advisor by clicking the Start Analysis button. The sessions created are kept in the `msdb` database for any server instance that you run the Database Engine Tuning Advisor against. It displays details about what is being analyzed and the progress that was made, which you can see in Figure 10-5.

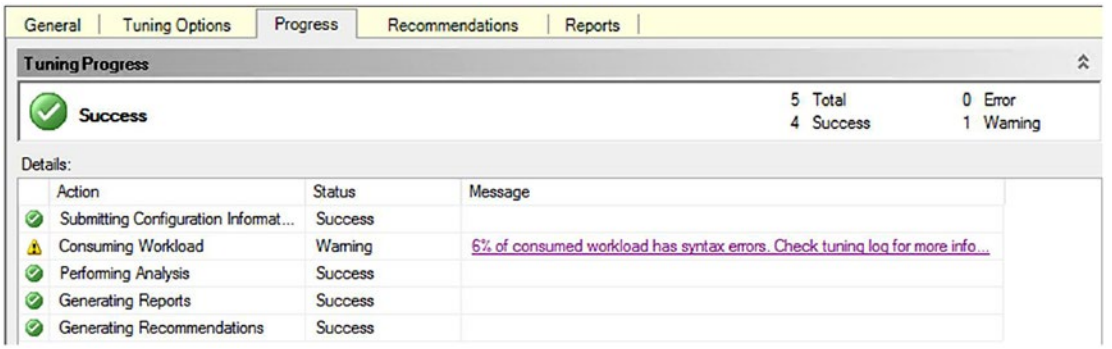


Figure 10-5. *Tuning progress*

You'll see more detailed examples of the progress displayed in the example analysis in the next session.

After the analysis completes, you'll get a list of recommendations (visible in Figure 10-6), and a number of reports become available. Table 10-1 describes the reports.

Table 10-1. *Database Engine Tuning Advisor Reports*

Report Name	Report Description
Column Access	Lists the columns and tables referenced in the workload
Database Access	Lists each database referenced in the workload and percentage of workload statements for each database
Event Frequency	Lists all events in the workload ordered by frequency of occurrence
Index Detail (Current)	Defines indexes and their properties referenced by the workload
Index Detail (Recommended)	Is the same as the Index Detail (Current) report but shows the information about the indexes recommended by the Database Engine Tuning Advisor
Index Usage (Current)	Lists the indexes and the percentage of their use referenced by the workload
Index Usage (Recommended)	Is the same as the Index Usage (Current) report but from the recommended indexes
Statement Cost	Lists the performance improvements for each statement if the recommendations are implemented
Statement Cost Range	Breaks down the cost improvements by percentiles to show how much benefit you can achieve for any given set of changes; these costs are estimated values provided by the optimizer
Statement Detail	Lists the statements in the workload, their cost, and the reduced cost if the recommendations are implemented
Statement-to-Index Relationship	Lists the indexes referenced by individual statements; current and recommended versions of the report are available
Table Access	Lists the tables referenced by the workload
View-to-Table Relationship	Lists the tables referenced by materialized views
Workload Analysis	Gives details about the workload, including the number of statements, the number of statements whose cost is decreased, and the number where the cost remains the same

Database Engine Tuning Advisor Examples

The best way to learn how to use the Database Engine Tuning Advisor is to use it. It's not a terribly difficult tool to master, so I recommend opening it and getting started.

Tuning a Query

You can use the Database Engine Tuning Advisor to recommend indexes for a complete database by using a workload that fairly represents all SQL activities. You can also use it to recommend indexes for a set of problematic queries.

To learn how you can use the Database Engine Tuning Advisor to get index recommendations on a set of problematic queries, say you have a simple query that is called rather frequently. Because of the frequency, you want a quick turnaround for some tuning. This is the query:

```
SELECT soh.DueDate,
       soh.CustomerID,
       soh.Status
FROM Sales.SalesOrderHeader AS soh
WHERE soh.DueDate
BETWEEN '1/1/2008' AND '2/1/2008';
```

To analyze the query, right-click it in the query window and select Analyze Query in the Database Engine Tuning Advisor. The advisor opens with a window where you can change the session name to something meaningful. In this case, I chose Report Query Round 1 – 1/16/2014. The database and tables don't need to be edited. The first tab, General, will look like Figure 10-6 when you're done.

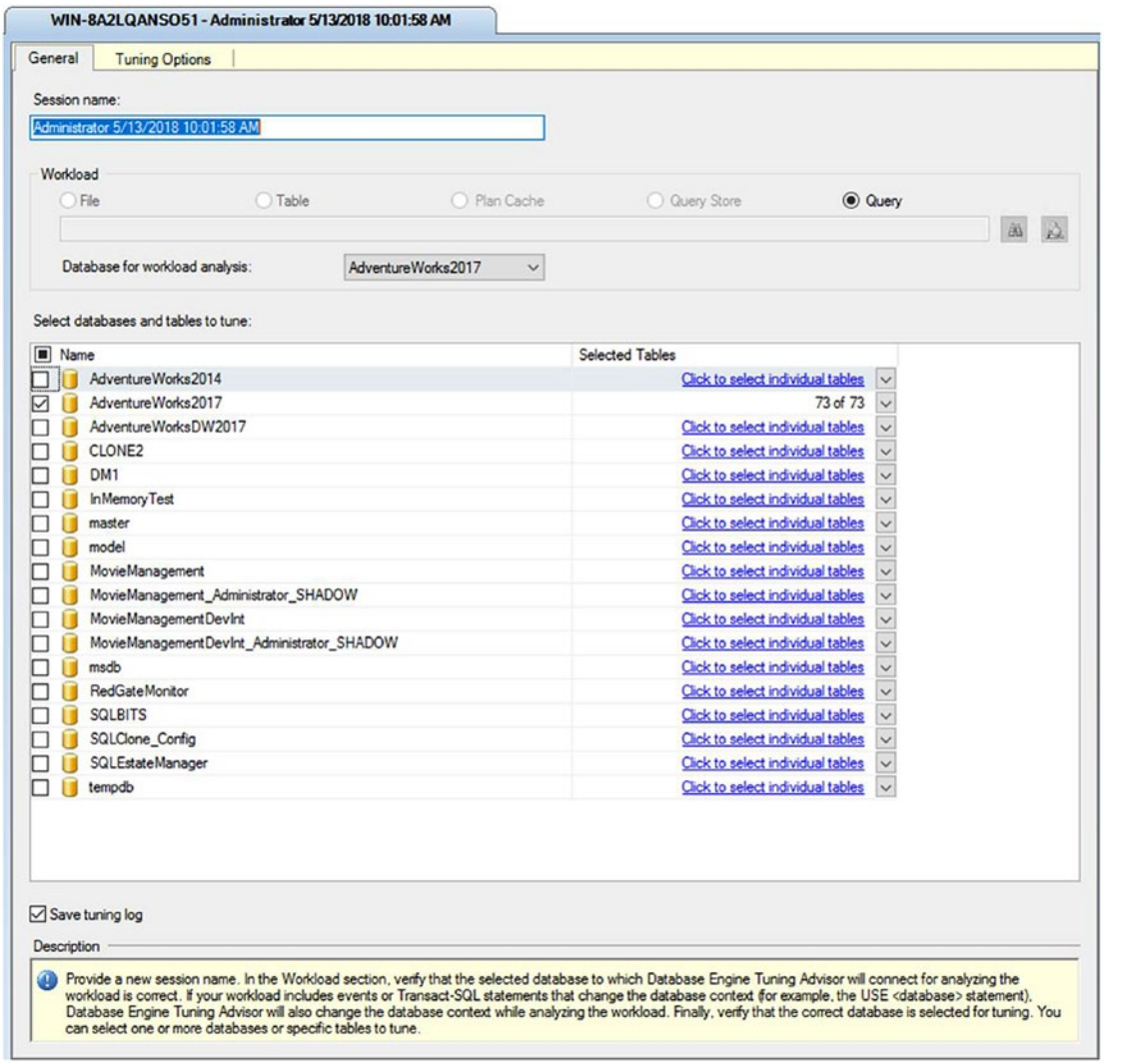


Figure 10-6. *Query tuning general settings*

Because this query is important and tuning it is extremely critical to the business, I'm going to change some settings on the Tuning Options tab to try to maximize the possible suggestions. For the purposes of the example, I'm going to let the Database Engine Tuning Advisor run for the default of one hour, but for bigger loads or more complex queries, you might want to consider giving the system more time. I'm going to select the Include Filtered Indexes check box so that if a filtered index will help, it can be considered. I'm also going to let it recommend columnstore indexes. Finally, I'm going

to allow the Database Engine Tuning Advisor to come up with structural changes if it can find any that will help by switching from Keep All Existing PDS to Do Not Keep Any Existing PDS. Once completed, the Tuning Options tab will look like Figure 10-7.

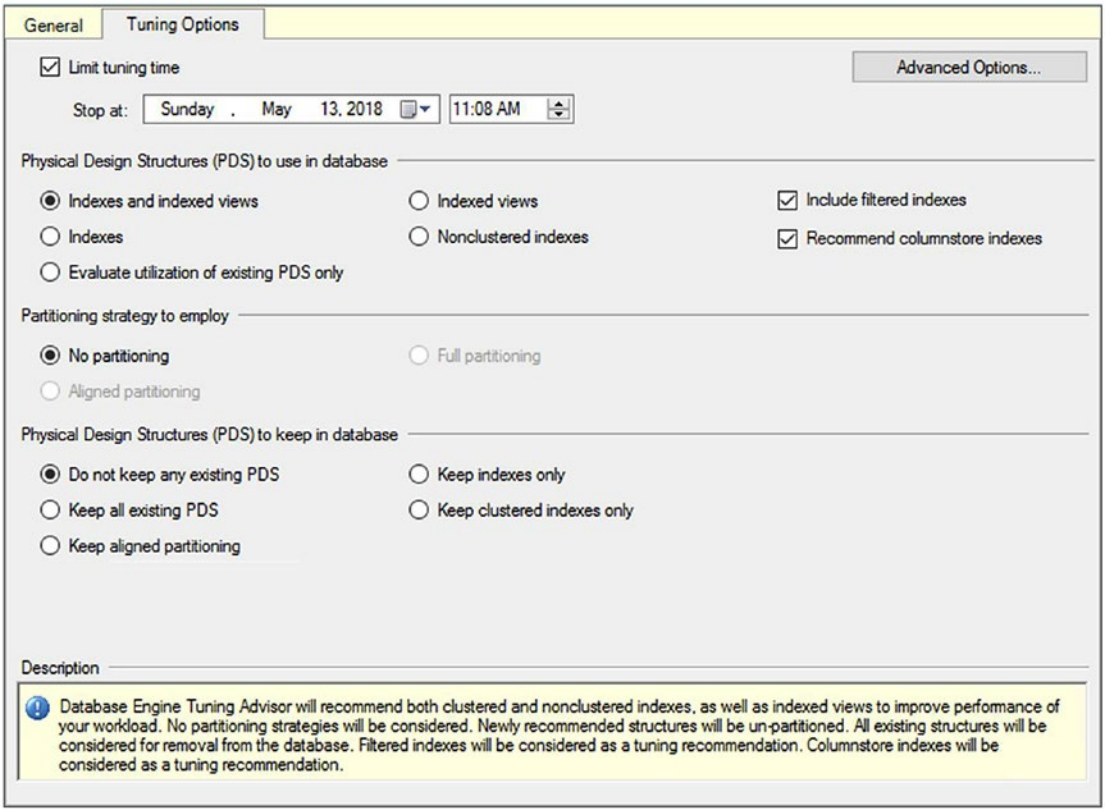


Figure 10-7. *Tuning Options tab adjusted*

Notice that the description at the bottom of the screen changes as you change the definitions in the selections made above. After starting the analysis, the progress screen should appear. Although the settings were for one hour of evaluations, it took only about a minute for the DTA to evaluate this query. The initial recommendations were not a good set of choices. As you can see in Figure 10-8, the Database Engine Tuning Advisor has recommended dropping a huge swath of indexes in the database. This is not the type of recommendation you want when running the tool.