4. Run your tests or run your system for a period of time that ensures you have covered the majority of queries within the system. This time will vary depending on your needs.
5. Change the compatibility mode.
6. Run the report Regressed Queries. This report will find queries that have suddenly started running slower than they had previously.
7. Investigate those queries. If it's obvious that the query plan has changed and is the cause of the change in performance, then pick a plan from prior to the change and use plan forcing to make that plan the one used by SQL Server.
8. Where necessary, take the time to rewrite the queries or restructure the system to ensure that the query can, on its own, compile a plan that performs well with the system.

This approach won't prevent all problems. You still must test your system. However, using the Query Store will provide you with mechanisms for dealing with internal changes within SQL Server that affect your query plans and subsequently your performance. You can use similar processes for applying a Cumulative Update or Service Pack too. You can also deal with regressions by using the Database Scoped Configuration settings, available in SQL Server 2016 SP1 and up, to enable the `LEGACY_CARDINALITY_ESTIMATION`, or you can add that as a hint. These are options in addition to, or instead of, using plan forcing. You can also just revert to the old compatibility mode, but that takes away a lot of functionality.

Summary

The Query Store adds to your abilities to identify poorly performing queries. While the functionality of the Query Store is wonderful, it's not going to completely replace any of the tools most people are already comfortable with using. It's not as granular as Extended Events. It doesn't have some of the immediacy of querying the plan cache. That said, the Query Store adds to both these methods by including additional information such as the standard deviation for values and holding all execution plans, even the ones that have been removed or replaced in cache. Further, the Query Store

adds the ability to perform extremely simple plan forcing that can help not only with issues around parameters or other behaviors but with plan regressions caused by upgrades from Microsoft. All of this combines to make the Query Store an incredibly useful addition to the query tuning toolkit.

Frequently, you will rely on nonclustered indexes to improve the performance of a SQL workload. This assumes you've already assigned a clustered index to your tables. Because the performance of a nonclustered index is highly dependent on the cost of the bookmark lookup associated with the nonclustered index, you will see in the next chapter how to analyze and resolve a lookup.

CHAPTER 12

Key Lookups and Solutions

To maximize the benefit from nonclustered indexes, you must minimize the cost of the data retrieval as much as possible. A major overhead associated with nonclustered indexes is the cost of excessive lookups, formerly known as *bookmark lookups*, which are a mechanism to navigate from a nonclustered index row to the corresponding data row in the clustered index or the heap. Therefore, it makes sense to look at the cause of lookups and to evaluate how to avoid this cost.

In this chapter, I cover the following topics:

- The purpose of lookups
- The drawbacks of using lookups
- Analysis of the cause of lookups
- Techniques to resolve lookups

Purpose of Lookups

When an application requests information through a query, the optimizer can use a nonclustered index, if available, on the columns in the `WHERE`, `JOIN`, or `HAVING` clauses to navigate to the data. Of course, it could also scan a heap or a clustered index, but we're assuming here that the predicate values and the key values of the nonclustered index are lined up. If the query refers to columns that are not part of the nonclustered index (either the key columns or the INCLUDE list) being used to retrieve the data, then navigation is required from the index row to the corresponding data row in the table to access these remaining columns.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_12

For example, in the following SELECT statement, if the nonclustered index used by the optimizer doesn't include all the columns, navigation will be required from a nonclustered index row to the data row in the clustered index or heap to retrieve the value of those columns.

```
SELECT p.Name,
       AVG(sod.LineTotal)
FROM Sales.SalesOrderDetail AS sod
    JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID
WHERE sod.ProductID = 776
GROUP BY sod.CarrierTrackingNumber,
         p.Name
HAVING MAX(sod.OrderQty) > 1
ORDER BY MIN(sod.LineTotal);
```

The SalesOrderDetail table has a nonclustered index on the ProductID column. The optimizer can use the index to filter the rows from the table. The table has a clustered index on SalesOrderID and SalesOrderDetailID, so they would be included in the nonclustered index. But since they're not referenced in the query, they won't help the query at all. The other columns (LineTotal, CarrierTrackingNumber, OrderQty, and LineTotal) referred to by the query are not available in the nonclustered index. To fetch the values for those columns, navigation from the nonclustered index row to the corresponding data row through the clustered index is required, and this operation is a key lookup. You can see this in action in Figure 12-1.

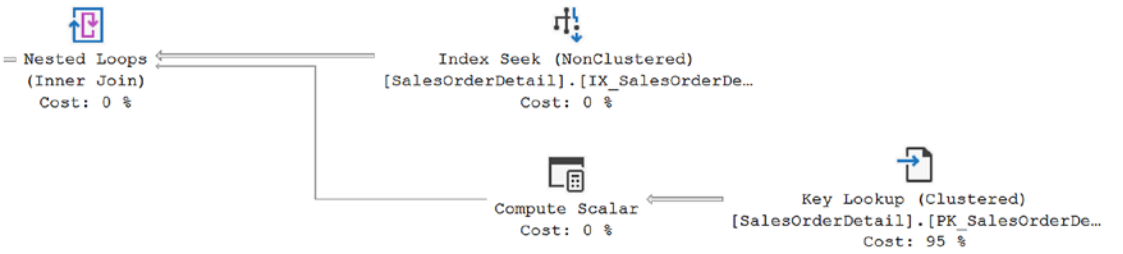


Figure 12-1. *Key lookup in part of a more complicated execution plan*

To better understand how a nonclustered index can cause a lookup, consider the following SELECT statement, which requests only a few rows but all columns because of the wildcard * from the SalesOrderDetail table by using a filter criterion on column ProductID:

```
SELECT  *
FROM Sales.SalesOrderDetail AS sod
WHERE sod.ProductID = 776 ;
```

The optimizer evaluates the WHERE clause and finds that the column ProductID included in the WHERE clause has a nonclustered index on it that filters the number of rows down. Since only a few rows, 228, are requested, retrieving the data through the nonclustered index will be cheaper than scanning the clustered index (containing more than 120,000 rows) to identify the matching rows. The nonclustered index on the column ProductID will help identify the matching rows quickly. The nonclustered index includes the column ProductID and the clustered index columns SalesOrderID and SalesOrderDetailID; all the other columns being requested are not included. Therefore, as you may have guessed, to retrieve the rest of the columns while using the nonclustered index, you require a lookup.

This is shown in the following Extended Events metrics and in the execution plan in Figure 12-2. Look for the Key Lookup (Clustered) operator. That is the lookup in action.

```
Duration: 176ms
Reads: 755
```

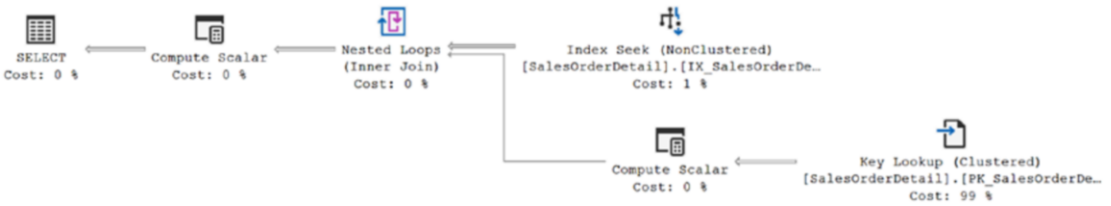


Figure 12-2. *Execution plan with a bookmark lookup*

Drawbacks of Lookups

A lookup requires data page access in addition to index page access. Accessing two sets of pages increases the number of logical reads for the query. Additionally, if the pages are not available in memory, a lookup will probably require a random (or nonsequential) I/O operation on the disk to jump from the index page to the data page as well as requiring the necessary CPU power to marshal this data and perform the necessary operations. This is because, for a large table, the index page and the corresponding data page usually won't be directly next to each other on the disk.

The increased logical reads and costly physical reads (if required) make the data retrieval operation of the lookup quite costly. In addition, you'll have processing for combining the data retrieved from the index with the data retrieved through the lookup operation, usually through one of the `JOIN` operators. The cost factor of lookups is the reason that nonclustered indexes are better suited for queries that return a small set of rows from the table. As the number of rows retrieved by a query increases, the overhead cost of a lookup becomes unacceptable. Also, if the optimizer has poor statistics and underestimates the number of rows being returned, lookups quickly become much more expensive than a scan.

To understand how a lookup makes a nonclustered index ineffective as the number of rows retrieved increases, let's look at a different example. The query that produced the execution plan in Figure 12-2 returned just a few rows from the `SalesOrderDetail` table. Leaving the query the same but changing the filter to a different value will, of course, change the number of rows returned. If you change the parameter value to look like this:

```
SELECT  *
FROM    Sales.SalesOrderDetail AS sod
WHERE   sod.ProductID = 793;
```

then running the query returns more than 700 rows, with different performance metrics and a completely different execution plan (Figure 12-3).

```
Duration: 195ms
Reads: 1,262
```

Figure 12-3. *A different execution plan for a query returning more rows*

To determine how costly it will be to use the nonclustered index, consider the number of logical reads (1,262) performed by the query during the table scan. If you force the optimizer to use the nonclustered index by using an index hint, like this:

```
SELECT  *
FROM    Sales.SalesOrderDetail AS sod WITH (INDEX (IX_SalesOrderDetail_
        ProductID))
WHERE   sod.ProductID = 793 ;
```

then the number of logical reads increases from 1,262 to 2,292.

```
Duration: 1,114ms
Reads: 2,292
```

Figure 12-4 shows the corresponding execution plan.

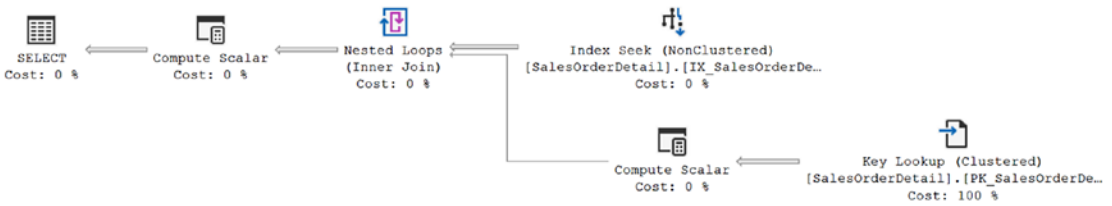


Figure 12-4. *Execution plan for fetching more rows with an index hint*

To benefit from nonclustered indexes, queries should request a relatively well-defined set of data. Application design plays an important role for the requirements that handle large result sets. For example, search engines on the Web mostly return a limited number of articles at a time, even if the search criterion returns thousands of matching articles. If the queries request a large number of rows, then the increased overhead cost of a lookup can make the nonclustered index unsuitable; subsequently, you have to consider the possibilities of avoiding the lookup operation.

Analyzing the Cause of a Lookup

Since a lookup can be a costly operation, you should analyze what causes a query plan to choose a lookup step in an execution plan. You may find that you are able to avoid the lookup by including the missing columns in the nonclustered index key or as `INCLUDE` columns at the index page level and thereby avoid the cost overhead associated with the lookup.

To learn how to identify the columns not included in the nonclustered index, consider the following query, which pulls information from the `HumanResources.Employee` table based on `NationalIDNumber`:

```
SELECT NationalIDNumber,
       JobTitle,
       HireDate
FROM HumanResources.Employee AS e
WHERE e.NationalIDNumber = '693168613';
```

This produces the following performance metrics and execution plan (see Figure 12-5):

```
Duration: 169 mc
Reads: 4
```

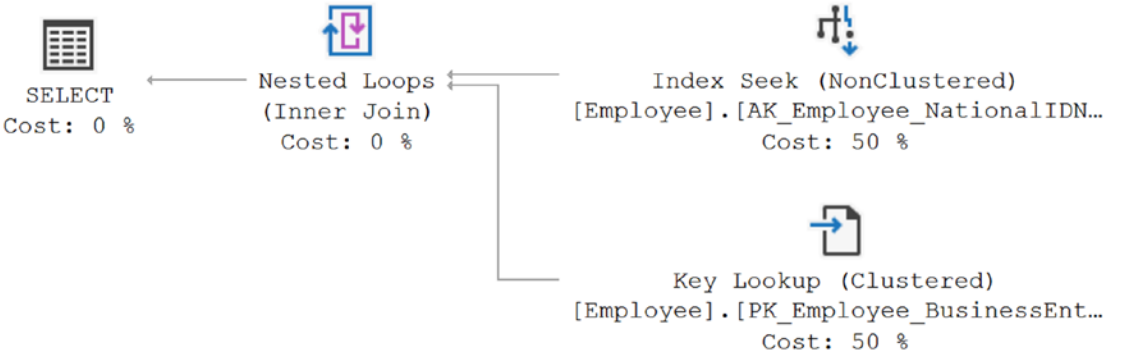


Figure 12-5. *Execution plan with a lookup*

As shown in the execution plan, you have a key lookup. The `SELECT` statement refers to columns `NationalIDNumber`, `JobTitle`, and `HireDate`. The nonclustered index on column `NationalIDNumber` doesn't provide values for columns `JobTitle` and `HireDate`, so a lookup operation was required to retrieve those columns from the data storage location. It's a `Key Lookup` because it's retrieving the data through the use of the clustered key stored with the nonclustered index. If the table were a heap, it would be an `RID` lookup. However, in the real world, it usually won't be this easy to identify all the columns used by a query. Remember that a lookup operation will be caused if all the columns referred to in any part of the query (not just the selection list) aren't part of the nonclustered index used.

In the case of a complex query based on views and user-defined functions, it may be too difficult to find all the columns referred to by the query. As a result, you need a standard mechanism to find the columns returned by the lookup that are not included in the nonclustered index.

If you look at the properties on the `Key Lookup (Clustered)` operation, you can see the output list for the operation. This shows you the columns being output by the lookup. To get the list of output columns quickly and easily and be able to copy them, right-click the operator, which in this case is `Key Lookup (Clustered)`. Then select the Properties menu item. Scroll down to the Output List property in the Properties window that opens (Figure 12-6). This property has an expansion arrow, which allows you to expand the column list, and has further expansion arrows next to each column, which allow you to expand the properties of the column.

⊟ Output List	[AdventureWorks2017].[HumanResources].[Employee].JobTitle, [A
⊟ [1]	[AdventureWorks2017].[HumanResources].[Employee].JobTitle
Alias	[e]
Column	JobTitle
Database	[AdventureWorks2017]
Schema	[HumanResources]
Table	[Employee]
⊟ [2]	[AdventureWorks2017].[HumanResources].[Employee].HireDate
Alias	[e]
Column	HireDate
Database	[AdventureWorks2017]
Schema	[HumanResources]
Table	[Employee]

Figure 12-6. *Key lookup Properties window*

To get the list of columns directly from the Properties window, click the ellipsis on the right side of the Output List property. This opens the output list in a text window from which you can copy the data for use when modifying your index (Figure 12-7).

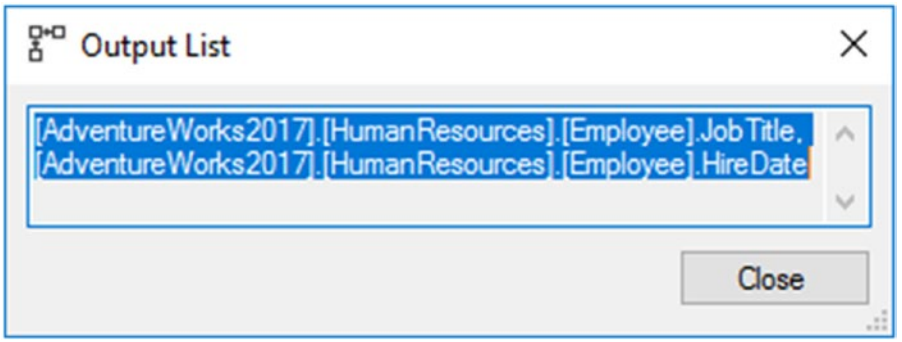


Figure 12-7. *The required columns that were not available in the nonclustered index*

Using that method does retrieve the data, but as you can see when comparing the information between Figures 12-6 and 12-7, there's a lot more information available if you drill in to the properties.

Resolving Lookups

Since the relative cost of a lookup can be high, you should, wherever possible, try to get rid of lookup operations. In the preceding section, you needed to obtain the values of columns `JobTitle` and `HireDate` without navigating from the index row to the data row. You can do this in three different ways, as explained in the following sections.

Using a Clustered Index

For a clustered index, the leaf page of the index is the same as the data page of the table. Therefore, when reading the values of the clustered index key columns, the database engine can also read the values of other columns without any navigation from the index row. In the previous example, if you convert the nonclustered index to a clustered index for a particular row, SQL Server can retrieve values of all the columns from the same page.

Simply saying that you want to convert the nonclustered index to a clustered index is easy to do. However, in this case, and in most cases you're likely to encounter, it isn't possible to do so since the table already has a clustered index in place. The clustered index on this table also happens to be the primary key. You would have to drop all foreign key constraints, drop and re-create the primary key as a nonclustered index, and then re-create the index against `NationallDNumber`. Not only do you need to take into

account the work involved, but you may seriously affect other queries that are dependent on the existing clustered index.

Note Remember that a table can have only one clustered index.

Using a Covering Index

In Chapter 8, you learned that a covering index is like a pseudoclustered index for the queries since it can return results without recourse to the table data. So, you can also use a covering index to avoid a lookup.

To understand how you can use a covering index to avoid a lookup, examine the query against the `HumanResources.Employee` table again.

```
SELECT  NationalIDNumber,
        JobTitle,
        HireDate
FROM HumanResources.Employee AS e
WHERE e.NationalIDNumber = '693168613';
```

To avoid this bookmark, you can add the columns referred to in the query, `JobTitle` and `HireDate`, directly to the nonclustered index key. This will make the nonclustered index a covering index for this query because all columns can be retrieved from the index without having to go to the heap or clustered index.

```
CREATE UNIQUE NONCLUSTERED INDEX AK_Employee_NationalIDNumber
ON HumanResources.Employee
(
    NationalIDNumber ASC,
    JobTitle ASC,
    HireDate ASC
)
WITH DROP_EXISTING;
```

Now when the query gets run, you'll see the following metrics and a different execution plan (Figure 12-8):

```
Duration: 164mc
Reads: 2
```

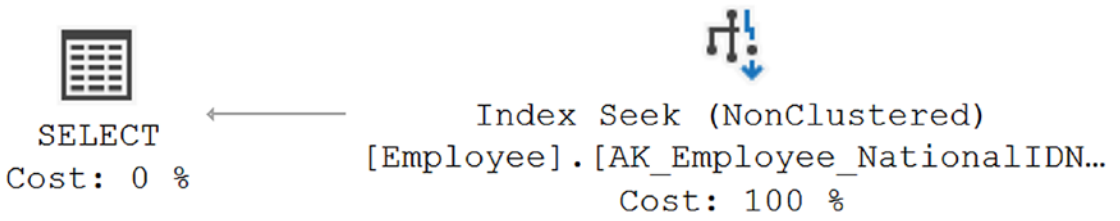


Figure 12-8. *Execution plan with a covering index*

There are a couple of caveats to creating a covering index by changing the key, however. If you add too many columns to a nonclustered index, it becomes wider. The index maintenance cost associated with the action queries can increase, as discussed in Chapter 8. Therefore, evaluate closely whether adding a key value will provide benefits to the general use of the index. If a key value is not going to be used for searches within the index, then it doesn't make sense to add it to the key. Also evaluate the number of columns (for size and data type) to be added to the nonclustered index key. If the total width of the additional columns is not too large (best determined through testing and measuring the resultant index size), then those columns can be added in the nonclustered index key to be used as a covering index. Also, if you add columns to the index key, depending on the index, of course, you may be affecting other queries in a negative fashion. They may have expected to see the index key columns in a particular order or may not refer to some of the columns in the key, causing the index to not be used by the optimizer. Modify the index by adding keys only if it makes sense based on these evaluations, especially because you have an alternative to modifying the key.

Another way to arrive at the covering index, without reshaping the index by adding key columns, is to use the `INCLUDE` columns. Change the index to look like this:

```
CREATE UNIQUE NONCLUSTERED INDEX AK_Employee_NationalIDNumber
ON HumanResources.Employee
(
    NationalIDNumber ASC
)
INCLUDE
(
    JobTitle,
    HireDate
)
WITH DROP_EXISTING;
```

Now when the query is run, you get the following metrics and execution plan (Figure 12-9):

```
Duration: 152mc
Reads: 2
```

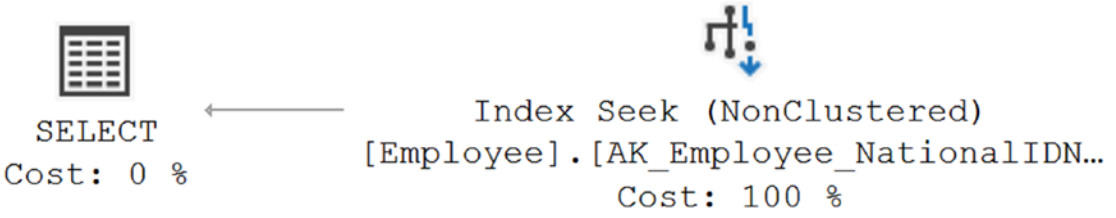


Figure 12-9. *Execution plan with INCLUDE columns*

The size of the index is, in this case, just a little bit smaller because of how the `INCLUDE` stores data on only the leaf pages instead of on every page. The index is still covering exactly as it was in the execution plan displayed in Figure 12-8. Because the data is stored at the leaf level of the index, when the index is used to retrieve the key values, the rest of the columns in the `INCLUDE` statement are available for use, almost like they were part of the key. Refer to Figure 12-10.

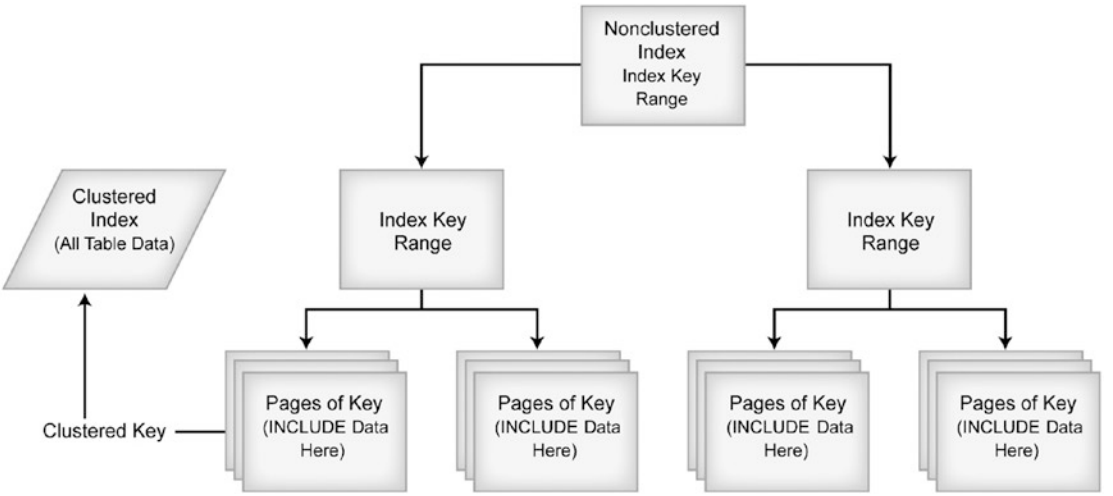


Figure 12-10. *Index storage using the INCLUDE keyword*

Another way to get a covering index is to take advantage of the structures within SQL Server. If the previous query were modified slightly to retrieve a different set of data instead of a particular `NationallDNumber` and its associated `JobTitle` and `HireDate`, this time the query would retrieve the `NationallDNumber` as an alternate key and the `BusinessEntitylD`, the primary key for the table, over a range of values.

```
SELECT  NationalIDNumber,
        BusinessEntityID
FROM    HumanResources.Employee AS e
WHERE   e.NationalIDNumber BETWEEN '693168613'
                           AND     '7000000000';
```

The original index, which we'll re-create now, on the table doesn't reference the `BusinessEntitylD` column in any way.

```
CREATE UNIQUE NONCLUSTERED INDEX AK_Employee_NationalIDNumber
ON HumanResources.Employee
(
    NationalIDNumber ASC
)
WITH DROP_EXISTING;
```

When the query is run against the table, you can see the results shown in Figure 12-11.

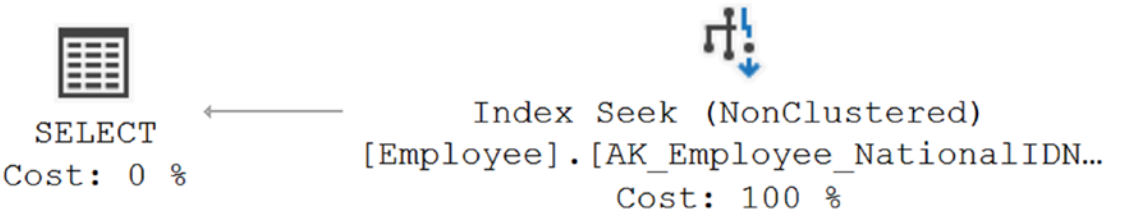
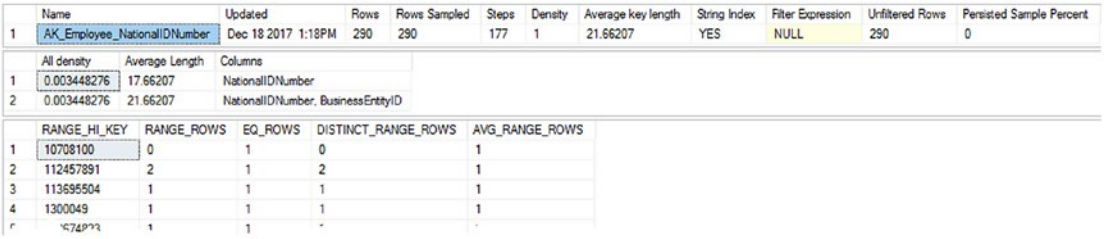


Figure 12-11. *Unexpected covering index*

How did the optimizer arrive at a covering index for this query based on the index provided? It's aware that on a table with a clustered index the clustered index key, in this case the `BusinessEntitylD` column, is stored as a pointer to the data with the nonclustered index. That means any query that incorporates a clustered index and a set of columns from a nonclustered index as part of the filtering mechanisms of the query, the `WHERE` clause, or the join criteria can take advantage of the covering index.

To see how these three different indexes are reflected in storage, you can look at the statistics of the indexes themselves using `DBCC SHOWSTATISTICS`. When you run the following query against the index, you can see the output in Figure 12-12:

```
DBCC SHOW_STATISTICS('HumanResources.Employee', AK_Employee_NationalIDNumber);
```



	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	AK_Employee_NationalIDNumber	Dec 18 2017 1:18PM	290	290	177	1	21.66207	YES	NULL	290	0

	All density	Average Length	Columns
1	0.003448276	17.66207	NationalIDNumber
2	0.003448276	21.66207	NationalIDNumber, BusinessEntityID

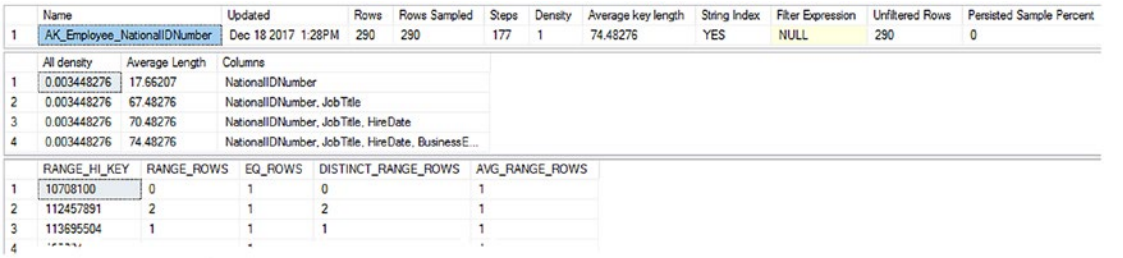
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	10708100	0	1	0	1
2	112457891	2	1	2	1
3	113695504	1	1	1	1
4	1300049	1	1	1	1

Figure 12-12. *DBCC SHOW_STATISTICS output for original index*

As you can see in the density graph of the statistics, the `NationalIDNumber` is listed first. The primary key for the table is included as part of the index, so a second row that includes the `BusinessEntityID` column is also part of the density graph. It makes the average length of the key about 22 bytes. This is how indexes that refer to the primary key values as well as the index key values can function as covering indexes.

If you run the same `DBCC SHOW_STATISTICS` on the first alternate index you tried, with all three columns included in the key, like so, you will see a different set of statistics (Figure 12-13):

```
CREATE UNIQUE NONCLUSTERED INDEX AK_Employee_NationalIDNumber
ON HumanResources.Employee
(
    NationalIDNumber ASC,
    JobTitle ASC,
    HireDate ASC
)
WITH DROP_EXISTING;
```



	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	AK_Employee_NationalIDNumber	Dec 18 2017 1:28PM	290	290	177	1	74.48276	YES	NULL	290	0

	All density	Average Length	Columns
1	0.003448276	17.66207	NationalIDNumber
2	0.003448276	67.48276	NationalIDNumber, JobTitle
3	0.003448276	70.48276	NationalIDNumber, JobTitle, HireDate
4	0.003448276	74.48276	NationalIDNumber, JobTitle, HireDate, BusinessE...

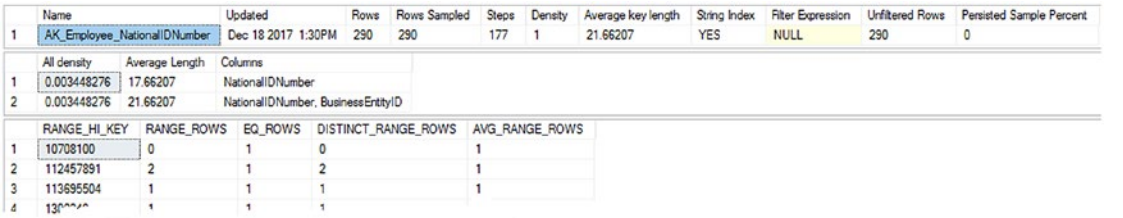
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	10708100	0	1	0	1
2	112457891	2	1	2	1
3	113695504	1	1	1	1
4					

Figure 12-13. *DBCC SHOW_STATISTICS output for a wide key covering index*

You now see the columns added up, all three of the index key columns, and finally the primary key added on. Instead of a width of 22 bytes, it's grown to 74. That reflects the addition of the `JobTitle` column, a `VARCHAR(50)` as well as the 6-byte-wide datetime field.

Finally, looking at the statistics for the second alternate index, with the included columns you'll see the output in Figure 12-14.

```
CREATE UNIQUE NONCLUSTERED INDEX AK_Employee_NationalIDNumber
ON HumanResources.Employee
(
    NationalIDNumber ASC
)
INCLUDE
(
    JobTitle,
    HireDate
)
WITH DROP_EXISTING;
```



	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	AK_Employee_NationalIDNumber	Dec 18 2017 1:30PM	290	290	177	1	21.66207	YES	NULL	290	0

	All density	Average Length	Columns
1	0.003448276	17.66207	NationalIDNumber
2	0.003448276	21.66207	NationalIDNumber, BusinessEntityID

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	10708100	0	1	0	1
2	112457891	2	1	2	1
3	113695504	1	1	1	1
4					

Figure 12-14. *DBCC SHOW_STATISTICS output for a covering index using INCLUDE*

Now the key width is back to the original size because the columns in the INCLUDE statement are stored not with the key but at the leaf level of the index.

There is more interesting information to be gleaned from the data stored about statistics, but I'll cover that in Chapter 13.

Using an Index Join

If the covering index becomes very wide, then you might consider a narrower index. As explained in Chapter 9, the optimizer can, if circumstances are just right, use an index intersection between two or more indexes to cover a query fully. Since an index join requires access to more than one index, it has to perform logical reads on all the indexes used in the index join. Consequently, it requires a higher number of logical reads than the covering index. But since the multiple narrow indexes used for the index join can serve more queries than a wide covering index (as explained in Chapter 9), you can certainly test your queries with multiple, narrow indexes to see whether you can get an index join to avoid lookups.

Note It is possible to get an index join, but they can be somewhat difficult to get the optimizer to recognize. You do need accurate statistics to assist the optimizer in this choice.

To better understand how an index join can be used to avoid lookups, run the following query against the PurchaseOrderHeader table to retrieve a PurchaseOrderID for a particular vendor on a particular date:

```
SELECT poh.PurchaseOrderID,
       poh.VendorID,
       poh.OrderDate
FROM Purchasing.PurchaseOrderHeader AS poh
WHERE VendorID = 1636
      AND poh.OrderDate = '2014/6/24';
```

When run, this query results in a Key Lookup operation (Figure 12-15) and the following I/O:

```
Duration: 251 mc
Reads: 10
```

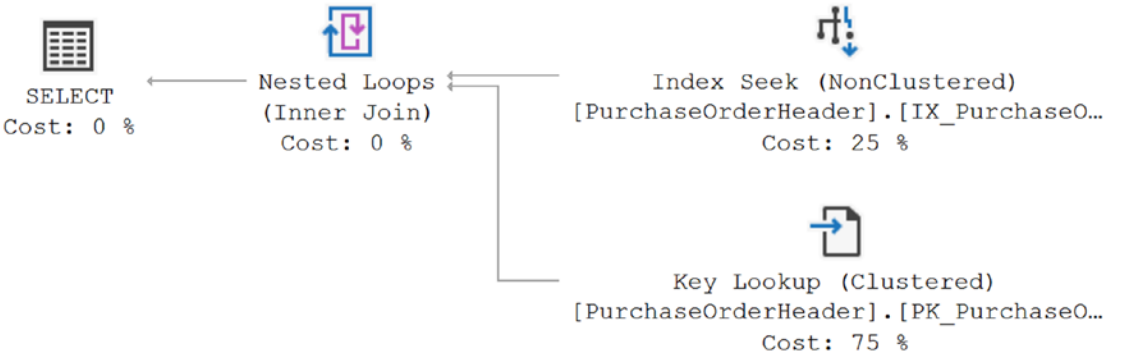


Figure 12-15. *A Key Lookup operation*

The lookup is caused since all the columns referred to by the SELECT statement and WHERE clause are not included in the nonclustered index on column VendorID. Using the nonclustered index is still better than not using it since that would require a scan on the table (in this case, a clustered index scan) with a larger number of logical reads.

To avoid the lookup, you can consider a covering index on the column OrderDate, as explained in the previous section. But in addition to the covering index solution, you can consider an index join. As you learned, an index join requires narrower indexes than the covering index and thereby provides the following two benefits:

- Multiple narrow indexes can serve a larger number of queries than the wide covering index.
- Narrow indexes require less maintenance overhead than the wide covering index.

To avoid the lookup using an index join, create a narrow nonclustered index on column OrderDate that is not included in the existing nonclustered index.

```
CREATE NONCLUSTERED INDEX IX_TEST
ON Purchasing.PurchaseOrderHeader
(
    OrderDate
);
```

If you run the SELECT statement again, the following output and the execution plan shown in Figure 12-16 are returned:

```
Duration: 219 mc
Reads: 4
```

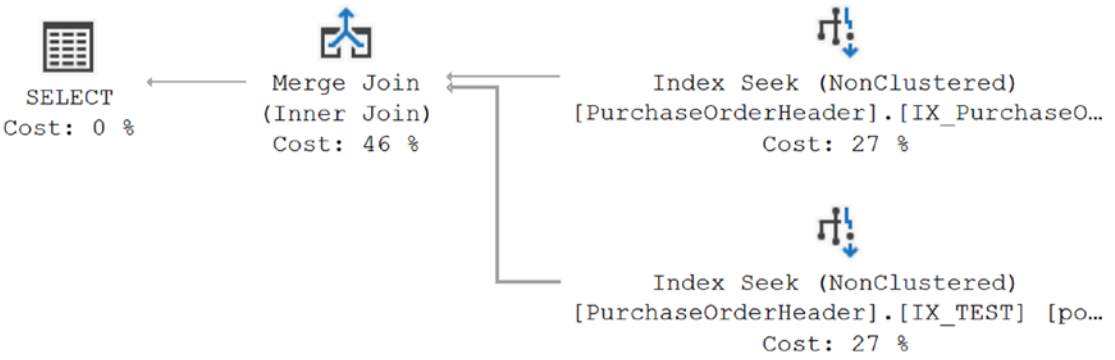


Figure 12-16. *Execution plan without a lookup*

From the preceding execution plan, you can see that the optimizer used the nonclustered index, IX_PurchaseOrder_VendorID, on column VendorID and the new nonclustered index, IX_TEST, on column OrderID to serve the query fully without hitting the storage location of the rest of the data. This index join operation avoided the lookup and consequently decreased the number of logical reads from 10 to 4.

It is true that a covering index on columns VendorID and OrderID could reduce the number of logical reads further. But it may not always be possible to use covering indexes since they can be wide and have their associated overhead. In such cases, an index join can be a good alternative.

Summary

As demonstrated in this chapter, the lookup step associated with a nonclustered index can make data retrieval through a nonclustered index very costly. The SQL Server optimizer takes this into account when generating an execution plan, and if it finds the overhead cost of using a nonclustered index to be high, it discards the index and performs a table scan (or a clustered index scan if the table is stored as a clustered index). Therefore, to improve the effectiveness of a nonclustered index, it makes sense

to analyze the cause of a lookup and consider whether you can avoid it completely by adding fields to the index key or to the `INCLUDE` column (or index join) and creating a covering index.

Up to this point, you have concentrated on indexing techniques and presumed that the SQL Server optimizer would be able to determine the effectiveness of an index for a query. In the next chapter, you will see the importance of statistics in helping the optimizer determine the effectiveness of an index.