

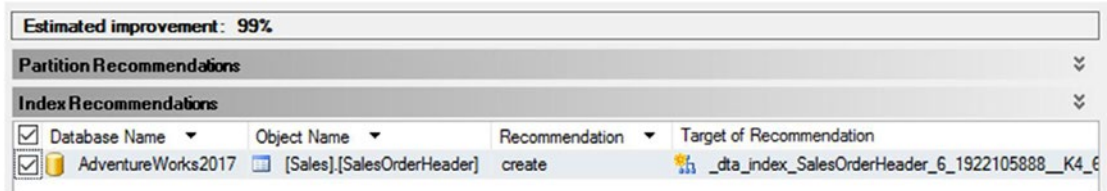
Figure 10-8. Query tuning initial recommendations

The Database Engine Tuning Advisor assumes that the load being tested is the full load of the database. For every test, every time. If the test you are running is not your representative workload, you could have serious issues with the suggested changes.

If there are indexes not being used, then they should be removed. This is a best practice and one that should be implemented on any database. However, in this case, this is a single query, not a full load of the system. To see whether the advisor can come up with a more meaningful set of recommendations, you must start a new session.

This time, I'll adjust the options so that the Database Engine Tuning Advisor will not be able to drop any of the existing structure. This is set on the Tuning Options tab (shown earlier in Figure 10-7). There I'll change the Physical Design Structure (PDS) to Keep in Database setting from Do Not Keep Any Existing PDS to Keep All Existing PDS. I'll keep

the running time the same because the evaluation worked well within the time frame. After running the Database Engine Tuning Advisor again, it finishes in less than a minute and displays the recommendations shown in Figure 10-9.

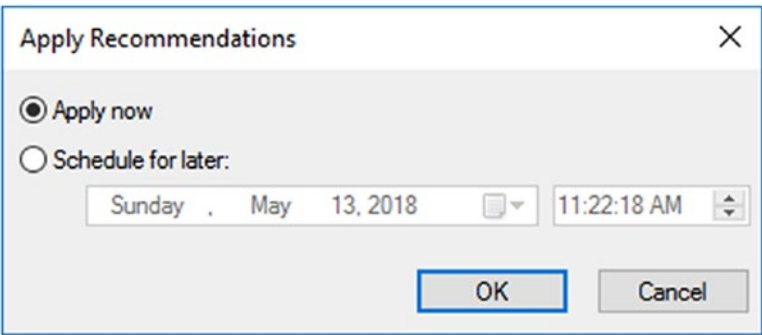


Estimated improvement: 99%			
Partition Recommendations			
Index Recommendations			
<input checked="" type="checkbox"/> Database Name	<input type="text"/> Object Name	<input type="text"/> Recommendation	<input type="text"/> Target of Recommendation
<input checked="" type="checkbox"/> AdventureWorks2017	<input type="text"/> [Sales].[SalesOrderHeader]	create	_dta_index_SalesOrderHeader_6_1922105888__K4_6

**Figure 10-9.** Query tuning recommendations

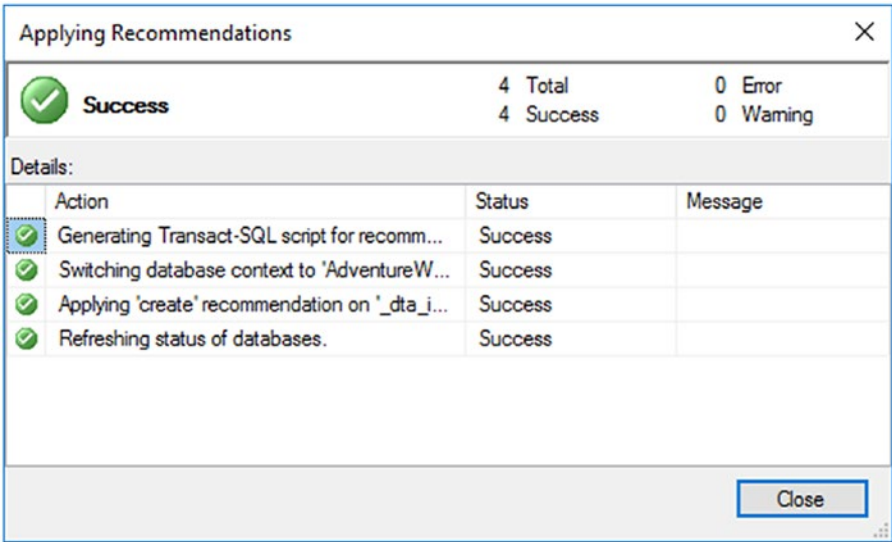
The first time through, the Database Engine Tuning Advisor suggested dropping most of the indexes on the tables being tested and a bunch of the related tables. This time it suggests creating a covering index on the columns referenced in the query. As outlined in Chapter 9, a covering index can be one of the best-performing methods of retrieving data. The Database Engine Tuning Advisor was able to recognize that an index with all the columns referenced by the query, a covering index, would perform best.

Once you've received a recommendation, you should closely examine the proposed T-SQL command. The suggestions are not always helpful, so you need to evaluate and test them to be sure. Assuming the examined recommendation looks good, you'll want to apply it. Select **Actions ► Evaluate Recommendations**. This opens a new Database Engine Tuning Advisor session and allows you to evaluate whether the recommendations will work using the same measures that made the recommendations in the first place. All of this is to validate that the original recommendation has the effect that it claims it will have. The new session looks just like a regular evaluation report. If you're still happy with the recommendations, select **Actions ► Apply Recommendation**. This opens a dialog box that allows you to apply the recommendation immediately or schedule the application (see Figure 10-10).



**Figure 10-10.** *Apply Recommendations dialog box*

If you click the OK button, the Database Engine Tuning Advisor will apply the index to the database where you’ve been testing queries (see Figure 10-11).



**Figure 10-11.** *A successful tuning session applied*

After you generate recommendations, you may want to, instead of applying them on the spot, save the T-SQL statements to a file and accumulate a series of changes for release to your production environment during scheduled deployment windows. Also, just taking the defaults, you’ll end up with a lot of indexes named something like this: `_dta_index_SalesOrderHeader_5_1266103551__K4_6_11`. That’s not terribly clear, so saving the changes to T-SQL will also allow you to make your changes more human

readable. Remember that applying indexes to tables, especially large tables, can cause a performance impact to processes actively running on the system while the index is being created.

Although getting index suggestions one at a time is nice, it would be better to be able to get large swaths of the database checked all at once. That's where tuning a trace workload comes in.

## Tuning a Trace Workload

Capturing a trace from the real-world queries that are running against a production server is a way to feed meaningful data to the Database Engine Tuning Advisor. (Capturing traces will be covered in Chapter 18.) The easiest way to define a trace for use in the Database Engine Tuning Advisor is to implement the trace using the Tuning template. Start the trace on the system you need to tune. I generated an artificial load by running queries in a loop from the PowerShell `sqlps.exe` command prompt. This is the PowerShell command prompt with the SQL Server configuration settings. It gets installed with SQL Server.

To find something interesting, I'm going to create one stored procedure with an obvious tuning issue.

```
CREATE PROCEDURE dbo.uspProductSize
AS
SELECT  p.ProductID,
        p.Size
FROM    Production.Product AS p
WHERE   p.Size = '62';
```

Here is the simple PowerShell script I used. You'll need to adjust the connection string for your environment. After you have downloaded the file to a location, you'll be able to run it by simply referencing the file and the full path through the command prompt. You may run into security issues since this is an unsigned, raw script. Follow the help guidance provided in that error message if you need to (`queryload.ps1`).

```

$SqlConnection = New-Object System.Data.SqlClient.SqlConnection
$SqlConnection.ConnectionString = 'Server=WIN-8A2LQANSO51;Database=AdventureWorks2017;trusted_connection=true'

# Load Product data
$ProdCmd = New-Object System.Data.SqlClient.SqlCommand
$ProdCmd.CommandText = "SELECT ProductID FROM Production.Product"
$ProdCmd.Connection = $SqlConnection
$SqlAdapter = New-Object System.Data.SqlClient.SqlDataAdapter
$SqlAdapter.SelectCommand = $ProdCmd
$ProdDataSet = New-Object System.Data.DataSet
$SqlAdapter.Fill($ProdDataSet)

# Set up the procedure to be run
$WhereCmd = New-Object System.Data.SqlClient.SqlCommand
$WhereCmd.CommandText = "dbo.uspGetWhereUsedProductID @StartProductID = @
ProductId, @CheckDate=NULL"
$WhereCmd.Parameters.Add("@ProductID",[System.Data.SqlDbType]"Int")
$WhereCmd.Connection = $SqlConnection

# And another one
$BomCmd = New-Object System.Data.SqlClient.SqlCommand
$BomCmd.CommandText = "dbo.uspGetBillOfMaterials @StartProductID = @
ProductId, @CheckDate=NULL"
$BomCmd.Parameters.Add("@ProductID",[System.Data.SqlDbType]"Int")
$BomCmd.Connection = $SqlConnection

# Bad Query
$BadQuerycmd = New-Object System.Data.SqlClient.SqlCommand
$BadQuerycmd.CommandText = "dbo.uspProductSize"
$BadQuerycmd.Connection = $SqlConnection

while(1 -ne 0)
{
    $RefID = $row[0]
    $SqlConnection.Open()
    $BadQuerycmd.ExecuteNonQuery() | Out-Null
    $SqlConnection.Close()
}

```

```

foreach($row in $ProdDataSet.Tables[0])
{
    $SqlConnection.Open()
    $BomCmd.Parameters["@ProductID"].Value = $ProductId
    $BomCmd.ExecuteNonQuery() | Out-Null
    $SqlConnection.Close()

    $SqlConnection.Open()
    $ProductId = $row[0]
    $WhereCmd.Parameters["@ProductID"].Value = $ProductId
    $WhereCmd.ExecuteNonQuery() | Out-Null
    $SqlConnection.Close()
}
}

```

---

**Note** For more information on PowerShell, check out *PowerShell in a Month of Lunches* by Don Jones and Jeffrey Hicks (Manning, 2016).

---

Once you've created the trace file, open the Database Engine Tuning Advisor. It defaults to a file type under the Workload section, so you'll only have to browse to the trace file location. As before, you'll want to select the AdventureWorks2017 database as the database for workload analysis from the drop-down list. To limit the suggestions, also select AdventureWorks2012 from the list of databases at the bottom of the screen. Set the appropriate tuning options and start the analysis. This time, it will take more than a minute to run (see Figure 10-12).

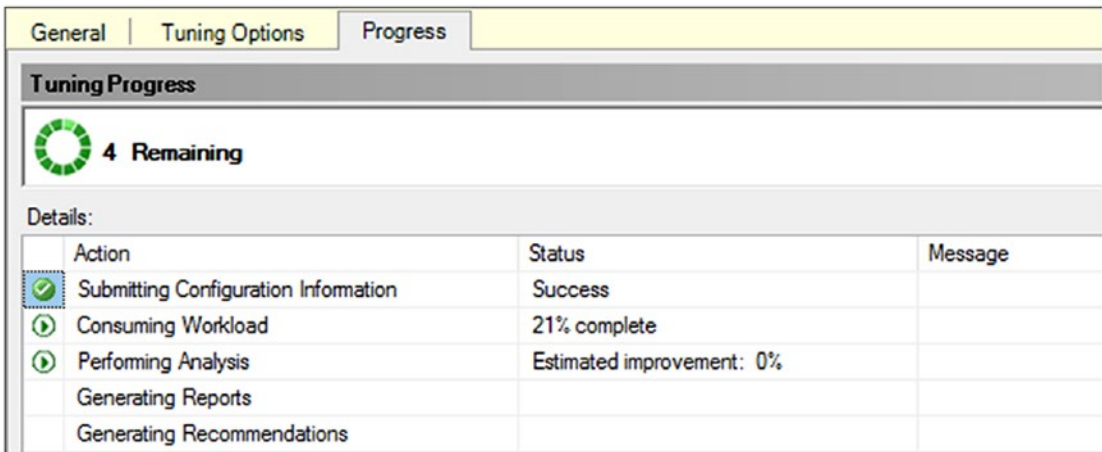


Figure 10-12. Database tuning engine in progress

The processing runs for about 15 minutes on my machine. Then it generates output, shown in Figure 10-13.

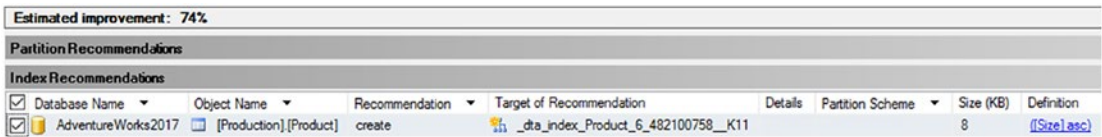


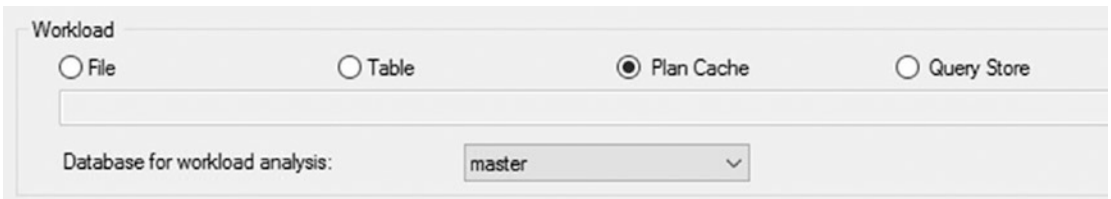
Figure 10-13. Recommendation for a manual statistic

After running all the queries through the Database Engine Tuning Advisor, the advisor came up with a suggestion for a new index for the Product table that would improve the performance the query. Now I just need to save that to a T-SQL file so that I can edit the name prior to applying it to my database.

## Tuning from the Procedure Cache

You can take advantage of the query plans that are stored in the cache as a source for tuning recommendations. The process is simple. There’s a choice on the General page that lets you choose the plan cache as a source for the tuning effort, as shown in Figure 10-14.





**Figure 10-14.** Selecting Plan Cache as the source for the DTA

All other options behave exactly the same way as previously outlined in this chapter. The processing time is radically less than when the advisor processes a workload. It has only the queries in cache to process so, depending on the amount of memory in your system, may be a short list. The results from processing my cache suggested several indexes and some individual statistics, as you can see in Figure 10-15.

Database Name	Object Name	Recommendation	Target of Recommendation	Details	Partition Scheme	Size (KB)	Definition
AdventureWorks2017	(Production) [BIO Materials]	create	idx_index_BIO Materials_6_1157579162_K2_K1_K5_K3_K8_K7			194	(ProductSearch) (I) as, UserData as, ItemData as, ComponentClass, PartSearch (I) as, BOMLevel
AdventureWorks2017	(Production) [BIO Materials]	create	idx_index_BIO Materials_6_1157579162_K3_K4_K5_K2_K8_K7			194	(ComponentCI) as, (PartDesc) as, (StdCost) as, (ProductAssemblyCI) as, (PartSearchCI) as, (BOMLevel)
AdventureWorks2017	(Production) [BIO Materials]	create	idx_index_1157579162_2_3_3_2_3_4				(ProductSearchCI) as, (ComponentCI) as, (PartSearchCI) as, (BOMLevel) as, (StdCost) as, (PartDesc)
AdventureWorks2017	(Production) [BIO Materials]	create	idx_index_1157579162_2_3_2_3_3_3				(ComponentCI) as, (ProductSearchCI) as, (PartDesc) as, (BOMLevel) as, (PartSearchCI) as, (StdCost)
AdventureWorks2017	(Production) [Product]	create	idx_index_Product_6_48250258_K1_K2_K8_K10			32	(ProductCI) as, (Name) as, (StandardCost) as, (PartDesc) as, (StdCost)
AdventureWorks2017	(Production) [Product]	create	idx_index_48250258_2_3_3_10_1				(StdCost) as, (StandardCost) as, (PartDesc) as, (ProductID)

**Figure 10-15.** Recommendations from the plan cache

This gives you one more mechanism to try to tune your system in an automated fashion. But it is limited to the queries that are currently in cache. Depending on the volatility of your cache (the speed at which plans age out or are replaced by new plans), this may or may not prove useful.

## Tuning from the Query Store

We're going to cover the Query Store in Chapter 11. However, we can take advantage of the information that the Query Store gathers in an attempt to get tuning suggestions from the Tuning Advisor. You select the Query Store workload from the list shown in Figure 10-14. Then you have to select a database because the Query Store is turned on only for individual databases. However, from there, the tuning options and behavior are the same. From my system there were several more suggestions than what were pulled from the plan cache, as shown in Figure 10-16.



Estimated improvement: 56%

Partition Recommendations						
Database Name	Object Name	Recommendation	Target of Recommendation	Details	Partition Scheme	Size (KB)
AdventureWorks2017	[Production].[BIO1Material]	create	idx_index_BIO1Material_6_1157579162_K2_K3_K4_K5_K6_K7			164
AdventureWorks2017	[Production].[BIO1Material]	create	idx_index_BIO1Material_6_1157579162_K3_K4_K5_K6_K7_K8_K9			164
AdventureWorks2017	[Production].[BIO1Material]	create	idx_index_1157579162_2_3_4_5_6_7_8_9			32
AdventureWorks2017	[Production].[Product]	create	idx_index_Product_6_402100758_K1_K2_K3_K10			200
AdventureWorks2017	[Production].[Product]	create	idx_index_402100758_2_3_10_...			216
AdventureWorks2017	[Production].[TransactionHistory]	create	idx_index_TransactionHistory_6_1230627427_K3_K1_K2			216
AdventureWorks2017	[Production].[TransactionHistory]	create	idx_index_1230627427_2_1			

Figure 10-16. Recommendations from the Query Store

The reason there were even more suggestions is because the Query Store contains more plans than those that are simply in the plan cache or those that are captured during an individual trace run. That improved data makes the Query Store an excellent resource to use for tuning recommendations.

## Database Engine Tuning Advisor Limitations

The Database Engine Tuning Advisor recommendations are based on the input workload. If the input workload is not a true representation of the actual workload, then the recommended indexes may sometimes have a *negative* effect on some queries that are missing in the workload. But most important, in many cases, the Database Engine Tuning Advisor may not recognize possible tuning opportunities. It has a sophisticated testing engine, but in some scenarios, its capabilities are limited.

For a production server, you should ensure that the SQL trace includes a complete representation of the database workload. For most database applications, capturing a trace for a complete day usually includes most of the queries executed on the database, although there are exceptions to this such as weekly, monthly, or year-end processing. Be sure you understand your load and what’s needed to capture it appropriately. A few of the other considerations/limitations with the Database Engine Tuning Advisor are as follows:

- Trace input using the SQL: BatchCompleted event:* As mentioned earlier, the SQL trace input to the Database Engine Tuning Advisor must include the SQL:BatchCompleted event; otherwise, the wizard won’t be able to identify the queries in the workload.
- Query distribution in the workload:* In a workload, a query may be executed multiple times with the same parameter value. Even a small performance improvement to the most common query can make a bigger contribution to the performance of the overall workload, compared to a large improvement in the performance of a query that is executed only once.

- *Index hints*: Index hints in a SQL query can prevent the Database Engine Tuning Advisor from choosing a better execution plan. The wizard includes all index hints used in a SQL query as part of its recommendations. Because these indexes may not be optimal for the table, remove all index hints from queries before submitting the workload to the wizard, bearing in mind that you need to add them back in to see whether they do actually improve performance.

Remember that the Tuning Advisor's recommendations are just that, recommendations. The suggestions it offers may not work as suggested by the advisor, and you may already have indexes in place that would serve just as well as the suggested indexes. Test and validate all suggestions prior to implementation.

## Summary

As you learned in this chapter, the Database Engine Tuning Advisor can be a useful tool for analyzing the effectiveness of existing indexes and recommending new indexes for a SQL workload. As the SQL workload changes over time, you can use this tool to determine which existing indexes are no longer in use and which new indexes are required to improve performance. It can be a good idea to run the wizard occasionally just to check that your existing indexes really are the best fit for your current workload. This assumes you're not capturing metrics and evaluating them yourself. The Database Engine Tuning Advisor also provides many useful reports for analyzing the SQL workload and the effectiveness of its own recommendations. Just remember that the limitations of the tool prevent it from spotting all tuning opportunities. Also remember that the suggestions provided by the DTA are only as good as the input you provide to it. If your database is in bad shape, this tool can give you a quick leg up. If you're already monitoring and tuning your queries regularly, you may see no benefit from the recommendations of the Database Engine Tuning Advisor.

Capturing query metrics and execution plans used to be a lot of work to automate and maintain. However, capturing that information is vital in your query tuning efforts. Starting with SQL Server 2016, the Query Store provides a wonderful mechanism for capturing query metrics and so much more. The next chapter will give you a thorough understanding of all the functionality that the Query Store offers.

## CHAPTER 11

# Query Store

The Query Store was introduced originally in Azure SQL Database in 2015 and was first introduced to SQL Server in version 2016. The Query Store provides three pieces of functionality that you're going to want to take advantage of. First, you get query metrics and execution plans, stored permanently in the database in structures that are easy to access so that you have good, flexible information about the performance of the queries on your system. Second, the Query Store creates a mechanism for directly controlling execution plan behavior in a way we've never had before. Finally, the Query Store acts as a safety and reporting mechanism for database upgrades that will enable you to protect your systems in new ways.

In this chapter, I cover the following topics:

- How the Query Store works and the information it collects
- Reports and mechanisms exposed through Management Studio for Query Store behavior
- Plan forcing, a method for controlling which execution plans are used by SQL Server and Azure SQL Database
- An upgrade method that helps you protect your system behavior

While Extended Events sessions are your go-to measure for precision, for most systems, the Query Store should be the principal means of monitoring your query performance.

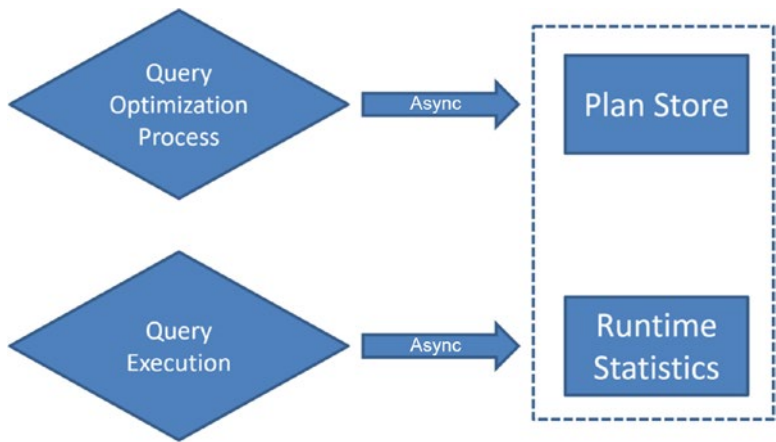
## Query Store Function and Design

The Query Store is probably the most lightweight mechanism in terms of impact on the system. It provides the core of what you need to properly understand the query performance of your system. Plus, because all the work around the Query Store is done with system views, you get to use T-SQL to work with it, so using it becomes

incredibly easy. Using the DMOs, trace events, and even, to a degree, Extended Events as mechanisms for monitoring query metrics really can be considered old school with the introduction of the Query Store.

## Query Store Behavior

The Query Store collects two pieces of information. First, it collects an aggregate of each query’s behavior on your system. Second, the Query Store captures, by default, every execution plan created on your system, up to the maximum number of plans per query (200 by default). You can turn the Query Store on and off on a database-by-database basis. When it’s on, the Query Store functions as shown in Figure 11-1.



**Figure 11-1.** Behavior of the Query Store in collecting data

The query optimization process occurs as normal. When a query is submitted to the system, an execution plan is created (covered in detail in Chapter 15) and stored in the plan cache (which will be explained in Chapter 16). After these processes are complete, an asynchronous process runs for the Query Store to capture execution plans from the plan cache. Initially it writes these plans to a separate piece of memory for temporary storage. Another asynchronous process will then write these execution plans to the Query Store in the database. All these are asynchronous processes to ensure that there is minimal, although not zero, impact on other processes within the system. The only exception to the flow of this process is plan forcing, which we’ll cover later in the chapter.

The query execution then occurs just as with any other query. Once the query execution is complete, query runtime metrics, such as the number of reads, the number of writes, the duration of the query, and wait statistics, are written to a separate memory space, again, asynchronously. At a later point, another asynchronous process will write that information to disk. The information that is gathered and written to disk is aggregated. The default aggregation time is 60-minute intervals.

All the information stored within the Query Store system tables is written permanently to the database on which the Query Store is enabled. The query metrics and the execution plans for the queries are kept with the database. They get backed up with the database, and they get restored with the database. In the event of your system going offline or failing over, it is possible to lose some of the Query Store information that was still in memory and not yet written to disk. The default interval for writing to the disk is 15 minutes. Considering this is aggregate data, that's not a bad interval for the possibility of some Query Store data loss for what should not be considered production-level data.

When you query the information from the Query Store, it combines both the in-memory data and the data written to disk. You don't have to do anything extra to access that information.

Before continuing with the rest of the chapter, if you want to follow along with some of the code and processing, you'll need to enable the Query Store on one of the databases. This command will make it happen:

```
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE = ON;
```

To ensure you have queries in the Query Store as you follow along, let's use this stored procedure:

```
CREATE OR ALTER PROC dbo.ProductTransactionHistoryByReference (
    @ReferenceOrderID int
)
AS
BEGIN
    SELECT p.Name,
           p.ProductNumber,
           th.ReferenceOrderID
```

```

FROM    Production.Product AS p
JOIN    Production.TransactionHistory AS th
        ON th.ProductID = p.ProductID
WHERE   th.ReferenceOrderID = @ReferenceOrderID;
END

```

If you execute the stored procedure with these three values, removing it from cache each time, you'll actually get three different execution plans.

```

DECLARE @Planhandle VARBINARY(64);

EXEC dbo.ProductTransactionHistoryByReference @ReferenceOrderID = 0;

SELECT @Planhandle = deps.plan_handle
FROM sys.dm_exec_procedure_stats AS deps
WHERE deps.object_id = OBJECT_ID('dbo.ProductTransactionHistoryByReference');

IF @Planhandle IS NOT NULL
BEGIN
    DBCC FREEPROCCACHE(@Planhandle);
END

EXEC dbo.ProductTransactionHistoryByReference @ReferenceOrderID = 53465;

SELECT @Planhandle = deps.plan_handle
FROM sys.dm_exec_procedure_stats AS deps
WHERE deps.object_id = OBJECT_ID('dbo.ProductTransactionHistoryByReference');

IF @Planhandle IS NOT NULL
BEGIN
    DBCC FREEPROCCACHE(@Planhandle);
END

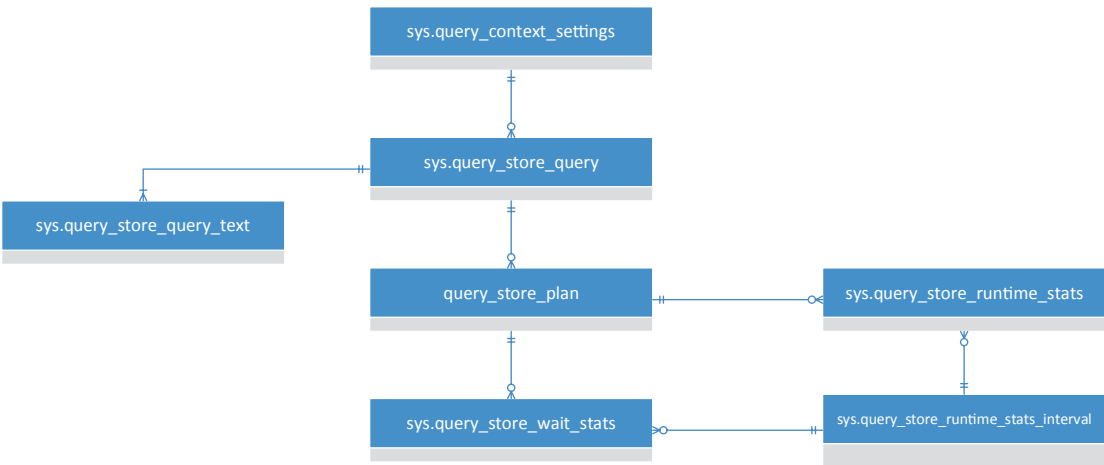
EXEC dbo.ProductTransactionHistoryByReference @ReferenceOrderID = 3849;

```

With this, you can be sure that you'll have information in the Query Store.

# Information Query Store Collects

The Query Store collects a fairly narrow but extremely rich set of data. Figure 11-2 represents the system tables and their relationships.



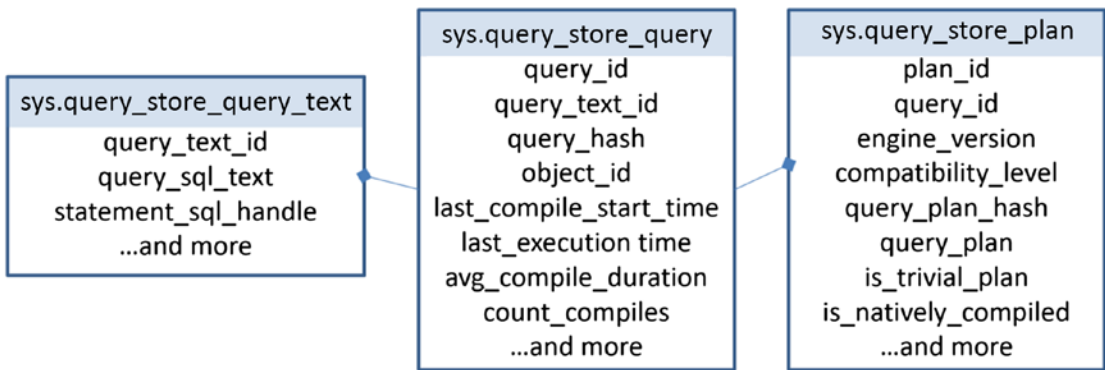
**Figure 11-2.** System views of the Query Store

The information stored within the Query Store breaks down into two basic sets. There is the information about the query itself, including the query text, the execution plan, and the query context settings. Then there is the runtime information that consists of the runtime intervals, the wait statistics, and the query runtime statistics. We’ll approach each section of information separately, starting with the information about the query.

## Query Information

The core piece of data to the Query Store is the query itself. The query is independent from, though it may be part of, stored procedures or batches. It comes down to the fundamental query text and the query\_hash value (a hash of the query text) that lets you identify any given query. This data is then combined with the query plans and the actual query text. Figure 11-3 shows the basic structure and some of the data.





**Figure 11-3.** Query information stored within the Query Store

These are system tables stored in the Primary file group of any database that has the Query Store enabled. While there are good reports built into the Management Studio interface, you can write your own queries to access the information from the Query Store. For example, this query could retrieve all the query statements for a given stored procedure along with the execution plan:

```
SELECT qsq.query_id,  
       qsq.object_id,  
       qsqt.query_sql_text,  
       CAST(qsp.query_plan AS XML) AS QueryPlan  
FROM sys.query_store_query AS qsq  
     JOIN sys.query_store_query_text AS qsqt  
         ON qsq.query_text_id = qsqt.query_text_id  
     JOIN sys.query_store_plan AS qsp  
         ON qsp.query_id = qsq.query_id  
WHERE qsq.object_id = OBJECT_ID('dbo.ProductTransactionHistoryByReference');
```

While each individual query statement is stored within the Query Store, you also get the `object_id`, so you can use functions such as `OBJECT_ID()` as I did to retrieve the information. Note that I also had to use the `CAST` command on the `query_plan` column. This is because the Query Store rightly stores this column as text, not as XML. The XML data type in SQL Server has a nesting limit that would require two columns, XML for those that meet the requirement and `NVARCHAR(MAX)` for those that don't. When building

the Query Store, they addressed that issue by design. If you want to be able to click the results, similar to Figure 11-4, to see the execution plan, you'll need to use CAST as I did earlier.

	query_id	object_id	query_sql_text	plan_id	QueryPlan
1	75	1255675521	(@ReferenceOrderID int)SELECT p.Nam...	75	<a href="#">&lt;ShowPlanXML xmlns="http://schemas.microsoft.com...</a>
2	75	1255675521	(@ReferenceOrderID int)SELECT p.Nam...	76	<a href="#">&lt;ShowPlanXML xmlns="http://schemas.microsoft.com...</a>
3	75	1255675521	(@ReferenceOrderID int)SELECT p.Nam...	82	<a href="#">&lt;ShowPlanXML xmlns="http://schemas.microsoft.com...</a>

**Figure 11-4.** Information retrieved from the Query Store using T-SQL

In this instance, for a single query, query\_id = 75, which is a one-statement stored procedure, I have three distinct execution plans as identified by the three different plan\_id values. We'll be looking at these plans a little later.

Another thing to note from the results of the Query Store is how the text is stored. Since this statement is part of a stored procedure with parameters, the parameter values that are used in the T-SQL text are defined. This is what the statement looks like within the Query Store (formatting left as is):

```
(@ReferenceOrderID int)SELECT  p.Name,                p.ProductNumber,
                                th.ReferenceOrderID    FROM    Production.Product
AS p      JOIN    Production.TransactionHistory AS
th
          ON th.ProductID = p.ProductID      WHERE
th.ReferenceOrderID = @ReferenceOrderID
```

Note the parameter definition at the start of the statement. Just a reminder from earlier, this is what the actual stored procedure definition looks like:

```
CREATE OR ALTER PROC dbo.ProductTransactionHistoryByReference (
    @ReferenceOrderID int
)
AS
BEGIN
    SELECT  p.Name,
            p.ProductNumber,
            th.ReferenceOrderID
    FROM    Production.Product AS p
    JOIN    Production.TransactionHistory AS th
```

```

        ON th.ProductID = p.ProductID
WHERE   th.ReferenceOrderID = @ReferenceOrderID;
END

```

The statements within the procedure and statement as stored in the Query Store are different. This can lead to some issues when attempting to find a particular query within the Query Store. Let's look at a different example, shown here:

```

SELECT a.AddressID,
       a.AddressLine1
FROM Person.Address AS a
WHERE a.AddressID = 72;

```

This is a batch instead of a stored procedure. Executing this for the first time will load it into the Query Store using the process outlined earlier. If we run some T-SQL to retrieve information on this statement as follows, there will be nothing returned:

```

SELECT qsq.query_id,
       qsq.query_hash,
       qsqt.query_sql_text
FROM sys.query_store_query AS qsq
     JOIN sys.query_store_query_text AS qsqt
         ON qsqt.query_text_id = qsq.query_text_id
WHERE qsqt.query_sql_text = 'SELECT a.AddressID,
       a.AddressLine1
FROM Person.Address AS a
WHERE a.AddressID = 72;';

```

Because this statement was so simple, the optimizer was able to perform a process called *simple parameterization* on it. Luckily, the Query Store has a function for dealing with automatic parameterization, `sys.fn_stmt_sql_handle_from_sql_stmt`. That function allows you to find the information from the query as follows:

```

SELECT qsq.query_id,
       qsq.query_hash,
       qsqt.query_sql_text,
       qsq.query_parameterization_type
FROM sys.query_store_query_text AS qsqt

```

```

JOIN sys.query_store_query AS qsq
    ON qsq.query_text_id = qsqt.query_text_id
JOIN sys.fn_stmt_sql_handle_from_sql_stmt(
    'SELECT a.AddressID,
        a.AddressLine1
FROM Person.Address AS a
WHERE a.AddressID = 72;',
    2) AS fsshfss
    ON fsshfss.statement_sql_handle = qsqt.statement_sql_handle;

```

The formatting and the white space all have to be the same in order for this to work. The hard-coded value can change, but all the rest has to be the same. Running the query results in what you see in Figure 11-5.

	query_id	query_hash	query_sql_text	query_parameterization_type
1	1054	0xDE0BD0B755E53296	(@1 tinyint)SELECT [a].[AddressID],[a].[AddressL...	2

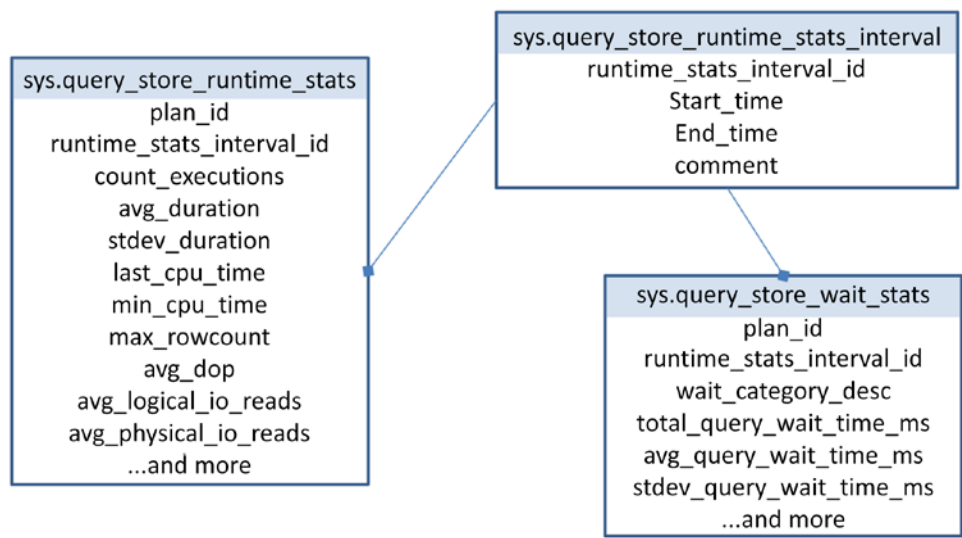
**Figure 11-5.** Results showing simple parameterization

You can see in the `query_sql_text` column where the parameter value for the simple parameterization has been added to the text just as it was for the stored procedure. The bad news is `sys.fn_stmt_sql_handle_from_sql_stmt` currently works only with automatic parameterization. It won't help you locate parameterized statements from any other source. To retrieve that information, you will be forced to use the `LIKE` command to search through the text or, as I did earlier, use the `object_id` for queries in stored procedures.

## Query Runtime Data

After you retrieve information about the query and the plan, the next thing you're going to want is to see runtime metrics. There are two keys to understanding the runtime metrics. First, the metrics connect back to the plan, not to the query. Since each plan could behave differently, with different operations against different indexes with different join types and all the rest, capturing runtime data and wait statistics means tying back to the plan. Second, the runtime and wait statistics are aggregated, but they are aggregated by the runtime interval. The default value for the runtime interval is 60 minutes. This means you'll have a different set of metrics for each plan for each runtime interval.

All this information is available as shown in Figure 11-6.



**Figure 11-6.** System tables containing runtime and wait statistics

When you begin to query the runtime metrics, you can easily combine them with the information on the query itself. You will have to deal with the intervals, and the best way to deal with them may be to group them and aggregate them, taking averages of the averages, and so on. That may seem like a pain, but you need to understand why the information is broken up that way. When you’re looking at query performance, you need several numbers, such as current performance, hoped-for performance, and future performance after we make changes. Without these numbers to compare, you can’t know whether something is slow or whether you have improved it. The same thing goes for the information in the Query Store. By breaking everything apart into intervals, you can compare today to yesterday, one moment in time to another. That’s how you can know that performance truly did degrade (or improve), that it ran faster/slower yesterday, and so on. If you have only averages and not averages over time, then you won’t see how the behavior changes over time. With the time intervals, you get some of the granularity of capturing the metrics yourself using Extended Events combined with the ease of use of querying the cache.

A query that retrieves performance metrics for a given moment in time can be written just like this:

```
DECLARE @CompareTime DATETIME = '2017-11-28 21:37';

SELECT CAST(qsp.query_plan AS XML),
       qsrs.count_executions,
       qsrs.avg_duration,
       qsrs.stdev_duration,
       qsws.wait_category_desc,
       qsws.avg_query_wait_time_ms,
       qsws.stdev_query_wait_time_ms
FROM sys.query_store_plan AS qsp
     JOIN sys.query_store_runtime_stats AS qsrs
         ON qsrs.plan_id = qsp.plan_id
     JOIN sys.query_store_runtime_stats_interval AS qsrsi
         ON qsrsi.runtime_stats_interval_id = qsrs.runtime_stats_interval_id
     JOIN sys.query_store_wait_stats AS qsws
         ON qsws.plan_id = qsrs.plan_id
         AND qsws.execution_type = qsrs.execution_type
         AND qsws.runtime_stats_interval_id = qsrs.runtime_stats_
interval_id
WHERE qsrs.object_id = OBJECT_ID('dbo.ProductTransactionHistoryByReference')
     AND @CompareTime BETWEEN qsrsi.start_time
                             AND qsrsi.end_time;
```

Let's break this down. You can see that we're starting off with a query plan just like in the earlier queries, from `sys.query_store_plan`. Then we're combining this with the table that has all the runtime metrics like average duration and standard deviation of the duration, `sys.query_store_runtime_stats`. Because I intend to filter based on a particular time, I want to be sure to join to the `sys.query_store_runtime_stats_interval` table where that data is stored. Then, I'm joining to the `sys.query_store_wait_stats`. There I have to use the compound key that directly links the waits and the runtime stats, the `plan_id`, the `execution_type`, and the `runtime_stats_interval_id`. I'm using a `plan_id` from earlier in the chapter, and I'm setting the data to return a particular time range. Figure 11-7 shows the resulting data.

(No column name)	count_executions	avg_duration	stdev_duration	wait_category_desc	avg_query_wait_time_ms	stdev_query_wait_time_ms
1	992	579.01310483871	530.744137979874	Memory	0.00604838709677419	0

**Figure 11-7.** Runtime metrics and wait statistics for one query in one time interval

It’s important to understand how the information in query\_store\_wait\_stats and query\_store\_runtime\_stats gets aggregated. It’s not simply by runtime\_stats\_interval\_id and plan\_id. The execution\_type also determines the aggregation because a given query may have an error or it could be canceled. This affects how the query behaves and the data is collected so it’s included in the performance metrics to differentiate one set of behaviors from another. Let’s see this by running the following script:

```
SELECT *
FROM sys.columns AS c,
     sys.syscolumns AS s;
```

That script results in a Cartesian join and takes about two minutes to run on my system. If we cancel the query while it’s running once and let it complete once, we can then see what’s in the Query Store.

```
SELECT qsqt.query_sql_text,
       qsrs.execution_type,
       qsrs.avg_duration
FROM sys.query_store_query AS qsq
     JOIN sys.query_store_query_text AS qsqt
         ON qsqt.query_text_id = qsq.query_text_id
     JOIN sys.query_store_plan AS qsp
         ON qsp.query_id = qsq.query_id
     JOIN sys.query_store_runtime_stats AS qsrs
         ON qsrs.plan_id = qsp.plan_id
WHERE qsqt.query_sql_text like '%FROM sys.columns AS c%';
```

You can see the results in Figure 11-8.

	query_sql_text	execution_type	avg_duration
1	SELECT * FROM sys.columns AS c, sys.sysco...	3	4800343
2	SELECT * FROM sys.columns AS c, sys.sysco...	0	122735336

**Figure 11-8.** Aborted execution shown as different execution type



You'll see aborted queries and queries that had errors showing different types. Also, their durations, waits, and so on, within the runtime metrics are stored separately. To get a proper set of waits and duration measures from the two respective tables, you must include the `execution_type`.

If you were interested in all the query metrics for a given query, you could retrieve the information from the Query Store with something like this:

```
WITH QSAggregate
AS (SELECT qsrs.plan_id,
          SUM(qsrs.count_executions) AS CountExecutions,
          AVG(qsrs.avg_duration) AS AvgDuration,
          AVG(qsrs.stdev_duration) AS StdDevDuration,
          qsws.wait_category_desc,
          AVG(qsws.avg_query_wait_time_ms) AS AvgWaitTime,
          AVG(qsws.stdev_query_wait_time_ms) AS StdDevWaitTime
FROM sys.query_store_runtime_stats AS qsrs
JOIN sys.query_store_wait_stats AS qsws
ON qsrs.plan_id = qsws.plan_id
AND qsws.runtime_stats_interval_id = qsrs.runtime_stats_
interval_id
GROUP BY qsrs.plan_id,
         qsws.wait_category_desc)
SELECT CAST(qsp.query_plan AS XML),
       qsa.*
FROM sys.query_store_plan AS qsp
JOIN QSAggregate AS qsa
ON qsa.plan_id = qsp.plan_id
WHERE qsq.object_id = OBJECT_ID('dbo.
ProductTransactionHistoryByReference');
```

The results of this query will be all the information currently contained within the Query Store for the `plan_id` specified. You can combine the information within the Query Store in any way you need going forward. Next, let's take control of the Query Store.

## Controlling the Query Store

You've already seen how to enable the Query Store for a database. To disable the Query Store, similar actions will work.

```
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE = OFF;
```

This command will disable the Query Store, but it won't remove the Query Store information. That data collected and managed by the Query Store will persist through reboots, failovers, backups, and the database going offline. It will even persist beyond disabling the Query Store. To remove the Query Store data, you have to take direct control like this:

```
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE CLEAR;
```

That will remove all data from the Query Store. You can get more selective if you want. You can simply remove a given query.

```
EXEC sys.sp_query_store_remove_query  
    @query_id = @queryid;
```

You can remove a query plan.

```
EXEC sys.sp_query_store_remove_plan @plan_id = @PlanID;
```

You can also reset the performance metrics.

```
EXEC sys.sp_query_store_reset_exec_stats  
    @plan_id = @PlanID;
```

All these simply require that you track down the plan or query in which you're interested in taking control of, and then you can do so. You may also find that you want to preserve the data in the Query Store that has been written to cache but not yet written to disk. You can force a flush of the cache.

```
EXEC sys.sp_query_store_flush_db;
```

Finally, you can change the default settings within the Query Store. First, it's a good idea to know where to go to get that information. You retrieve the current settings on the Query Store on a per-database basis by running the following:

```
SELECT * FROM sys.database_query_store_options AS dqso;
```

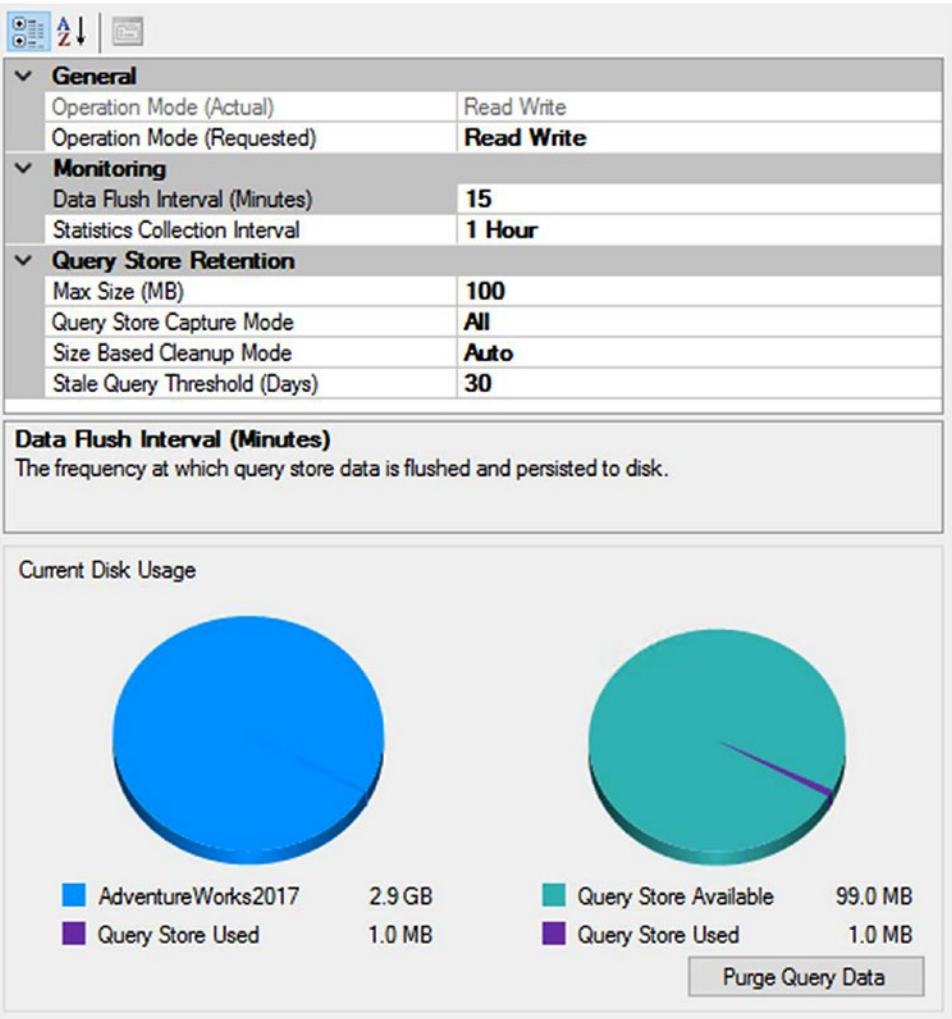
As with so many other aspects of the Query Store, these settings are controlled on a per-database level. This enables you to, for example, change the statistics aggregation time interval on one database and not another. Controlling the various aspects of the Query Store settings is simply a matter of running this query:

```
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE (MAX_STORAGE_SIZE_MB = 200);
```

That command changes the default storage size of the Query Store from 100MB to 200MB, allowing for more space in the database that was altered. When making these changes, no reboots of the server are required. You also won't affect the behavior of plans in the plan cache or any other part of the query processing within the database you are modifying. The default settings should be adequate for most people in most situations. Depending on your circumstances, you may want to modify the manner in which the Query Store behaves. Be sure that you monitor your servers when you make these changes to ensure that you haven't negatively impacted the server.

The only setting that I suggest you consider changing out of the box is the Query Store Capture Mode. By default, it captures all queries and all query plans, regardless of how often they are called, how long they run, or any other settings. For many of us, this behavior is adequate. However, if you have changed your system settings to use Optimize for Ad Hoc, you've done this because you get a lot of ad hoc queries and you're trying to manage memory use (more on this in Chapter 16). That setting means you're less interested in capturing every single plan. You may also be in the situation where because of the volume of transactions, you simply don't want to capture every single query or plan. These situations may lead you to change the Query Store Capture Mode setting. The other options are None and Auto. None will stop the Query Store from capturing queries and metrics but still allow for plan forcing if you set that for any queries (you'll find details on plan forcing later in this chapter). Auto will only capture queries that run for a certain length of time, consume a certain amount of resources, or get called a certain number of times. These values are all subject to change from Microsoft and are controlled internally within the Query Store. You can't control the values here, only whether they get used. On most systems, just to help reduce the noise and overhead, I recommend changing from All to Auto. However, this is absolutely an individual decision, and your situation may dictate otherwise.

You have the ability to take control of the Query Store using the SQL Server Management Studio GUI as well. Right-click any database in the Object Explorer window and from the context menu select Properties. When the Properties window opens, you can click the pane for Query Store and should see something similar to Figure 11-9.



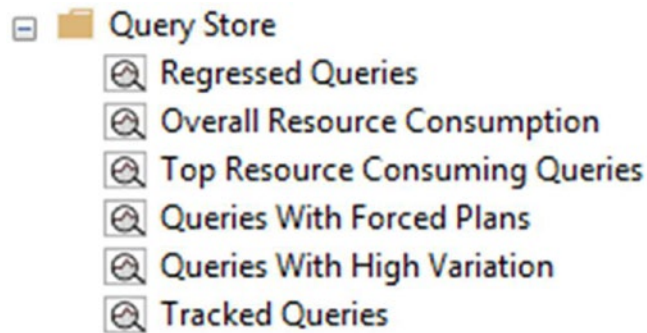
**Figure 11-9.** SSMS GUI for controlling the Query Store

Immediately you can see some of the settings that we’ve already covered in our exploration of the Query Store within this chapter. You also get to see just how much data the Query Store is using and how much room is left in the allocated space. As with using the T-SQL command shown earlier, any changes made here are immediately reflected in the Query Store behavior and will not require any sort of reboot of the system.

## Query Store Reporting

For some of your work, using T-SQL to take direct control over the Query Store and using the system tables to retrieve data about the Query Store is going to be the preferred approach. However, for a healthy percentage of the work, we can take advantage of the built-in reports and their behavior when working with the Query Store.

To see these reports, you just have to expand the database within the Object Explorer window in Management Studio. For any database with the Query Store enabled, there will be a new folder with the reports visible, as shown in Figure 11-10.



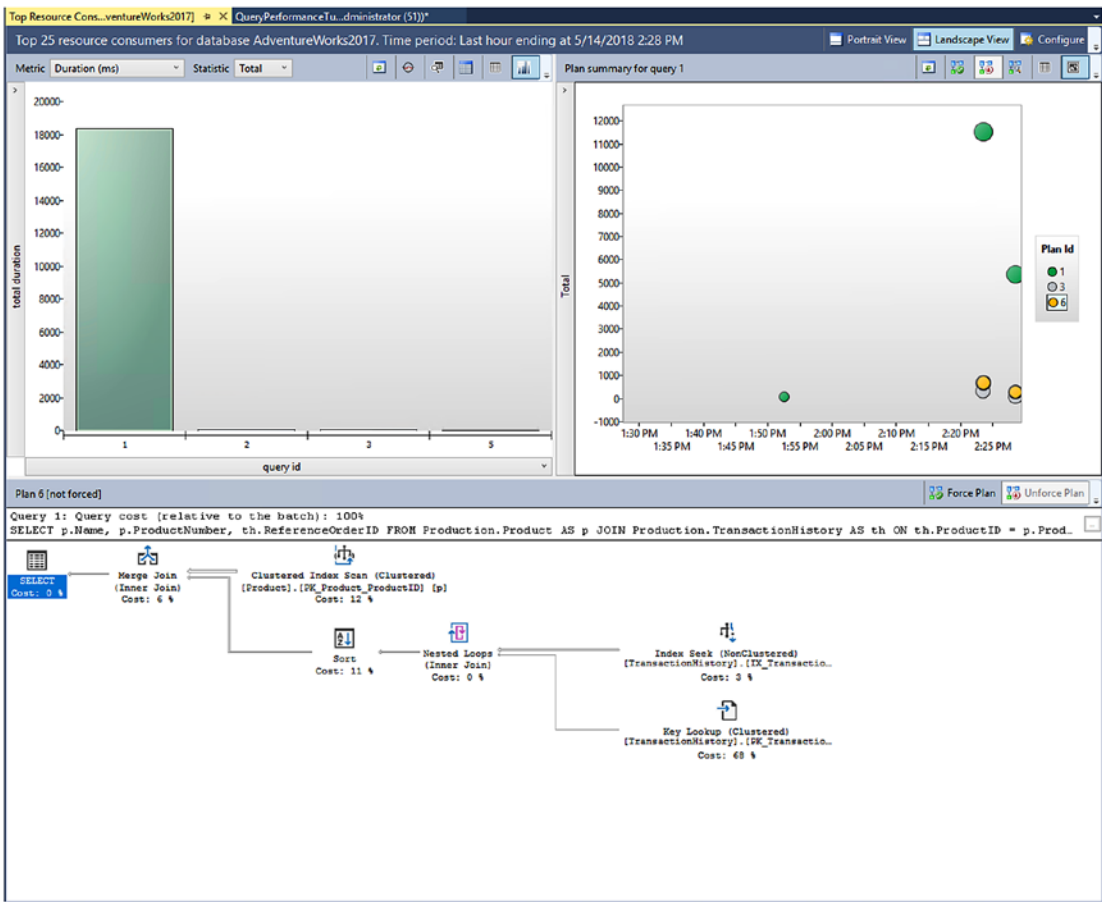
**Figure 11-10.** Query Store reports within the AdventureWorks2017 database

The reports are as follows:

- *Regressed Queries*: You'll see queries that have changed their performance in a negative way over time.
- *Overall Resource Consumption*: This report shows the resource consumption by various queries across a defined time frame. The default is the last month.
- *Top Resource Consuming Queries*: Here you find the queries that are using the most resources, without regard to a timeframe.
- *Query With Forced Plans*: Any queries that you have defined to have a forced plan will be visible in this report.

- *Queries With High Variation:* This report displays queries that have a high degree of variation in their runtime statistics, frequently with more than one execution plan.
- *Tracked Queries:* With the Query Store, you can define a query as being of interest and instead of having to attempt to track it down in the other reports, you can mark the query and find it here.

Each of these reports is unique, and each one is useful for differing purposes, but we don’t have the time and space to explore them all in detail. Instead, I’ll focus on the behavior of one, Top Resource Consuming Queries, because it generally represents the behavior of all the others and because it’s one that you’re likely to use fairly frequently. Opening the report, you’ll see something similar to Figure 11-11.



**Figure 11-11.** Top 25 Resource Consumers report for the Query Store

There are three windows in the report. The first in the upper left shows queries, aggregated by the `query_id` value. The second window on the right shows the various query behaviors over time as well as the different plans for those queries. You can see that the number-one query, highlighted in the first pane, has three different execution plans. Clicking any one of those plans opens that plan in the third window on the bottom of the screen.

You're not limited to the default behavior. The first window, showing the queries aggregated by Duration by default, drives the other two. You have a drop-down at the top of the screen that gives you 13 choices currently, as shown in Figure 11-12.

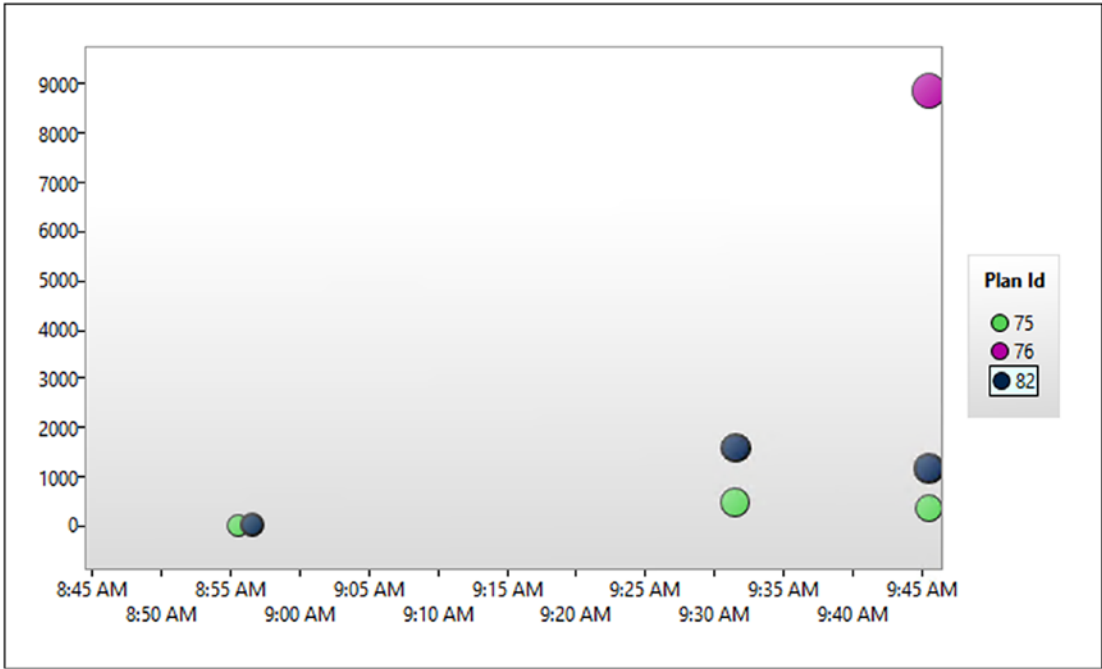
Execution Count
Duration (ms)
CPU Time (ms)
Logical Reads (KB)
Logical Writes (KB)
Physical Reads (KB)
CLR Time (ms)
DOP
Memory Consumption (KB)
Row Count
Log Memory Used (KB)
Temp DB Memory Used (KB)
Wait Time (ms)

**Figure 11-12.** *Different aggregations for the Top 25 Resource Consumers report*

Selecting any one of them will change the values being aggregated for the report. You can also change how the report is aggregated using another drop-down. This list includes, average, minimum, maximum, total, and standard deviation. Additional functionality for the first window includes the ability to change to a grid format, mark a query for tracking later (in the Tracked Queries report), refresh the report, and look at the query text. All this is useful in attempting to identify the query to spend time with when you are working to determine performance issues.



The next window shows the performance metrics from those selected in the first window. Each dot represents both a moment in time and a particular execution plan. The information in Figure 11-13 illustrates how query performance varied from 8:45 a.m. to 9:45 a.m. and how the query’s performance and execution plans changed over that time frame.



**Figure 11-13.** Different performance behaviors and different execution plans for one query

The size of each of the dots corresponds to the number of executions of the given plan within the given time frame. If you hover over any given dot, it will show you additional information about that moment in time. Figure 11-14 shows the information about the dot at the top of the screen, plan\_id = 76, at the 9:45 a.m. time frame.

<b>Plan Id</b>	76
<b>Execution Type</b>	Completed
<b>Plan Forced</b>	No
<b>Interval Start</b>	2017-12-08 09:45:00.000 -08:00
<b>Interval End</b>	2017-12-08 09:46:00.000 -08:00
<b>Execution Count</b>	63591
<b>Total Duration (ms)</b>	8871.63
<b>Avg Duration (ms)</b>	0.14
<b>Min Duration (ms)</b>	0.08
<b>Max Duration (ms)</b>	17.6
<b>Std Dev Duration (ms)</b>	0.25
<b>Variation Duration (ms)</b>	1.8

**Figure 11-14.** Details of the information on display for a given plan

You can see the number of executions and other metrics about that particular plan for the query in question. Whichever dot you click, you'll see the execution plan for that dot in the final window. The execution plans shown function like any other graphical plan within Management Studio, so I won't detail the behavior here. One additional piece of functionality that is on display here is the ability to force a plan. You'll see two buttons in the upper right of the execution plan window, as shown in Figure 11-15.



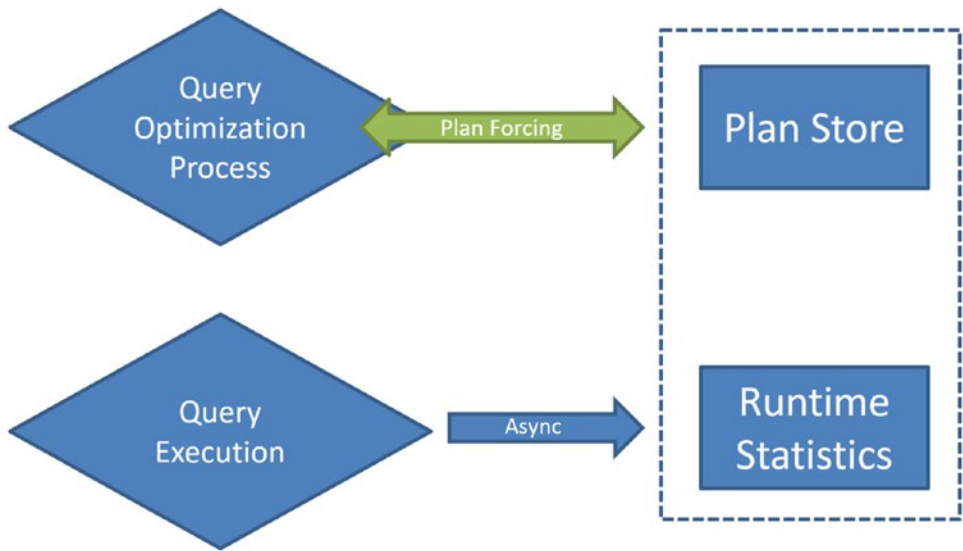
**Figure 11-15.** Forcing and unforcing plans from the reports

You have the ability to force, or unforce, a plan directly from the report. I'll cover plan forcing in detail in the next section.

## Plan Forcing

While the majority of the functionality around the Query Store is all about collecting and observing the behavior of the queries and the query plans, one piece of functionality changes all that, plan forcing. *Plan forcing* is where you mark a particular plan as being the plan you would like SQL Server to use. Since everything within the Query Store is

written to the database and so survives reboots, backups, and so on, this means you can ensure that a given plan will always be used. This process does change somewhat how the Query Store interacts with the optimization process and the plan cache, as illustrated in Figure 11-16.



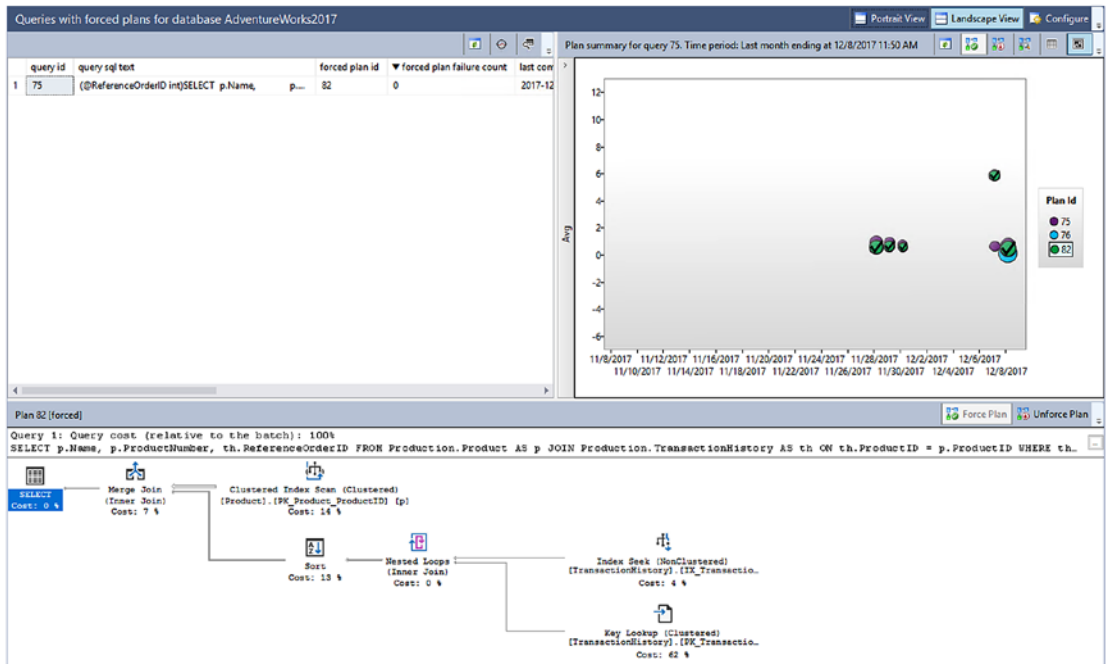
**Figure 11-16.** *The query optimization process with plan forcing added*

What happens now is that if a plan has been marked as being forced, when the optimizer completes its process, before it stores the plan in cache for use with the query, it first checks with the plans in the Query Store. If this query has a forced plan, that plan will always be used instead. The only exception to this is if something has changed internally in the system to make that plan an invalid plan for the query.

The function of plan forcing is actually quite simple. You have to supply a `plan_id` and a `query_id`, and you can force a plan. For example, my system has three possible plans for the query whose syntax, query hash, and query settings match the `query_id` value of 75. Note, while I’m using the `query_id` to mark a query, that’s an artificial key. The identifying factors of a query are the text, the hash, and the context settings. The query to force a plan is then extremely simple.

```
EXEC sys.sp_query_store_force_plan 75,82;
```

That is all that is required. From this point forward, no matter if a query is recompiled or removed from cache, when the optimization process is complete, the plan that corresponds to the plan\_id of 82 will be used. With this in place, we can look at the Queries with Forced Plans report to see what gets displayed, as shown in Figure 11-17.



**Figure 11-17.** *Queries with Forced Plans report*

You can see that while overall this report is the same as the Top Resource Consuming Queries report, there are differences. The listing of queries in the first window is just that, a listing of the queries. The second window corresponds almost exactly with the previous window on display in Figures 11-11 and 11-13. However, the difference is, the plan that has been marked as being forced has a check mark in place. The final window is the same with one minor difference. At the top, instead of Force Plan being enabled, Unforce Plan is. You can easily unforce the plan from here by clicking that button. You can also unforce a plan with a single command.

```
EXEC sys.sp_query_store_unforce_plan 214,248;
```

Just as with clicking the button, this will stop the plan forcing. From this point forward, the optimization process goes back to normal. I'm going to save a full demonstration of plan forcing until we get to Chapter 17 when we talk about parameter

sniffing. Suffice to say that plan forcing becomes extremely useful when dealing with bad parameter sniffing. It also is handy when dealing with regressions, situations where the changes to SQL Server cause previously well-behaving queries to suddenly generate badly performing execution plans. This most often occurs during an upgrade when the compatibility mode gets changed without testing.

## Query Store for Upgrades

While general query performance monitoring and tuning may be a day-to-day common use for the Query Store, one of the most powerful purposes behind the tool is its use as a safety net for upgrading SQL Server.

Let's assume you are planning to migrate from SQL Server 2012 to SQL Server 2017. Traditionally you would upgrade your database on a test instance somewhere and then run a battery of tests to ensure that things are going to work well. If you catch and document all the issues, great. Unfortunately, it might require some code rewrites because of some of the changes to the optimizer or the cardinality estimator. That could cause delays to the upgrade, or the business might even decide to try to avoid it altogether (a frequent, if poor, choice). That assumes you catch the issues. It's entirely possible to miss that a particular query has suddenly started behaving poorly because of changes in estimated row counts or something else.

This is where the Query Store becomes your safety net for upgrades. First, you should do all the testing and attempt to address issues using standard methods. That shouldn't change. However, the Query Store adds additional functionality to the standard methods. Here are the steps to follow:

1. Restore your database to the new SQL Server instance or upgrade your instance. This assumes the production machine, but you can do this with a test machine as well.
2. Leave the database in the older compatibility mode. Do not change it to the new mode because you will enable both the new optimizer and the new cardinality estimator before you capture data.
3. Enable the Query Store. It can gather metrics running in the old compatibility mode.

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

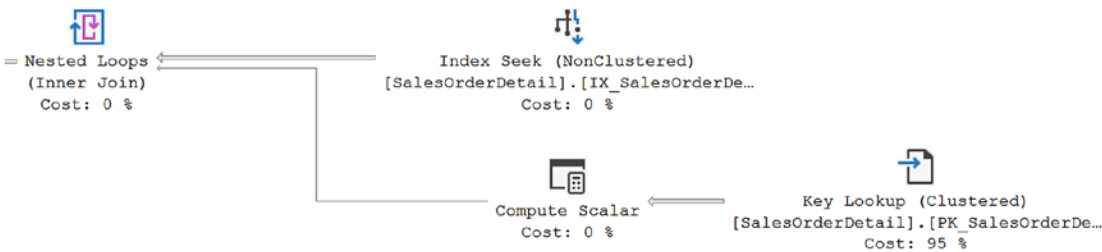
## Purpose of Lookups

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.

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:

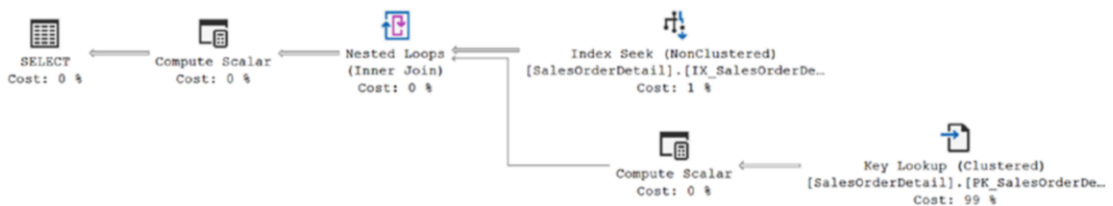
```
SELECT *
FROM Sales.SalesOrderDetail AS sod
WHERE sod.ProductID = 776 ;
```

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:

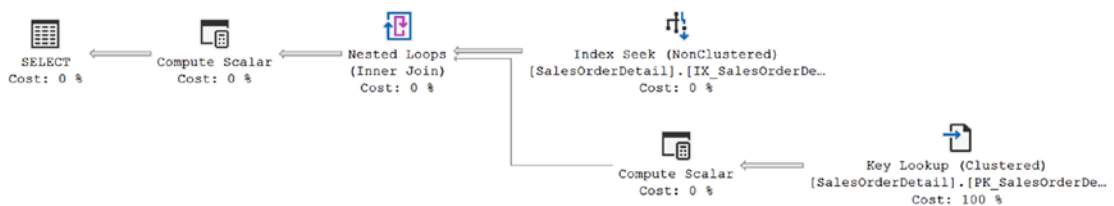
```
SELECT *
FROM Sales.SalesOrderDetail AS sod WITH (INDEX (IX_SalesOrderDetail_
      ProductID))
WHERE sod.ProductID = 793 ;
```

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

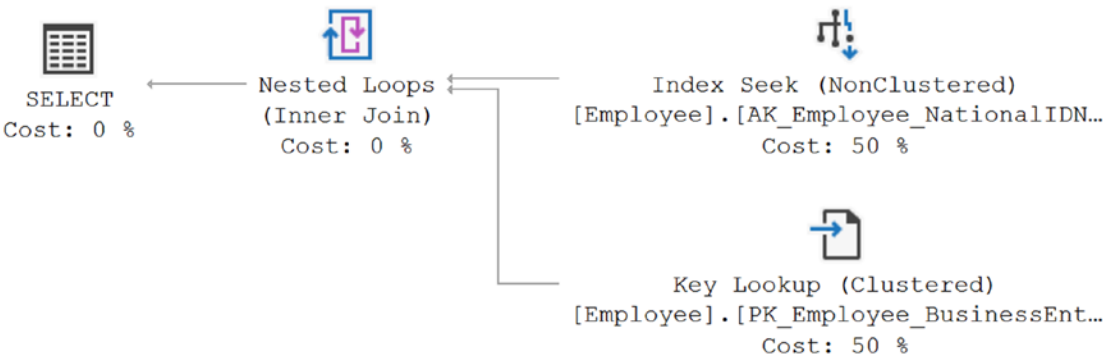
Since a lookup can be a costly operation, you should analyze what causes a query plan to choose a lookup step in an execution plan. You may find that you are able to avoid the lookup by including the missing columns in the nonclustered index key or as `INCLUDE` columns at the index page level and thereby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

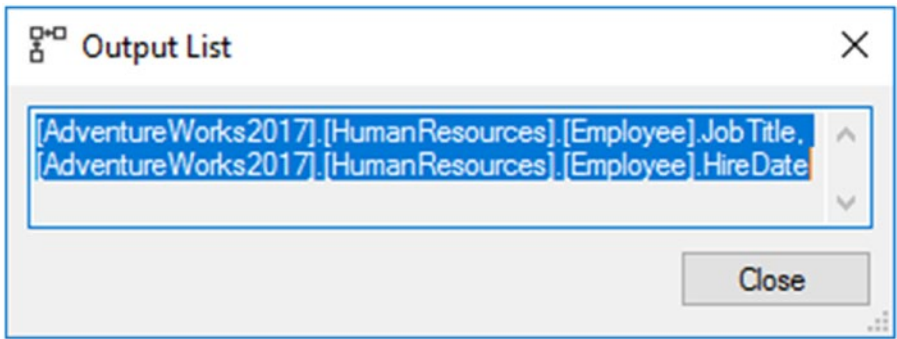
In the case of a complex query based on views and user-defined functions, it may be too difficult to find all the columns referred to by the query. As a result, you need a standard mechanism to find the columns returned by the lookup that are not included in the nonclustered index.

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 NationalIDNumber. 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.

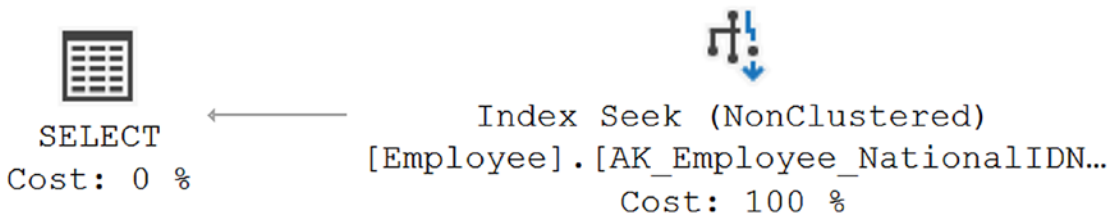
```
SELECT  NationalIDNumber,
        JobTitle,
        HireDate
FROM    HumanResources.Employee AS e
WHERE   e.NationalIDNumber = '693168613';
```

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.

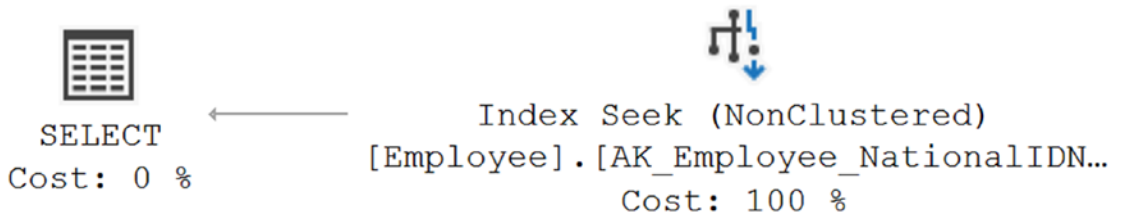
Another way to arrive at the covering index, without reshaping the index by adding key columns, is to use the INCLUDE columns. Change the index to look like this:

```
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):

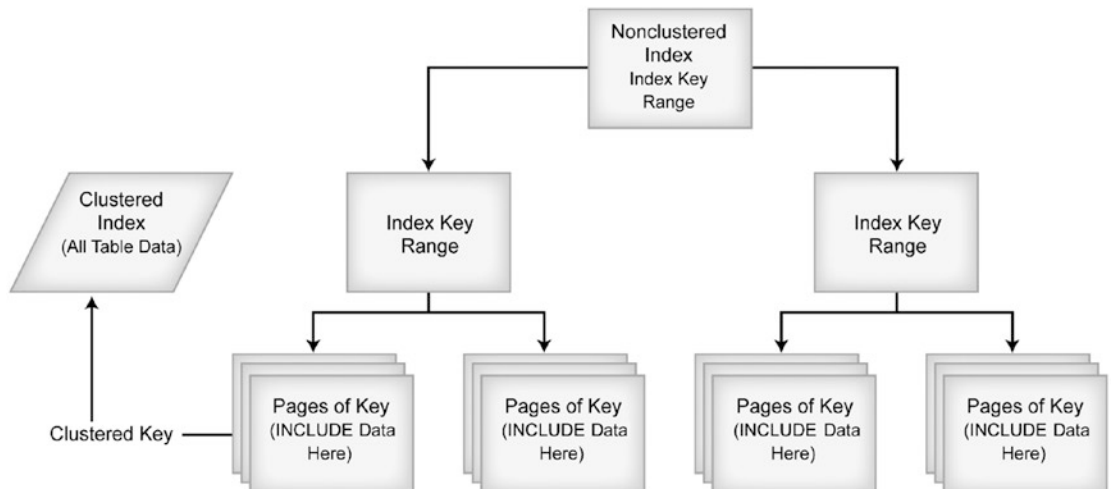
Duration: 152mc

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

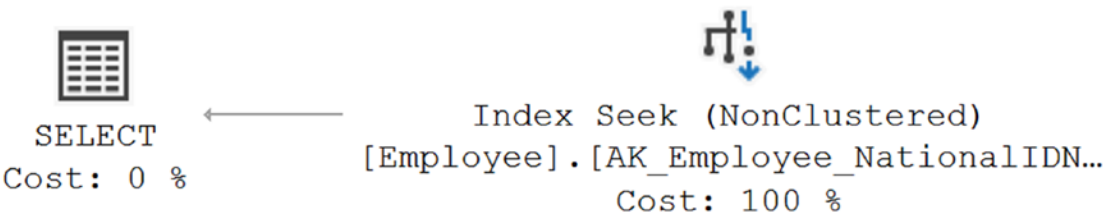
Another way to get a covering index is to take advantage of the structures within SQL Server. If the previous query were modified slightly to retrieve a different set of data instead of a particular NationalIDNumber and its associated JobTitle and HireDate, this time the query would retrieve the NationalIDNumber as an alternate key and the BusinessEntityID, 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 BusinessEntityID 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 BusinessEntityID 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	13000049	1	1	1	1						
5	13749773	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	136999999	1	1	1	1						

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';
```