```
CREATE OR ALTER PROCEDURE dbo.PersonByFirstName @FirstName NVARCHAR(50)
AS
--gets anyone by first name from the Person table
SELECT p.BusinessEntityID,
       p.Title,
       p.LastName,
       p.FirstName,
       p.PersonType
FROM Person.Person AS p
WHERE p.FirstName = @FirstName;
GO

CREATE OR ALTER PROCEDURE dbo.ProductTransactionsSinceDate
    @LatestDate DATETIME,
    @ProductName NVARCHAR(50)
AS
--Gets the latest transaction against
--all products that have a transaction
SELECT p.Name,
       th.ReferenceOrderID,
       th.ReferenceOrderLineID,
       th.TransactionType,
       th.Quantity
FROM Production.Product AS p
    JOIN Production.TransactionHistory AS th
        ON p.ProductID = th.ProductID
           AND th.TransactionID = (   SELECT TOP (1)
                                             th2.TransactionID
                                      FROM Production.TransactionHistory AS
th2
                                      WHERE th2.ProductID = p.ProductID
                                      ORDER BY th2.TransactionID DESC)
WHERE th.TransactionDate > @LatestDate
      AND p.Name LIKE @ProductName;
GO
```

```
CREATE OR ALTER PROCEDURE dbo.PurchaseOrderBySalesPersonName
    @LastName NVARCHAR(50),
    @VendorID INT = NULL
AS
SELECT poh.PurchaseOrderID,
       poh.OrderDate,
       pod.LineTotal,
       p.Name AS ProductName,
       e.JobTitle,
       per.LastName + ', ' + per.FirstName AS SalesPerson,
       poh.VendorID
FROM Purchasing.PurchaseOrderHeader AS poh
    JOIN Purchasing.PurchaseOrderDetail AS pod
        ON poh.PurchaseOrderID = pod.PurchaseOrderID
    JOIN Production.Product AS p
        ON pod.ProductID = p.ProductID
    JOIN HumanResources.Employee AS e
        ON poh.EmployeeID = e.BusinessEntityID
    JOIN Person.Person AS per
        ON e.BusinessEntityID = per.BusinessEntityID
WHERE per.LastName LIKE @LastName
      AND poh.VendorID = COALESCE(@VendorID,
                                  poh.VendorID)
ORDER BY per.LastName,
         per.FirstName;
GO

CREATE OR ALTER PROCEDURE dbo.TotalSalesByProduct @ProductID INT
AS
--retrieve aggregation of sales based on a productid
SELECT SUM((isnull((sod.UnitPrice*((1.0)-sod.UnitPriceDiscount))*sod.
OrderQty,(0.0)))) AS TotalSales,
    AVG(sod.OrderQty) AS AverageQty,
    AVG(sod.UnitPrice) AS AverageUnitPrice,
    SUM(sod.LineTotal)
FROM Sales.SalesOrderDetail AS sod
```

```
WHERE sod.ProductID = @ProductID
GROUP BY sod.ProductID;
GO
```

Please remember that this is just meant to be an illustrative example, not a literal and real load placed on a server. Real procedures are generally much more complex, but there's only so much space we can devote to setting up a simulated production load. With these procedures in place, you can execute them using the following script:

```
EXEC dbo.PurchaseOrderBySalesPersonName @LastName = 'Hill%';
GO
EXEC dbo.ShoppingCart @ShoppingCartId = '20621';
GO
EXEC dbo.ProductBySalesOrder @SalesOrderID = 43867;
GO
EXEC dbo.PersonByFirstName @FirstName = 'Gretchen';
GO
EXEC dbo.ProductTransactionsSinceDate @LatestDate = '9/1/2004',
                                      @ProductName = 'Hex Nut%';
GO
EXEC dbo.PurchaseOrderBySalesPersonName @LastName = 'Hill%',
                                        @VendorID = 1496;
GO
EXEC dbo.TotalSalesByProduct @ProductID = 707;
GO
```

I know I'm repeating myself, but I want to be clear. This is an extremely simplistic workload that just illustrates the process. You're going to see hundreds and thousands of additional calls across a much wider set of procedures and ad hoc queries in a typical system. As simple as it is, however, this sample workload consists of the different types of queries you usually execute on SQL Server.

- Queries using aggregate functions
- Point queries that retrieve only one row or a small number of rows
- Queries joining multiple tables

- Queries retrieving a narrow range of rows
- Queries performing additional result set processing, such as providing a sorted output

The first optimization step is to capture the workload, meaning see how these queries are performing, as explained in the next section.

Capturing the Workload

As part of the diagnostic-data collection step, you must define an Extended Events session to capture the workload on the database server. You can use the tools and methods recommended in Chapter 6 to do this. Table 27-1 lists the specific events you can use to measure how many resources your queries use.

Table 27-1. *Events to Capture Information About Costly Queries*

Category	Event
Execution	`rpc_completed` `sql_batch_completed`

As explained in Chapter 6, for production databases it is recommended that you capture the output of the Extended Events session to a file. Here are a couple significant advantages to capturing output to a file:

- Since you intend to analyze the SQL queries once the workload is captured, you do not need to display the SQL queries while capturing them.
- Running the session through SSMS doesn't provide a flexible timing control over the tracing process.

Let's look at the timing control more closely. Assume you want to start capturing events at 11 p.m. and record the SQL workload for 24 hours. You can define an Extended Events session using the GUI or T-SQL. However, you don't have to start the process until you're ready. This means you can create commands in SQL Agent or with some other scheduling tool to start and stop the process with the `ALTER EVENT SESSION` command.

```
ALTER EVENT SESSION <sessionname>
ON SERVER
STATE = <start/stop>;
```

For this example, I've put a filter on the session to capture events only from the AdventureWorks2017 database. The file will capture queries against only that database, reducing the amount of information I need to deal with. This may be a good choice for your systems, too. While Extended Events sessions can be very low cost, especially when compared to the older trace events, they are not free. Good filtering should always be applied to ensure minimum impact.

Analyzing the Workload

Once the workload is captured in a file, you can analyze the workload either by browsing through the data using SSMS or by importing the content of the output file into a database table.

SSMS provides the following two methods for analyzing the content of the file, both of which are relatively straightforward:

- *Sort the output on a data column by right-clicking to select a sort order or to group by a particular column*: You may want to select columns from the Details tab and use the "Show column in table" command to move them up. Once there, you can issue grouping and sorting commands on that column.
- *Rearrange the output to a selective list of columns and events*: You can change the output displayed through SSMS by right-clicking the table and selecting Pick Columns from the context menu. This lets you do more than simply pick and choose columns; it also lets you combine them into new columns.

As I've shown throughout the book, the Live Data Explorer within SSMS when used with Extended Events can be used to put together basic aggregations. For example, if you wanted to group by the text of queries or the object ID and then get the average duration or a count of the number of executions, you can. In fact, SSMS is an way to do this type of simpler aggregation.

If, on the other hand, you want to do an in-depth analysis of the workload, you must import the content of the trace file into a database table. Then you can create much more complex queries. The output from the session puts most of the important data into an XML field, so you'll want to query it as you load the data as follows:

```
DROP TABLE IF EXISTS dbo.ExEvents;
GO
WITH xEvents
AS (SELECT object_name AS xEventName,
           CAST(event_data AS XML) AS xEventData
    FROM sys.fn_xe_file_target_read_file('C:\PerfData\QueryPerfTuning2017*.xel',
                                         NULL,
                                         NULL,
                                         NULL) )
SELECT xEventName,
       xEventData.value('(/event/data[@name="duration"]/value)[1]',
                        'bigint') AS Duration,
       xEventData.value('(/event/data[@name="physical_reads"]/value)[1]',
                        'bigint') AS PhysicalReads,
       xEventData.value('(/event/data[@name="logical_reads"]/value)[1]',
                        'bigint') AS LogicalReads,
       xEventData.value('(/event/data[@name="cpu_time"]/value)[1]',
                        'bigint') AS CpuTime,
       CASE xEventName
           WHEN 'sql_batch_completed' THEN
               xEventData.value('(/event/data[@name="batch_text"]/value)[1]',
                                'varchar(max)')
           WHEN 'rpc_completed' THEN
               xEventData.value('(/event/data[@name="statement"]/value)[1]',
                                'varchar(max)')
       END AS SQLText,
       xEventData.value('(/event/data[@name="query_plan_hash"]/value)[1]',
                        'binary(8)') AS QueryPlanHash
INTO dbo.ExEvents
FROM xEvents;
```

You need to substitute your own path and file name for <ExEventsFileName>. Once you have the content in a table, you can use SQL queries to analyze the workload. For example, to find the slowest queries, you can execute this SQL query:

```
SELECT  *
FROM    dbo.ExEvents AS ee
ORDER BY ee.Duration DESC;
```

The preceding query will show the single costliest query, and it is adequate for the tests you're running in this chapter. You may also want to run a query like this on a production system; however, it's more likely you'll want to work from aggregations of data, as in this example:

```
SELECT  ee.SQLText,
        SUM(Duration) AS SumDuration,
        AVG(Duration) AS AvgDuration,
        COUNT(Duration) AS CountDuration
FROM    dbo.ExEvents AS ee
GROUP BY ee.SQLText;
```

Executing this query lets you order things by the fields you're most interested in—say, CountDuration to get the most frequently called procedure or SumDuration to get the procedure that runs for the longest cumulative amount of time. You need a method to remove or replace parameters and parameter values. This is necessary to aggregate based on just the procedure name or just the text of the query without the parameters or parameter values (since these will be constantly changing).

Another mechanism is to simply query the cache to see the costliest queries through there. It is easier than setting up Extended Events. Further, you'll probably capture most of the bad queries most of the time. Because of this, if you're just getting started with query tuning your system for the first time, you may want to skip setting up Extended Events to identify the costliest queries. However, I've found that as time goes on and you begin to quantify your systems behaviors, you're going to want the kind of detailed data that using Extended Events provides.

One more method we have already explored in the book is using the Query Store to gather metrics on the behavior of the queries in your system. It has the benefits of being extremely easy to set up and easy to query, with no XML involved. The only detriment is if you need granular and detailed performance metrics on individual calls to queries and

procedures. In that case, again, you'll find yourself calling on Extended Events to satisfy the need for that type of data.

In short, you have a lot of choices and flexibility in how you put this information together. With SQL Server 2016 and SQL Server 2017, you can even start putting data analysis using R or Python to work to enhance the information presented. For our purposes, though, I'll stick with the first method I outlined, using Live Data within SSMS.

The objective of analyzing the workload is to identify the costliest query (or costly queries in general); the next section covers how to do this.

Identifying the Costliest Query

As just explained, you can use SSMS or the query technique to identify costly queries for different criteria. The queries in the workload can be sorted on the `CPU`, `Reads`, or `Writes` column to identify the costliest query, as discussed in Chapter 3. You can also use aggregate functions to arrive at the cumulative cost, as well as individual costs. In a production system, knowing the procedure that is accumulating the longest run times, the most CPU usage, or the largest number of reads and writes is frequently more useful than simply identifying the query that had the highest numbers one time.

Since the total number of reads usually outnumbers the total number of writes by at least seven to eight times for even the heaviest OLTP database, sorting the queries on the `Reads` column usually identifies more bad queries than sorting on the `Writes` column (but you should always test this on your systems). It's also worth looking at the queries that simply take the longest to execute. As outlined in Chapter 5, you can capture wait states with Performance Monitor and view those along with a given query to help identify why a particular query is taking a long time to run. You can also capture specific waits for a given query using Extended Events and add that to your calculations. Each system is different. In general, I approach the most frequently called procedures first, then the longest-running, and, finally, those with the most reads. Of course, performance tuning is an iterative process, so you will need to reexamine each category on a regular basis.

To analyze the sample workload for the worst-performing queries, you need to know how costly the queries are in terms of duration or reads. Since these values are known only after the query completes its execution, you are mainly interested in the completed events. (The rationale behind using completed events for performance analysis is explained in detail in Chapter 6.)

For presentation purposes, open the trace file in SSMS. Figure 27-1 shows the captured trace output after moving several columns to the grid and then choosing to sort by duration by clicking that column (twice to get the sort to be descending instead of ascending).

timestamp	batch_text	duration	logical_reads	cpu_time
2018-05-09 14:54:53.3291321	EXEC dbo.PurchaseOrderBySalesPersonNam...	464194	8671	62000
2018-05-09 14:54:53.6476233	EXEC dbo.ProductTransactionsSinceDate @...	86095	1044	63000
2018-05-09 14:54:53.5260951	EXEC dbo.PersonByFirstName @FirstName =...	46332	219	15000
2018-05-09 14:54:53.7761907	EXEC dbo.TotalSalesByProduct @ProductID ...	33239	1264	16000
2018-05-09 14:54:53.4304814	EXEC dbo.ProductBySalesOrder @SalesOrde...	19406	279	16000
2018-05-09 14:54:53.3765471	EXEC dbo.ShoppingCart @ShoppingCartId = ...	13283	66	0
2018-05-09 14:54:53.6994256	EXEC dbo.PurchaseOrderBySalesPersonNam...	6345	180	0

Figure 27-1. *Extended Events session output showing the SQL workload*

The worst-performing query in terms of duration is also one of the worst in terms of CPU usage and reads. That procedure, `dbo.PurchaseOrderBySalesPersonName`, is at the top in Figure 27-1 (you may have different values, but this query is likely to be the worst-performing query or at least one of the worst of the example queries).

Once you've identified the worst-performing query, the next optimization step is to determine the resources consumed by the query.

Determining the Baseline Resource Use of the Costliest Query

The current resource use of the worst-performing query can be considered as a baseline figure before you apply any optimization techniques. You may apply different optimization techniques to the query, and you can compare the resultant resource use of the query with the baseline figure to determine the effectiveness of a given optimization technique. The resource use of a query can be presented in two categories.

- Overall resource use
- Detailed resource use

Overall Resource Use

The overall resource use of the query provides a gross figure for the amount of hardware resources consumed by the worst-performing query. You can compare the resource use of an optimized query to the overall resource use of a nonoptimized query to ensure the overall effectiveness of the performance techniques you've applied.

You can determine the overall resource use of the query from the workload trace. You'll use the first call of the procedure since it displays the worst behavior. Table 27-2 shows the overall use of the query from the trace in Figure 27-1. One point, the durations in the table are in milliseconds, while the values in Figure 27-1 are in microseconds. Remember to take this into account when working with Extended Events.

Table 27-2. *Data Columns Representing the Amount of Resources Used by a Query*

Data Column	Value	Description
`LogicalReads`	8671	Number of logical reads performed by the query. If a page is not found in memory, then a logical read for the page will require a physical read from the disk to fetch the page to the memory first.
`Writes`	0	Number of pages modified by the query.
`CPU`	62ms	How long the CPU was used by the query.
`Duration`	464.1ms	The time it took SQL Server to process this query from compilation to returning the result set.

Note In your environment, you may have different figures for the preceding data columns. Irrespective of the data columns' absolute values, it's important to keep track of these values so that you can compare them with the corresponding values later.

Detailed Resource Use

You can break down the overall resource use of the query to locate bottlenecks on the different database objects accessed by the query. This detailed resource use helps you determine which operations are the most problematic. Understanding the wait states in your system will help you identify where you need to focus your tuning. A rough rule of thumb can be to simply look at duration; however, duration can be affected by so many factors, especially blocking, that it's an imperfect measure at best. In this case, I'll spend time on all three: CPU usage, reads, and duration. Reads are a popular measure of performance, but they can be as problematic to look at in isolation as duration. This is why I prefer to capture multiple values and have the ability to compare them across iterations of the query.

As you saw in Chapter 6, you can obtain the number of reads performed on the individual tables accessed by a given query from the `STATISTICS IO` output for that query. You can also set the `STATISTICS TIME` option to get the basic execution time and CPU time for the query, including its compile time. You can obtain this output by reexecuting the query with the `SET` statements as follows (or by selecting the Set Statistics IO check box in the query window):

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
DBCC DROPCLEANBUFFERS;
GO
SET STATISTICS TIME ON;
GO
SET STATISTICS IO ON;
GO
EXEC dbo.PurchaseOrderBySalesPersonName @LastName = 'Hill%';
GO
SET STATISTICS TIME OFF;
GO
SET STATISTICS IO OFF;
GO
```

To simulate the same first-time run shown in Figure 27-1, clean out the data stored in memory using `DBCC DROPCLEANBUFFERS` (not to be run on a production system) and remove the queries from the specified database from the cache by using the

database-scoped configuration command CLEAR PROCEDURE_CACHE (also not to be run on a production system).

The STATISTICS output for the worst-performing query looks like this:

```
DBCC execution completed. If DBCC printed error messages, contact your
system administrator.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 31 ms, elapsed time = 40 ms.

(1496 rows affected)
Table 'Employee'. Scan count 0, logical reads 2992, physical reads 2, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'Product'. Scan count 0, logical reads 2992, physical reads 4, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'PurchaseOrderDetail'. Scan count 763, logical reads 1539, physical reads
9, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead
reads 0.
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'PurchaseOrderHeader'. Scan count 1, logical reads 44, physical reads
1, read-ahead reads 42, lob logical reads 0, lob physical reads 0, lob
read-ahead reads 0.
Table 'Person'. Scan count 1, logical reads 4, physical reads 1, read-ahead
reads 2, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

 SQL Server Execution Times:
   CPU time = 15 ms,  elapsed time = 93 ms.
```

```
 SQL Server Execution Times:
   CPU time = 46 ms,  elapsed time = 133 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

One caveat that is worth mentioning is that there is some overhead to return this information along with the data, and it will impact some of your performance metrics including the duration measure of the query. For most of us, most of the time, it's not a problem, but sometimes it's noticeably causing issues. Be aware that by capturing information in this way, you are making a choice.

Table 27-3 summarizes the output of `STATISTICS IO`.

Table 27-3. *Breaking Down the Output from STATISTICS IO*

Table	Logical Reads
`Person.Employee`	2,992
`Production.Product`	2,992
`Purchasing.PurchaseOrderDetail`	1,539
`Purchasing.PurchaseOrderHeader`	44
`Person.Person`	4

Usually, the sum of the reads from the individual tables referred to in a query will be less than the total number of reads performed by the query. This is because additional pages have to be read to access internal database objects, such as `sysobjects`, `syscolumns`, and `sysindexes`.

Table 27-4 summarizes the output of `STATISTICS TIME`.

Table 27-4. *Breaking Down the Output from STATISTICS TIME*

Event	Duration	CPU
`Compile`	40 ms	31 ms
`Execution`	93 ms	15 ms
`Completion`	133 ms	46 ms

Don't use the logical reads in isolation from the execution times. You need to take all the measures into account when determining poorly performing queries. Conversely, don't assume that the execution time is a perfect measure, either. Resource contention plays a big part in execution time, so you'll see some variation in this measure. Use both values, but use them with a full understanding that either in isolation may not be an accurate reflection of reality.

You can also add additional metrics to these details. As I outlined in Chapters 2–4, wait statistics are an important measure to understand what's happening on your system. The same thing applies to queries. In SQL Server 2016 and newer, you can see waits that are over 1ms when you capture an actual execution plan. That information is in the properties of the `SELECT` operator in the query execution plan. You can also use Extended Events to capture wait statistics for a given query, which will show all the waits, not just the ones that exceed 1ms. These are useful additions to the detailed metrics for measuring your query's performance.

Once the worst-performing query has been identified and its resource use has been measured, the next optimization step is to determine the factors that are affecting the performance of the query. However, before you do this, you should check to see whether any factors external to the query might be causing that poor performance.

Analyzing and Optimizing External Factors

In addition to factors such as query design and indexing, external factors can affect query performance. Thus, before diving into the execution plan of the query, you should analyze and optimize the major external factors that can affect query performance. Here are some of those external factors:

- The connection options used by the application
- The statistics of the database objects accessed by the query
- The fragmentation of the database objects accessed by the query

Analyzing the Connection Options Used by the Application

When making a connection to SQL Server, various options, such as `ANSI_NULL` or `CONCAT_NULL_YIELDS_NULL`, can be set differently than the defaults for the server or the database. However, changing these settings per connection can lead to recompiles of

stored procedures, causing slower behavior. Also, some options, such as `ARITHABORT`, must be set to `ON` when dealing with indexed views and certain other specialized indexes. If they are not, you can get poor performance or even errors in the code. For example, setting `ANSI_WARNINGS` to `OFF` will cause the optimizer to ignore indexed views and indexed computed columns when generating the execution plan. You can look to the properties of the execution plans again for this information. When an execution plan is created, the ANSI settings are stored with it. So, if you query the cache to look at a plan and retrieve it from the Query Store, capturing using Extended Events or SSMS, you'll have the ANSI settings at the time the plan was compiled. Further, if the same query is called and the ANSI settings are different from what is currently in cache, a new plan will be compiled (and stored in the Query Store alongside the other plan). The properties are in the `SELECT` operator, as shown in Figure 27-2.

Set Options	ANSI_NULL
ANSI_NULLS	True
ANSI_PADDING	True
ANSI_WARNINGS	True
ARITHABORT	True
CONCAT_NULL_YIELDS_NULL	True
NUMERIC_ROUNDABORT	False
QUOTED_IDENTIFIER	True

Figure 27-2. *Properties of the execution plan showing the Set Options properties*

I recommend using the ANSI standard settings, in which you set the following options to `TRUE`: `ANSI_NULLS`, `ANSI_NULL_DFLT_ON`, `ANSI_PADDING`, `ANSI_WARNINGS`, `CURSOR_CLOSE_ON_COMMIT`, `IMPLICIT_TRANSACTIONS`, and `QUOTED_IDENTIFIER`. You can use the single command `SET ANSI_DEFAULTS ON` to set them all to `TRUE` at the same time. Querying `sys.query_context_settings` is also a helpful way for seeing the history of settings used across workloads.

Analyzing the Effectiveness of Statistics

The statistics of the database objects referred to in the query are one of the key pieces of information that the query optimizer uses to decide upon certain execution plans. As explained in Chapter 13, the optimizer generates the execution plan for a query based on

the statistics of the objects referred to in the query. The number of rows that the statistics suggest is a major part of the cost estimation process that drives the optimizer. In this way, it determines the processing strategy for the query. If a database object's statistics are not accurate, then the optimizer may generate an inefficient execution plan for the query. Several problems can arise: you can have a complete lack of statistics because `auto_create` statistics have been disabled, out-of-date statistics because automatic updates are not enabled, outdated statistics because the statistics have simply aged, or inaccurate statistics because of data distribution or sampling size issues.

As explained in Chapter 13, you can check the statistics of a table and its indexes using `DBCC SHOW_STATISTICS` or `sys.dm_db_stats_properties` and `sys.dm_db_stats_histogram`. There are five tables referenced in this query: `Purchasing.PurchaseOrderHeader`, `Purchasing.PurchaseOrderDetail`, `Person.Employee`, `Person.Person`, and `Production.Product`. You must know which indexes are in use by the query to get the statistics information about them. You can determine this when you look at the execution plan. Specifically, you can now look to the execution plan to get the specific statistics used by the optimizer when building the execution plan. As with so much other interesting information, this is stored in the `SELECT` operator, as you can see in Figure 27-3.

OptimizerStatsUsage	
⊟ [1]	
Database	[AdventureWorks2017]
LastUpdate	10/27/2017 2:33 PM
ModificationCount	0
SamplingPercent	100
Schema	[Purchasing]
Statistics	[IX_PurchaseOrderDetail_ProductID]
Table	[PurchaseOrderDetail]
⊟ [2]	
Database	[AdventureWorks2017]
LastUpdate	10/27/2017 2:33 PM
ModificationCount	0
SamplingPercent	100
Schema	[HumanResources]
Statistics	[PK_Employee_BusinessEntityID]
Table	[Employee]
⊟ [3]	
Database	[AdventureWorks2017]
LastUpdate	10/27/2017 2:33 PM
ModificationCount	0
SamplingPercent	100
Schema	[Purchasing]
Statistics	[PK_PurchaseOrderDetail_PurchaseOrderID_PurchaseOrderDetailID]
Table	[PurchaseOrderDetail]
⊞ [4]	
⊞ [5]	
⊞ [6]	
⊞ [7]	

Figure 27-3. *Statistics used by the optimizer to create the execution plan for the query we're exploring*

While there are five tables, you can see that there were seven statistics objects used in generating the plan. As you can see, more than one object in the `PurchaseOrderDetail` table was used. You may see several different stats from any given table in use. This is a great way to easily identify the statistics of which you need to determine the efficiency.

For now, I'll check the statistics on the primary key of the `HumanResources.Employee` table since it had the most reads. Now run the following query:

```
DBCC SHOW_STATISTICS('HumanResources.Employee', 'PK_Employee_
BusinessEntityID');
```

When the preceding query completes, you'll see the output shown in Figure 27-4.

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	PK_Employee_BusinessEntityID	Oct 27 2017 2:33PM	290	290	146	1	4	NO	NULL	290	0

	All density	Average Length	Columns
1	0.003448276	4	BusinessEntityID

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	1	0	1
2	3	1	1	1	1
3	4	0	1	0	1
4	6	1	1	1	1
5	8	1	1	1	1
6	10	1	1	1	1
7	12	1	1	1	1
8	'	1	1	1	1

Figure 27-4. *SHOW_STATISTICS output for HumanResources.Employee*

You can see the selectivity on the index is very high since the density is quite low, as shown in the `All density` column. You can see that all rows were scanned in these statistics and that the distribution was in 146 steps. In this instance, it's doubtful that statistics are likely to be the cause of this query's poor performance. It's probably a good idea, where possible, to look at the actual execution plan and compare estimated versus actual rows there. You can also check the `Updated` column to determine the last time this set of statistics was updated. If it has been more than a few days since the statistics were updated, then you need to check your statistics maintenance plan, and you should update these statistics manually. That of course does depend on the frequency of data change within your database. In this case, these statistics could be seriously out-of-date considering the data provided (however, they're not because this is a sample database that has not been updated).

Analyzing the Need for Defragmentation

As explained in Chapter 14, a fragmented table increases the number of pages to be accessed by a query performing a scan, which adversely affects performance. However, fragmentation is frequently not an issue for point queries. If you are seeing a lot of scans, you should ensure that the database objects referred to in the query are not too fragmented.

You can determine the fragmentation of the five tables accessed by the worst-performing query by running a query against `sys.dm_db_index_physical_stats`. Begin by running the query against the `HumanResources.Employee` table.

```
SELECT s.avg_fragmentation_in_percent,
       s.fragment_count,
       s.page_count,
       s.avg_page_space_used_in_percent,
       s.record_count,
       s.avg_record_size_in_bytes,
       s.index_id
FROM sys.dm_db_index_physical_stats(DB_ID('AdventureWorks2017'),
                                    OBJECT_ID(N'HumanResources.Employee'),
                                    NULL,
                                    NULL,
                                    'Sampled') AS s
WHERE s.record_count > 0
ORDER BY s.index_id;
```

Figure 27-5 shows the output of this query.

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes	index_id
1	0	1	7	91.1645416357796	290	176.158	1
2	0	1	1	60.4892512972572	290	14.889	2
3	0	1	1	67.6550531257722	290	16.889	3
4	33.3333333333333	2	3	70.1672102792192	290	56.772	4
5	50	2	2	56.6963182604398	290	29.662	5
6	0	1	1	93.1307141092167	290	24	6

Figure 27-5. *The index fragmentation of the HumanResources.Employee table*

If you run the same query for the other four tables (in order `Purchasing.PurchaseOrderHeader`, `Purchasing.PurchaseOrderDetail`, `Production.Product`, and `Person.Person`), the output will look like Figure 27-6.

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes	index_id
1	0	4	42	99.110452186805	4012	82	1
2	14.2857142857143	2	7	99.110452186805	4012	12	2
3	14.2857142857143	2	7	99.110452186805	4012	12	3

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes	index_id
1	0	3	64	99.0089201877934	8845	56	1
2	5	3	20	98.32592043489	8845	16	2

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes	index_id
1	7.69230769230769	2	13	92.7999876451693	504	191.793	1
2	50	2	2	94.6009389671361	504	28.392	2
3	25	2	4	79.0832715591796	504	48.817	3
4	50	2	2	80.9241413392637	504	24	4

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes	index_id
1	0.183678824455523	161	3811	85.6245737583395	19972	1320.83	1
2	6.66666666666667	8	105	97.2379416851989	19972	39.388	2
3	0	7	65	98.6755621447986	19972	24	3
4	0	1	3	68.2151470224858	195	82.974	256000
5	0	2	2152	99.6210155670867	301696	55.53	256001
6	0.503959683225342	52	1389	99.4253521126761	301696	35.059	256002
7	1.078360891445	66	1391	99.282357301705	301696	35.059	256003
8	0.503959683225342	57	1389	99.4253521126761	301696	35.059	256004

Figure 27-6. *The index fragmentation for the four tables in the problem query*

The fragmentation of all the indexes is very low, and the space used for all of them is very high. This means it's unlikely that any of them are negatively impacting performance. If you also examine the fact that most of the indexes here have less than 100 pages in them, this makes them very small indexes, and even if they were fragmented, the degree of this affecting the query must be extremely minimal. In fact, fragmentation is far too frequently a crutch to try to improve performance without doing the hard work of identifying the actual issues in the system, usually internal behavior of the query.

Worth noting is the index that has 301,696 rows instead of the 19,972 in the other indexes. If you look that up, it's an XML index, so the difference is in the XML tree. It's not used anywhere in these queries, so we'll ignore it here.

Once you've analyzed the external factors that can affect the performance of a query and resolved the nonoptimal ones, you should analyze internal factors, such as improper indexing and query design.