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.

# Analyzing the Internal Behavior of the Costliest Query

Now you need to analyze the processing strategy for the query chosen by the optimizer to determine the internal factors affecting the query's performance. Analyzing the internal factors that can affect query performance involves these steps:

- Analyzing the query execution plan
- Identifying the costly steps in the execution plan
- Analyzing the effectiveness of the processing strategy

## Analyzing the Query Execution Plan

To see the execution plan, click the Show Actual Execution Plan button to enable it and then run the stored procedure. Be sure you're doing these types of tests on a nonproduction system, while, at the same time, have it be as much like production as possible so that the behavior there mirrors what you see in production. We covered execution plans in Chapter 16. For more details on reading execution plans, check out my book *SQL Server Execution Plans* (Red Gate Publishing, 2018). Figure 27-7 shows the graphical execution plan of the worst-performing query.

***Figure 27-7.*** *The actual execution plan of the worst-performing query*

The graphic of this plan is somewhat difficult to read. I'll break down a few of the interesting details in case you're not following along with code. You could observe the following from this execution plan:

- `SELECT` properties
  - `Optimization Level: Full`
  - `Reason for Early Termination: Good enough plan found`

    - Query Time Statistics: 30ms CpuTime and 244 ms ElapsedTime
    - WaitStats: WaitCount 4, WaitTimeMs 214, WaitType ASYNC_NETWORK_IO
- Data access
    - Index seek on nonclustered index, Person.IX_Person_LastName_FirstName_MiddleName
    - Clustered index scan on, PurchaseOrderHeader.PK_PruchaseOrderHeader_PurchaseOrderID
    - Clustered index seek on PurchaseOrderDetail.PK_PurchaseOrderDetail_PurchaseOrderDetailID
    - Clustered Index seek on Product.PK_Product_ProductID
    - Clustered Index seek on Employee.PK_Employee_BusinessEntityID
    - Join strategy
    - Nested loop join between the constant scan and Person.Person table with the Person.Person table as the outer table
    - Nested loop join between the output of the previous join and Purchasing.PurchaseOrderHeader with the Purchasing.PurchaseOrderHeader table as the outer table
    - Nested loop join between the output of the previous join and the Purchasing.PurchaseOrderDetail table that was also the outer table
    - Nested loop join between the output of the previous join and the Production.Product table with Production.Product as the outer table
    - Nested loop join between the previous join and the HumanResources.Employee table with the HumanResource.Employee table as the outer table

- Additional processing
  - Constant scan to provide a placeholder for the `@LastName` variable's `LIKE` operation
  - Compute scalar that defined the constructs of the `@LastName` variable's `LIKE` operation, showing the top and bottom of the range and the value to be checked
  - Compute scalar that combines the `FirstName` and `LastName` columns into a new column
  - Compute scalar that calculates the `LineTotal` column from the `Purchasing.PurchaseOrderDetail` table
  - Compute scalar that takes the calculated `LineTotal` and stores it as a permanent value in the result set for further processing

All this information is available by browsing the details of the operators exposed in the properties sheet from the graphical execution plan.

## Identifying the Costly Steps in the Execution Plan

Once you understand the execution plan of the query, the next step is to identify the steps estimated as the most costly in the execution plan. Although these costs are estimated and don't reflect reality in any way, they are the only numbers you will receive that measure the function of the plan, so identifying, understanding, and possibly addressing the most costly operations can result in massive performance benefit. You can see that the following are the two costliest steps:

- *Costly step 1*: The clustered index scan on the `Purchasing.PurchaseOrderHeader` table is 36 percent.
- *Costly step 2*: The hash match join operation is 32 percent.

The next optimization step is to analyze the costliest steps so you can determine whether these steps can be optimized through techniques such as redesigning the query or indexes.

## Analyzing the Processing Strategy

While the optimizer completed optimizing the plan, which you know because the reason for early termination of the optimization process was "Good Enough Plan Found" (or, because it showed FULL optimization without a reason for early termination), that doesn't mean there are not tuning opportunities in the query and structure. You can begin evaluating it by following the traditional steps.

Costly step 1 is a clustered index scan. Scans are not necessarily a problem. They're just an indication that a full scan of the object in question, in this case the entire table, was less costly than the alternatives to retrieve the information needed by the query.

Costly step 2 is the hash match join operation of the query. This again is not necessarily a problem. But, sometimes, a hash match is an indication of bad or missing indexes, or queries that can't make use of the existing indexes, so they are frequently an area that needs work. At least, that's frequently the case for OLTP systems. For large data warehouse systems, a hash match may be ideal for dealing with the types of queries you'll see there.

---

**Tip** At times you may find that no improvements can be made to the costliest step in a processing strategy. In that case, concentrate on the next costliest step to identify the problem. If none of the steps suggests indications for optimization, then you may need to consider changing the database design or the construction of the query.

---

# Optimizing the Costliest Query

Once you've diagnosed the queries with costly steps, the next stage is to implement the necessary corrections to reduce the cost of these steps.

The corrective actions for a problematic step can have one or more alternative solutions. For example, should you create a new index or structure the query differently? In such cases, you should prioritize the solutions based on their expected effectiveness and the amount of work required. For example, if a narrow index can more or less do the job, then it is usually better to prioritize that over changes to code that might lead to business testing. Making changes to code may also be the less intrusive approach. You need to evaluate each situation within the business and application that you're dealing with.

Apply the solutions individually in the order of their expected benefit and measure their individual effect on the query performance. Finally, you can apply the solution (or solutions) that provides the greatest performance improvement to correct the problematic step. Sometimes, it may be evident that the best solution will hurt other queries in the workload. For example, a new index on a large number of columns can hurt the performance of action queries. However, since that's not always an issue, it's better to determine the effect of such optimization techniques on the complete workload through testing. If a particular solution hurts the overall performance of the workload, choose the next best solution while keeping an eye on the overall performance of the workload.

## Modifying the Code

The costliest operation in the query is a clustered index scan of the `PurchaseOrderHeader` table. The first thing you need to do is understand if the clustered index scan is necessary for the query and data returned or may be there because of the code or even because another index or a different index structure could work better. To begin to understand why you're getting a clustered index scan, you should look at the properties of the scan operation. Since you're getting a scan, you also need to look to the code to ensure it's sargable. Specifically, you're interested in the `Predicate` property, as shown in Figure 27-8.

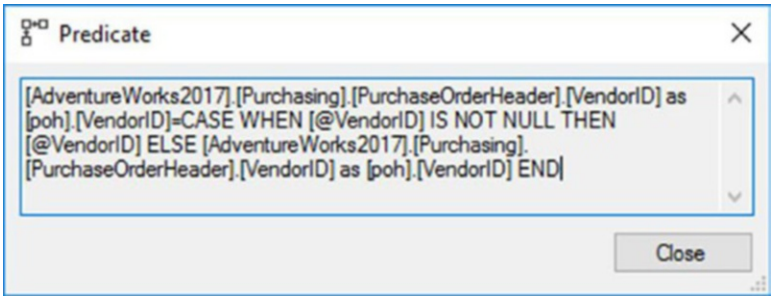


***Figure 27-8.*** *The predicate of the clustered index scan*

This is a calculation. There is an existing index on the `VendorID` column of the `PurchaseOrderTable` that might be of use to this query, but because you're using a `COALESCE` statement to filter values, a scan of the entire table is necessary to retrieve the information. The `COALESCE` operator is basically a way to take into account that a given

value might be NULL and, if it is NULL, to provide an alternate value, possibly several alternate values. However, it's a function, and a function against a column within a WHERE clause, the JOIN criteria, or a HAVING clause is likely to lead to scans, so you need to get rid of the function. Because of this function, you can't simply add or modify the index because you'd still end up with a scan. You could try rewriting the query with an OR clause like this:

```
...WHERE   per.LastName LIKE @LastName AND
        poh.VendorID = @VendorID
        OR poh.VendorID = poh.VendorID…
```

But logically, that's not the same as the COALESCE operation. Instead, it's substituting one part of the WHERE clause for another, not just using the OR construct. So, you could rewrite the entire stored procedure definition like this:

```
CREATE OR ALTER PROCEDURE dbo.PurchaseOrderBySalesPersonName
    @LastName NVARCHAR(50),
    @VendorID INT = NULL
AS
IF @VendorID IS NULL
BEGIN
    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
    ORDER BY per.LastName,
             per.FirstName;
END
ELSE
BEGIN
    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 = @VendorID
    ORDER BY per.LastName,
             per.FirstName;
END
GO
```

Using the IF construct breaks the query in two. Running it with the same set of parameters resulted in a change in execution time from 434ms to 128ms (as measured in Extended Events), which is a fairly strong improvement. The reads went up from 8,671 to 9,243. While the execution time went down quite a lot, we had a small increase in reads. The execution plan is certainly different, as shown in Figure 27-9.

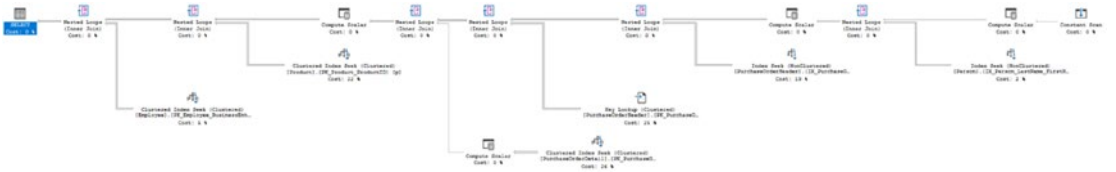

***Figure 27-9.*** *New execution plan after breaking apart the query*

The two costliest operators are now different. There are no more scan operations, and all the join operations are now loop joins. But, a new data access operation has been added. You're now seeing a `Key Lookup` operation, as described in Chapter 12, so you have more tuning opportunities.

## Fixing the Key Lookup Operation

Now that you know you have a key lookup, you need to determine whether any of the methods for addressing it suggested in Chapter 12 can be applied. First, you need to know what columns are being retrieved in the operation. This means accessing the properties of the `Key Lookup` operator. The properties show the `VendorID` and `OrderDate` columns. This means you only need to add those columns to the leaf pages of the index through the `INCLUDE` part of the nonclustered index. You can modify that index as follows:

```
CREATE NONCLUSTERED INDEX IX_PurchaseOrderHeader_EmployeeID
ON Purchasing.PurchaseOrderHeader
(
    EmployeeID ASC
)
INCLUDE
(
    VendorID,
    OrderDate
)
WITH DROP_EXISTING;
```

Applying this index results in a change in the execution plan and a modification in the performance. The previous structure and code resulted in 128ms. With this new index in place, the query execution time dropped to 110ms, and the reads have dropped to 7748. The execution plan is now completely different, as shown in Figure 27-10.

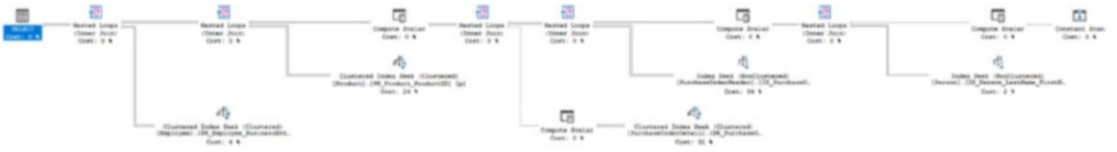

***Figure 27-10.*** *New execution plan after modifying the index*

At this point there are nothing but nested loop joins and index seeks. There's not even a sort operation anymore despite the `ORDER BY` statement in the query. This is because the output of the index seek against the `Person` table is `Ordered` and the rest of the operations maintain that order. In short, you're largely in good shape as far as this query goes, but there were two queries in the procedure now.

## Tuning the Second Query

Eliminating `COALESCE` allowed you to use existing indexes, but in doing this you effectively created two paths through your query. Because you've explored the first path only because you have used only the single parameter, you've been ignoring the second query. Let's modify the test script to see how the second path through the query will work.

```
EXEC dbo.PurchaseOrderBySalesPersonName @LastName = 'Hill%',
    @VendorID = 1496;
```

Running this query results in a different execution plan entirely. You can see the most interesting part of this in Figure 27-11.

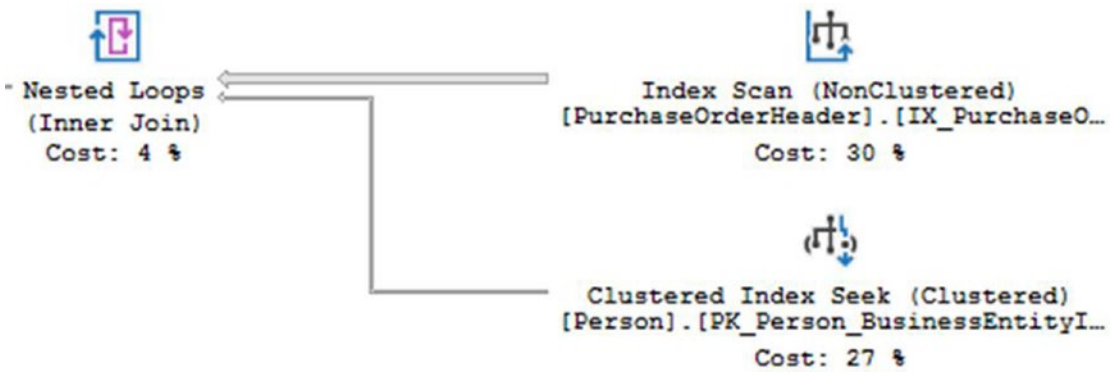


***Figure 27-11.*** *Execution plan for the other query in the procedure*

This new query has different behaviors because of the differences in the query. The main issue here is a clustered index scan against the `PurchaseOrderHeader` table. You're seeing a scan despite that there is an index on `VendorID`. Again, you can look to see what the output of the operator includes. This time, it's more than just two columns: `OrderDate`, `EmployeeID`, `PurchaseOrderID`. These are not very large columns, but they will add to the size of the index. You'll need to evaluate whether this increase in index size is worth the performance benefits of the elimination of the scan of the index. I'm going to go ahead and try it by modifying the index as follows:

```
CREATE NONCLUSTERED INDEX IX_PurchaseOrderHeader_VendorID
ON Purchasing.PurchaseOrderHeader
(
    VendorID ASC
)
INCLUDE
(
    OrderDate,
    EmployeeID,
    PurchaseOrderID
)
WITH DROP_EXISTING;
GO
```

Prior to applying the index, the execution time was around 4.3ms with 273 reads. After applying the index, the execution time dropped to 2.3ms and 263 reads. The execution plan now looks like Figure 27-12.

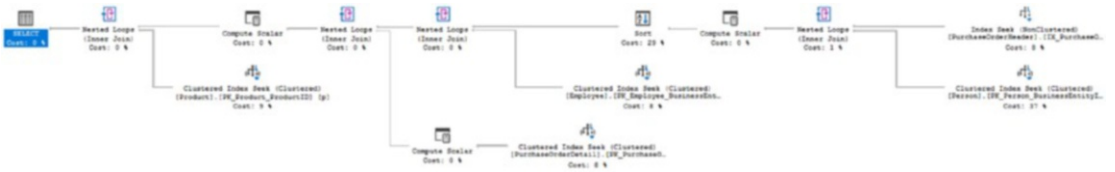

***Figure 27-12.*** *The second execution plan after modifying the index*

The new execution plan consists of index seeks and nested loops joins. There is a sort operator, the second costliest in the plan, ordering the data by `LastName` and `FirstName`. Getting this to be taken care of by the retrieval process might help to improve performance, but I've had a fairly successful tuning to this point, so I'll leave it as is for now.

One additional consideration should be made for the split query. When the optimizer processes a query like this, both statements will be optimized for the parameter values passed in. Because of this, you may see bad execution plans, especially for the second query that uses the `VendorID` for filtering, because of parameter sniffing gone bad. To avoid that situation, one additional tuning effort should be made.

## Creating a Wrapper Procedure

Because you've created two paths within the procedure to accommodate the different mechanisms of querying the data, you have the potential for getting bad parameter sniffing because both paths will be compiled, regardless of the parameters passed. One mechanism around this is to run the procedure you have into a wrapper procedure. But first, you have to create two new procedures, one for each query like this:

```
CREATE OR ALTER PROCEDURE dbo.PurchaseOrderByLastName @LastName
NVARCHAR(50)
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
ORDER BY per.LastName,
         per.FirstName;
GO

CREATE OR ALTER PROCEDURE dbo.PurchaseOrderByLastNameVendor
    @LastName NVARCHAR(50),
    @VendorID INT
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 = @VendorID
ORDER BY per.LastName,
         per.FirstName;
GO
```

Then you have to modify the existing procedure so that it looks like this:

```
CREATE OR ALTER PROCEDURE dbo.PurchaseOrderBySalesPersonName
    @LastName NVARCHAR(50),
    @VendorID INT = NULL
AS
IF @VendorID IS NULL
BEGIN
    EXEC dbo.PurchaseOrderByLastName @LastName;
END
ELSE
BEGIN
    EXEC dbo.PurchaseOrderByLastNameVendor @LastName, @VendorID;
END
GO
```

With that in place, regardless of the code path chosen, the first time these queries are called, each procedure will get its own unique execution plan, avoiding bad parameter sniffing. And, this won't negatively impact performance time. If I run both the queries now, the results are approximately the same. This pattern works very well for a small number of paths. If you have some large number of paths, certainly more than 10 or so, this pattern breaks down, and you may need to look to dynamic execution methods.

Taking the performance from 434ms to 110ms or 2.3ms, depending on our new queries, is a pretty good reduction in execution time, and we also had equally big wins on reads. If this query were called hundreds of times in a minute, that level of reduction would be quite serious indeed. But, you should always go back and assess the impact on the overall database workload.

# Analyzing the Effect on Database Workload

Once you've optimized the worst-performing query, you must ensure that it doesn't hurt the performance of the other queries; otherwise, your work will have been in vain.

To analyze the resultant performance of the overall workload, you need to use the techniques outlined in Chapter 15. For the purposes of this small test, reexecute the complete workload and capture extended events to record the overall performance.

---

**Tip** For proper comparison with the original extended events, please ensure that the graphical execution plan is off.

---

Figure 27-13 shows the corresponding Extended Events output captured.

| name | batch_text | duration | logical_reads | row_count |
|---|---|---|---|---|
| sql_batch_completed | EXEC dbo.PurchaseOrderBySalesPer... | 138084 | 7719 | 1496 |
| sql_batch_completed | EXEC dbo.ShoppingCart @Shopping... | 4006 | 6 | 2 |
| sql_batch_completed | EXEC dbo.ProductBySalesOrder @Sa... | 473 | 43 | 18 |
| sql_batch_completed | EXEC dbo.PersonByFirstName @First... | 1999 | 110 | 1 |
| sql_batch_completed | EXEC dbo.ProductTransactionsSince... | 433 | 117 | 23 |
| sql_batch_completed | EXEC dbo.PurchaseOrderBySalesPer... | 744 | 260 | 28 |
| sql_batch_completed | EXEC dbo.TotalSalesByProduct @Pr... | 13445 | 1248 | 1 |

***Figure 27-13.*** *The Extended Events output showing the effect of optimizing the costliest query on the complete workload*

It's possible that the optimization of the worst-performing query may hurt the performance of some other query in the workload. However, as long as the overall performance of the workload is improved, you can retain the optimizations performed on the query. In this case, the other queries were not impacted. But now, there is a query that takes longer than the others. It too might need optimization, and the whole process starts again. This is also a place where having the Query Store in place so that you can look for regression or changes in behavior easily becomes a great resource.

# Iterating Through Optimization Phases

An important point to remember is that you need to iterate through the optimization steps multiple times. In each iteration, you can identify one or more poorly performing queries and optimize the query or queries to improve the performance of the overall workload. You must continue iterating through the optimization steps until you achieve adequate performance or meet your service level agreement (SLA).

Besides analyzing the workload for resource-intensive queries, you must also analyze the workload for error conditions. For example, if you try to insert duplicate rows into a table with a column protected by the unique constraint, SQL Server will reject the new rows and report an error condition to the application. Although the data was not entered into the table and no useful work was performed, valuable resources were used to determine that the data was invalid and must be rejected.

To identify the error conditions caused by database requests, you will need to include the following in your Extended Events session (alternatively, you can create a new session that looks for these events in the `errors` or `warnings` category):

- `error_reported`
- `execution_warning`
- `hash_warning`
- `missing_column_statistics`
- `missing_join_predicate`
- `sort_warning`
- `hash_spill_details`

For example, consider the following SQL queries:

```
INSERT  INTO Purchasing.PurchaseOrderDetail
        (PurchaseOrderID,
         DueDate,
         OrderQty,
         ProductID,
         UnitPrice,
         ReceivedQty,
         RejectedQty,
```

```
         ModifiedDate
        )
VALUES  (1066,
         '1/1/2009',
         1,
         42,
         98.6,
         5,
         4,
         '1/1/2009'
        ) ;
GO

SELECT  p.[Name],
        psc.[Name]
FROM    Production.Product AS p,
        Production.ProductSubCategory AS psc ;
GO
```

Figure 27-14 shows the corresponding session output.

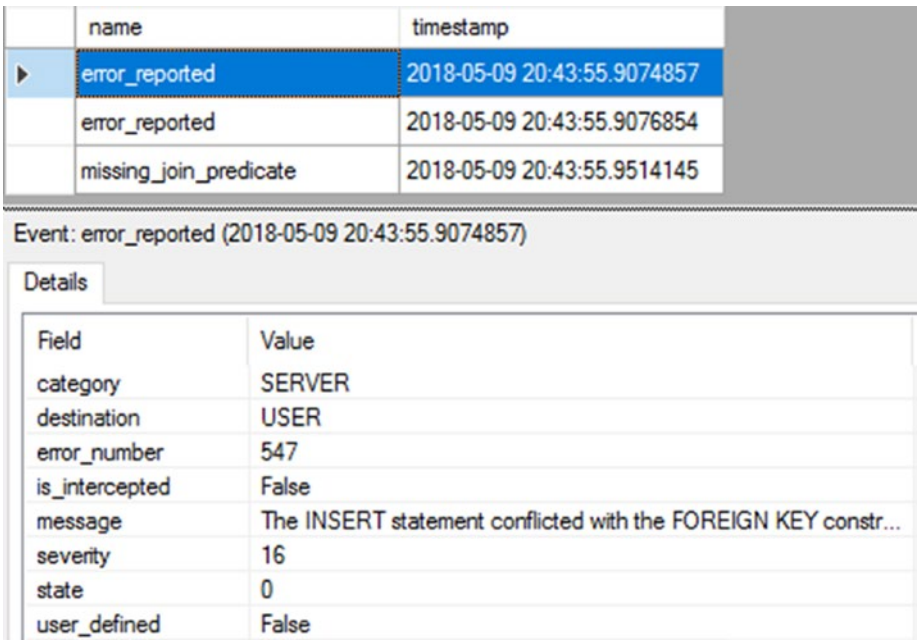


***Figure 27-14.*** *Extended Events output showing errors raised by a SQL workload*

From the Extended Events output in Figure 27-14, you can see that the two errors I intentionally generated occurred.

- `error_reported`
- `missing_join_predicate`

The `error_reported` error was caused by the `INSERT` statement, which tried to insert data that did not pass the referential integrity check; namely, it attempted to insert `Productld = 42` when there is no such value in the `Production.Product` table. From the `error_number` column, you can see that the error number is 547. The `message` column shows the full description for the error. It's worth noting, though, that `error_reported` can be quite chatty with lots of data returned and not all of it useful.

The second type of error, `missing_join_predicate`, is caused by the `SELECT` statement.

```
SELECT p.Name,
       c.Name
FROM Production.Product AS p,
     Production.ProductSubcategory AS c;
```

If you take a closer look at the `SELECT` statement, you will see that the query does not specify a `JOIN` clause between the two tables. A missing join predicate between the tables usually leads to an inaccurate result set and a costly query plan. This is what is known as a *Cartesian join*, which leads to a *Cartesian product*, where every row from one table is combined with every row from the other table. You must identify the queries causing such events in the `Errors and Warnings` section and implement the necessary fixes. For instance, in the preceding `SELECT` statement, you should not join every row from the `Production.ProductCategory` table to every row in the `Production.Product` table—you must join only the rows with matching `ProductCategorylD`, as follows:

```
SELECT p.Name,
       c.Name
FROM Production.Product AS p
    JOIN Production.ProductSubcategory AS c
        ON p.ProductSubcategoryID = c.ProductSubcategoryID;
```

Even after you thoroughly analyze and optimize a workload, you must remember that workload optimization is not a one-off process. The workload or data distribution on a database can change over time, so you should periodically check whether your queries are optimized for the current situation. It's also possible that you may identify shortcomings in the design of the database itself. Too many joins from overnormalization or too many columns from improper denormalization can both lead to queries that perform badly, with no real optimization opportunities. In this case, you will need to consider redesigning the database to get a more optimized structure.

## Summary

As you learned in this chapter, optimizing a database workload requires a range of tools, utilities, and commands to analyze different aspects of the queries involved in the workload. You can use Extended Events to analyze the big picture of the workload and identify the costly queries. Once you've identified the costly queries, you can use the execution plan and various SQL commands to troubleshoot the problems associated with the costly queries. Based on the problems detected with the costly queries, you can apply one or more sets of optimization techniques to improve the query performance. The optimization of the costly queries should improve the overall performance of the workload; if this does not happen, you should roll back the change or changes.

In the next chapter, I summarize the performance-related best practices in a nutshell. You'll be able to use this information as a quick and easy-to-read reference.

CHAPTER 28

# SQL Server Optimization Checklist

If you have read through the previous 27 chapters of this book, then you understand the major aspects of performance optimization. You also understand that it is a challenging and ongoing activity.

What I hope to do in this chapter is to provide a performance-monitoring checklist that can serve as a quick reference for database developers and DBAs when in the field. The idea is similar to the notion of tear-off cards of *best practices*. This chapter does not cover everything, but it does summarize, in one place, some of the major tuning activities that can have a quick and demonstrable impact on the performance of your SQL Server systems. I have categorized these checklist items into the following sections:

- Database design
- Configuration settings
- Database administration
- Database backup
- Query design

Each section contains a number of optimization recommendations and techniques. Where appropriate, each section also cross-references specific chapters in this book that provide more detailed information.




G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_28

# Database Design

Database design is a broad topic, and it can't be given due justice in a small section in this query tuning book; nevertheless, I advise you to keep an eye on the following design aspects to ensure that you pay attention to database performance from an early stage:

- Use entity-integrity constraints.
- Maintain domain and referential integrity constraints.
- Adopt index-design best practices.
- Avoid the use of the `sp_` prefix for stored procedure names.
- Minimize the use of triggers.
- Put tables into in-memory storage.
- Use columnstore indexes.

## Use Entity-Integrity Constraints

*Data integrity* is essential to ensuring the quality of data in the database. An essential component of data integrity is *entity integrity,* which defines a row as a unique entity for a particular table; that is, every row in a table must be uniquely identifiable. The column or columns serving as the unique row identifier for a table must be represented as the primary key of the table.

Sometimes, a table may contain an additional column (or columns) that also can be used to uniquely identify a row in the table. For example, an `Employee` table may have the `EmployeeID` and `SocialSecurityNumber` columns. The `EmployeeID` column serves as the unique row identifier, and it can be defined as the *primary key*. Similarly, the `SocialSecurityNumber` column can be defined as the *alternate key*. In SQL Server, alternate keys can be defined using unique constraints, which are essentially the younger siblings to primary keys. In fact, both the unique constraint and the primary key constraint use unique indexes behind the scenes.

It's worth noting that there is honest disagreement regarding the use of a natural key (for example, the `SocialSecurityNumber` column in the previous example) or an artificial key (for example, the `EmployeeID` column). I've seen both designs succeed, but each approach has strengths and weaknesses. Rather than suggest one over the other,

I'll provide you with a couple of reasons to use both and some of the costs associated with each and thereby avoid the religious argument. An identity column is usually an `INT` or a `BIGINT`, which makes it narrow and easy to index, improving performance. Also, separating the value of the primary key from any business knowledge is considered good design in some circles. Sometimes globally unique identifiers (GUIDs) may be used as a primary key. They work fine but are difficult to read, so it impacts troubleshooting, and they can lead to greater index fragmentation. The breadth of the key can also cause a negative impact to performance. One of the drawbacks of artificial keys is that the numbers sometimes acquire business meaning, which should never happen. Another thing to keep in mind is that you have to create a unique constraint for the alternate keys to prevent the creation of multiple rows where none should exist. This increases the amount of information you have to store and maintain. Natural keys provide a clear, human-readable, primary key that has true business meaning. They tend to be wider fields—sometimes very wide—making them less efficient inside indexes. Also, sometimes the data may change, which has a profound trickle-down effect within your database because you will have to update every single place that key value is in use instead of simply one place with an artificial key. With the introduction of compliance like the General Data Protection Regulation (GDPR) in the European Union, natural keys become more problematic when worrying about your ability to modify data without removing it.

Let me just reiterate that either approach can work well and that each provides plenty of opportunities for tuning. Either approach, properly applied and maintained, will protect the integrity of your data.

Besides maintaining data integrity, unique indexes—the primary vehicle for entity-integrity constraints—help the optimizer generate efficient execution plans. SQL Server can often search through a unique index faster than it can search through a nonunique index. This is because each row in a unique index is unique; and, once a row is found, SQL Server does not have to look any further for other matching rows (the optimizer is aware of this fact). If a column is used in sort (or `GROUP BY` or `DISTINCT`) operations, consider defining a unique constraint on the column (using a unique index) because columns with a unique constraint generally sort faster than ones with no unique constraint. Also, a unique constraint adds additional information for the optimizer's cardinality estimation. Even an "unused" or "disused" index may still be helpful for optimization because of the effects on cardinality estimation.

To understand the performance benefit of entity-integrity or unique constraints, consider an example. Assume you want to modify the existing unique index on the `Production.Product` table.

```
CREATE NONCLUSTERED INDEX AK_Product_Name
ON Production.Product
(
    Name ASC
)
WITH (DROP_EXISTING = ON)
ON [PRIMARY];
GO
```

The nonclustered index does not include the `UNIQUE` constraint. Therefore, although the `[Name]` column contains unique values, the absence of the `UNIQUE` constraint from the nonclustered index does not provide this information to the optimizer in advance. Now, let's consider the performance impact of the `UNIQUE` constraint (or a missing `UNIQUE` constraint) on the following `SELECT` statement:

```
SELECT DISTINCT
       (p.Name)
FROM Production.Product AS p;
```

Figure 28-1 shows the execution plan of this `SELECT` statement.


***Figure 28-1.*** *An execution plan with no UNIQUE constraint on the [Name] column*

From the execution plan, you can see that the nonclustered `AK_ProductName` index is used to retrieve the data, and then a `Stream Aggregate` operation is performed on the data to group the data on the `[Name]` column so that the duplicate `[Name]` values can be removed from the final result set. Note that the `Stream Aggregate` operation would not have been required if the optimizer had been told in advance about the uniqueness of

the [Name] column. You can accomplish this by defining the nonclustered index with a UNIQUE constraint, as follows:

```
CREATE UNIQUE NONCLUSTERED INDEX [AK_Product_Name]
ON [Production].[Product]([Name] ASC)
WITH (
DROP_EXISTING = ON)
ON    [PRIMARY];
GO
```

Figure 28-2 shows the new execution plan of the SELECT statement.

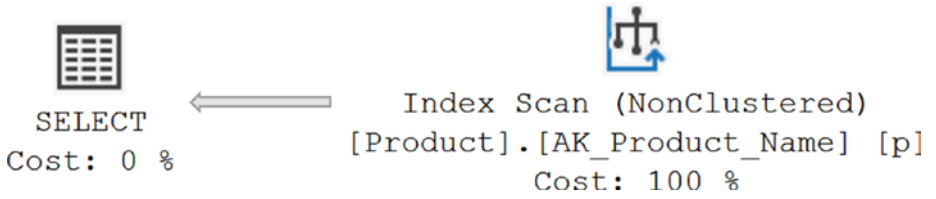


***Figure 28-2.*** *An execution plan with a UNIQUE constraint on the [Name] column*

In general, the entity-integrity constraints (in other words, primary keys and unique constraints) provide useful information to the optimizer about the expected results, assisting the optimizer in generating efficient execution plans. Of note is the fact that sys.dm_db_index_usage_stats doesn't show when a constraint check has been run against the index that defines the unique constraint.

## Maintain Domain and Referential Integrity Constraints

The other two important components of data integrity are *domain integrity* and *referential integrity*. Domain integrity for a column can be enforced by restricting the data type of the column, defining the format of the input data, and limiting the range of acceptable values for the column. Referential integrity is enforced by the use of foreign key constraints defined between tables. SQL Server provides the following features to implement the domain and referential integrity: data types, FOREIGN KEY constraints, CHECK constraints, DEFAULT definitions, and NOT NULL definitions. If an application requires that the values for a data column be restricted to a range of values, then this business rule can be implemented either in the application code or in the database schema. Implementing such a business rule in the database using domain constraints (such as the CHECK constraint) can help the optimizer generate efficient execution plans.

To understand the performance benefit of domain integrity, consider this example:

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (
     C1 INT,
     C2 INT CHECK (C2 BETWEEN 10 AND 20)
    ) ;
INSERT  INTO dbo.Test1
VALUES  (11, 12);
GO
DROP TABLE IF EXISTS dbo.Test2;
GO
CREATE TABLE dbo.Test2 (C1 INT, C2 INT);
INSERT  INTO dbo.Test2
VALUES  (101, 102);
```

Now execute the following two SELECT statements:

```
SELECT T1.C1,
       T1.C2,
       T2.C2
FROM dbo.Test1 AS T1
    JOIN dbo.Test2 AS T2
        ON T1.C1 = T2.C2
           AND T1.C2 = 20;
GO
SELECT T1.C1,
       T1.C2,
       T2.C2
FROM dbo.Test1 AS T1
    JOIN dbo.Test2 AS T2
        ON T1.C1 = T2.C2
           AND T1.C2 = 30;
```

The two SELECT statements appear to be the same, except for the predicate values (20 in the first statement and 30 in the second). Although the two SELECT statements have the same form, the optimizer treats them differently because of the CHECK constraint on the T1.C2 column, as shown in the execution plan in Figure 28-3.

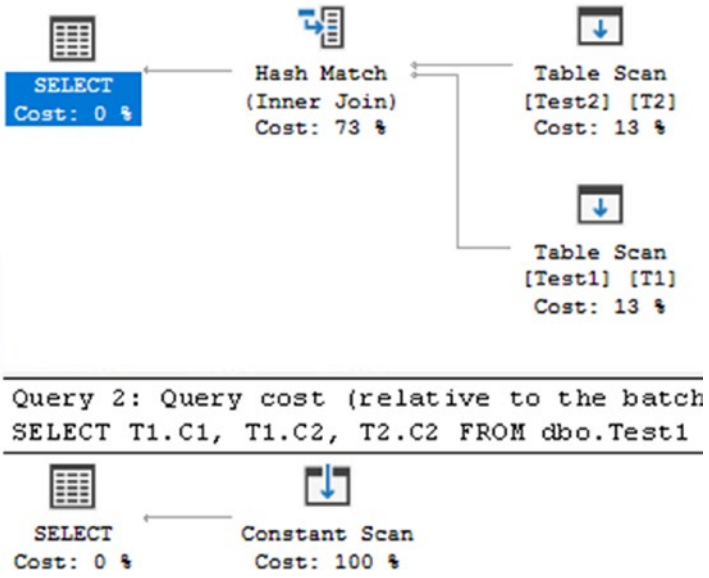


***Figure 28-3.*** *Execution plans with predicate values within and outside the CHECK constraint boundaries*

From the execution plan, you can see that, for the first query (with T1.C2 = 20), the optimizer accesses the data from both tables. For the second query (with T1.C2 = 30), the optimizer understands from the corresponding CHECK constraint on the column T1.C2 that the column can't contain any value outside the range of 10 to 20. Thus, the optimizer doesn't even access the data from either table. Consequently, the relative estimated cost, and the actual performance measurement of doing almost nothing, of the second query is 0 percent.

I explained the performance advantage of referential integrity in detail in the "Declarative Referential Integrity" section of Chapter 19.

Therefore, you should use domain and referential constraints not only to implement data integrity but also to facilitate the optimizer in generating efficient query plans. Make sure that your foreign key constraints are created using the WITH CHECK option, or the optimizer will ignore them. To understand other performance benefits of domain and referential integrity, please refer to the "Using Domain and Referential Integrity" section of Chapter 19.

## Adopt Index-Design Best Practices

The most common optimization recommendation—and frequently one of the biggest contributors to good performance—is to implement the correct indexes for the database workload. Indexes are unlike tables, which are used to store data and can be designed even without knowing the queries thoroughly (as long as the tables properly represent the business entities). Instead, indexes must be designed by reviewing the database queries thoroughly. Except in common and obvious cases, such as primary keys and unique indexes, please don't fall into the trap of designing indexes without knowing the queries. Even for primary keys and unique indexes, I advise you to validate the applicability of those indexes as you start designing the database queries. Considering the importance of indexes for database performance, you must be careful when designing indexes.

Although the performance aspect of indexes is explained in detail in Chapters 8, 9, 12, and 13, I'll reiterate a short list of recommendations for easy reference here:

- Choose narrow columns for index keys.
- Ensure that the selectivity of the data in the candidate column is very high (that is, the column must have a low number of candidate values returned).
- Prefer columns with the integer data type (or variants of the integer data type). Also, avoid indexes on columns with string data types such as `VARCHAR`.
- Consider listing columns having higher selectivity first in a multicolumn index.
- Use the `INCLUDE` list in an index as a way to make an index cover the index key structure without changing that structure. Do this by adding columns to the key, which enables you to avoid expensive lookup operations.
- When deciding which columns to index, pay extra attention to the queries' `WHERE` clauses and `JOIN` criteria columns and `HAVING` clause. These can serve as the entry points into the tables, especially if a `WHERE` clause criterion on a column filters the data on a highly selective value or constant. Such a clause can make the column a prime candidate for an index.

- When choosing the type of an index (clustered or nonclustered, columnstore or rowstore), keep in mind the advantages and disadvantages of the various index types.

Be extra careful when designing a clustered index because every nonclustered index on the table depends on the clustered index. Therefore, follow these recommendations when designing and implementing clustered indexes:

- Keep the clustered indexes as narrow as possible. You don't want to widen all your nonclustered indexes by having a wide clustered index.
- Create the clustered index first and then create the nonclustered indexes on the table.
- If required, rebuild a clustered index in a single step using the `DROP_EXISTING = {ON|OFF}` command in the `CREATE INDEX` command. You don't want to rebuild all the nonclustered indexes on the table twice: once when the clustered index is dropped and again when the clustered index is re-created.
- Do not create a clustered index on a frequently updated column. If you do so, the nonclustered indexes on the table will create additional load by remaining in sync with the clustered index key values.

To keep track of the indexes you've created and determine others that you need to create, you should take advantage of the dynamic management views that SQL Server 2017 and Azure SQL Database make available to you. By checking the data in `sys.dm_db_index_usage_stats` on a regular basis—say once a week or so—you can determine which of your indexes are actually being used and which are redundant. Indexes that are not contributing to your queries to help you improve performance are just a drain on the system. They require both more disk space and additional I/O to maintain the data inside the index as the data in the table changes. On the other hand, querying `sys.dm_db_missing_indexes_details` will show potential indexes deemed missing by the system and even suggest `INCLUDE` columns. You can access the DMV `sys.dm_db_missing_indexes_groups_stats` to see aggregate information about the number of times queries are called that could have benefited from a particular group of indexes. Just remember to test these suggestions thoroughly and don't assume that they will be correct. All these suggestions are just that: suggestions. All these tips can be combined to give you an optimal method for maintaining the indexes in your system over the long term.

## Avoid the Use of the sp_Prefix for Stored Procedure Names

As a rule, don't use the `sp_` prefix for user stored procedures since SQL Server assumes that stored procedures with the `sp_` prefix are system stored procedures, and these are supposed to be in the master database. Using `sp` or `usp` as the prefix for user stored procedures is quite common. This is neither a major performance hit nor a major problem, but why court trouble? The performance hit of the `sp_` prefix is explained in detail in the "Be Careful Naming Stored Procedures" section of Chapter 20. Getting rid of prefixes entirely is a fine way to go. You have plenty of space for descriptive object names. There is no need for odd abbreviations that don't add to the functional definition of the queries.

## Minimize the Use of Triggers

Triggers provide an attractive method for automating behavior within the database. Since they fire as data is manipulated by other processes (regardless of the processes), triggers can be used to ensure certain functions are run as the data changes. That same functionality makes them dangerous since they are not immediately visible to the developer or DBA working on a system. They must be taken into account when designing queries and when troubleshooting performance problems. Because they carry a somewhat hidden cost, triggers should be considered carefully. Before using a trigger, make sure that the only way to solve the problem presented is with a trigger. If you do use a trigger, document that fact in as many places as you can to ensure that the existence of the trigger is taken into account by other developers and DBAs.

## Put Tables into In-Memory Storage

While there are a large number of limitations on in-memory storage mechanisms, the performance benefits are high. If you have a high-volume OLTP system and you're seeing lots of contention on I/O, especially around latches, the in-memory storage is a viable option. You may also want to explore using in-memory storage for table variables to help enhance their performance. If you have data that doesn't have to persist, you can even create the table in-memory using the `SCHEMA_ONLY` durability option. The general approach is to use the in-memory objects to help with high-throughput OLTP where you may have concurrency issues, as opposed to greater degrees of scans, and

so on, experienced in a data warehouse situation. All these methods lead to significant performance benefits. But remember, you must have the memory available to support these options. There's nothing magic here. You're enhancing performance by throwing significant amounts of memory, and therefore money, at the problem.

## Use Columnstore Indexes

If you're designing and building a data warehouse, the use of columnstore indexes is almost automatic. Most of your queries are likely to involve aggregations across large groups of data, so the columnstore index is a natural performance enhancer. However, don't forget about putting the nonclustered columnstore index to work in your OLTP systems where you get frequent analytical style queries. There is additional maintenance overhead, and it will increase the size of your databases, but the benefits are enormous. You can create a rowstore table, as defined by the clustered index, that can use columnstore indexes. You can also create a columnstore table, as defined by its clustered index, that can use rowstore indexes. Using both these mechanisms, you can ensure that you meet the most common query style, analytical versus OLTP, while still supporting the other.

# Configuration Settings

Here's a checklist of the server and database configurations settings that have a big impact on database performance:

- Memory configuration options
- Cost threshold for parallelism
- Max degree of parallelism
- Optimize for ad hoc workloads
- Blocked process threshold
- Database file layout
- Database compression

I cover these settings in more detail in the sections that follow.

## Memory Configuration Options

As explained in the "SQL Server Memory Management" section of Chapter 2, it is strongly recommended that the `max server memory` setting be configured to a nondefault value determined by the system configuration. These memory configurations of SQL Server are explained in detail in the "Memory Bottleneck Analysis" and "Memory Bottleneck Resolutions" sections of Chapter 2.

## Cost Threshold for Parallelism

On systems with multiple processors, the parallel execution of queries is possible. The default value for parallelism is 5. This represents a cost estimate by the optimizer of a five-second execution on the query. In most circumstances, I've found this value to be radically too low; in other words, a higher threshold for parallelism results in better performance. Testing on your system will help you determine the appropriate value. Suggesting a value for this can be considered somewhat dangerous, but I'm going to do it anyway. I'd begin testing with a value of 35 and see where things go from there. Even better, use the data from the Query Store to determine the average cost of all your queries and then go two to three standard deviations above that average for the value of the Cost Threshold for Parallelism. In that way, you're looking at 95 percent to 98 percent of your queries will not go parallel, but the ones that really need it will. Finally, remember what type of system you're running. An OLTP system is much more likely to benefit from a lot of queries using a minimal amount of CPU each, while an analytical system is much more likely to benefit from more of the queries using more CPU.

## Max Degree of Parallelism

When a system has multiple processors available, by default SQL Server will use all of them during parallel executions. To better control the load on the machine, you may find it useful to limit the number of processors used by parallel executions. Further, you may need to set the affinity so that certain processors are reserved for the operating system and other services running alongside SQL Server. OLTP systems may receive a benefit from disabling parallelism entirely, although that's a questionable choice. First try increasing the cost threshold for parallelism because, even in OLTP systems, there are queries that will benefit from parallel execution, especially maintenance jobs. You may also explore the possibility of using the Resource Governor to control some workloads.

## Optimize for Ad Hoc Workloads

If the primary calls being made to your system come in as ad hoc or dynamic T-SQL instead of through well-defined stored procedures or parameterized queries, such as you might find in some of the implementations of object-relational mapping (ORM) software, then turning on the `optimize for ad hoc workloads` setting will reduce the consumption of procedure cache because plan stubs are created for initial query calls instead of full execution plans. This is covered in detail in Chapter 18.

## Blocked Process Threshold

The `blocked process threshold` setting defines in seconds when a blocked process report is fired. When a query runs and exceeds the threshold, the report is fired. An alert, which can be used to send an e-mail or a text message, is also fired. Testing an individual system determines what value to set this to. You can monitor for this using events within Extended Events.

## Database File Layout

For easy reference, the following are the best practices you should consider when laying out database files:

- Place the data and transaction log files of a user database on different I/O paths. This allows the transaction log disk head to progress sequentially without being moved randomly by the nonsequential I/Os commonly used for the data files.
- Placing the transaction log on a dedicated disk also enhances data protection. If a database disk fails, you will be able to save the completed transactions until the point of failure by performing a backup of the transaction log. By using this last transaction log backup during the recovery process, you will be able to recover the database up to the point of failure. This is known as *point-in-time recovery*.
- Avoid RAID 5 for transaction logs because, for every write request, RAID 5 disk arrays incur twice the number of disk I/Os compared to RAID 1 or 10.

- You may choose RAID 5 for data files since even in a heavy OLTP system, the number of read requests is usually seven to eight times the number of write requests. Also, for read requests the performance of RAID 5 is similar to that of RAID 1 and RAID 10 with an equal number of total disks.
- Look into moving to a more modern disk subsystem like SSD or FusionIO.
- Have multiple files for tempdb. The general rule would be half or one-quarter the files for the number of logical processor cores. All allocations in tempdb now use uniform extents. You'll also see the files will automatically grow at the same size now.

For a detailed understanding of database file layout and RAID subsystems, please refer to the "Disk Bottleneck Resolutions" section of Chapter 3.

## Database Compression

SQL Server has supplied data compression since 2008 with the Enterprise and Developer editions of the product. This can provide a great benefit in space used and in performance as more data gets stored on a page. These benefits come at the cost of added overhead in the CPU and memory of the system; however, the benefits usually far outweigh the costs. Take this into account as you implement compression.

# Database Administration

For your reference, here is a short list of the performance-related database administrative activities that you should perform on a regular basis as part of the process of managing your database server:

- Keep the statistics up-to-date.
- Maintain a minimum amount of index defragmentation.
- Avoid automatic database functions such as `AUTOCLOSE` or `AUTOSHRINK`.

In the following sections, I cover the preceding activities in more detail.

**Note** For a detailed explanation of SQL Server 2017 administration needs and methods, please refer to the Microsoft SQL Server Books Online article "Database Engine Features and Tasks" (http://bit.ly/SIlz8d).

## Keep the Statistics Up-to-Date

The performance impact of database statistics is explained in detail in Chapter 13 (and in various places throughout the book); however, this short list will serve as a quick and easy reference for keeping your statistics up-to-date:

- Allow SQL Server to automatically maintain the statistics of the data distribution in the tables by using the default settings for the configuration parameters `AUTO_CREATE_ STATISTICS` and `AUTO_UPDATE_STATISTICS`.
- As a proactive measure, you can programmatically update the statistics of every database object on a regular basis as you determine it is needed and supported within your system. This practice partly protects your database from having outdated statistics in case the auto update statistics feature fails to provide a satisfactory result. In Chapter 13, I illustrate how to set up a SQL Server job to programmatically update the statistics on a regular basis.
- Remember that you also have the ability to update the statistics in an asynchronous fashion. This reduces the contention on stats as they're being updated; thus, if you have a system with fairly constant access, you can use this method to update the statistics more frequently. Async is more likely to be helpful if you're seeing waits on statistics updates.

**Note** Please ensure that the statistics update job is scheduled before the completion of the index defragmentation job, as explained later in this chapter.

## Maintain a Minimum Amount of Index Defragmentation

The following best practices will help you maintain a minimum amount of index defragmentation:

- Defragment a database on a regular basis during nonpeak hours.
- On a regular basis, determine the level of fragmentation on your indexes; then, based on that fragmentation, either rebuild the index or defrag the index by executing the defragmentation queries outlined in Chapter 14.
- Remember that very small tables don't need to be defragmented at all.
- Different rules may apply for very large databases when it comes to defragmenting indexes.
- If you have indexes that are only ever used for single seek operations, then fragmentation doesn't impact performance.
- In Azure SQL Database, it's much more important to only rebuild indexes if you really need to. Index rebuilds can use up a lot of I/O bandwidth and can lead to throttling.

Also remember that index fragmentation is much less of a problem than most people make it out to be. Some experts are even suggesting that defragmenting of indexes is a waste of time. While I still think there are benefits, that is on a situational basis, so be sure you're monitoring and measuring your performance metrics carefully so that you can tell whether defragmentation is a benefit.

## Avoid Database Functions Such As AUTO_CLOSE or AUTO_SHRINK

`AUTO_CLOSE` cleanly shuts down a database and frees all its resources when the last user connection is closed. This means all data and queries in the cache are automatically flushed. When the next connection comes in, not only does the database have to restart but all the data has to be reloaded into the cache. Also, stored procedures and the other queries have to be recompiled. That's an extremely expensive operation for most database systems. Leave `AUTO_CLOSE` set to the default of `OFF`.

AUTO_SHRINK periodically shrinks the size of the database. It can shrink the data files and, when in Simple Recovery mode, the log files. While doing this, it can block other processes, seriously slowing down your system. More often than not, file growth is also set to occur automatically on systems with AUTO_SHRINK enabled, so your system will be slowed down yet again when the data or log files have to grow. Further, you're going to see the physical file storage get fragmented at the operating system level, seriously impacting performance. Set your database sizes to an appropriate size, and monitor them for growth needs. If you must grow them automatically, do so by physical increments, not by percentages.

# Database Backup

Database backup is a broad topic and can't be given due justice in this query optimization book. Nevertheless, I suggest that when it comes to database performance, you be attentive to the following aspects of your database backup process:

- Differential and transaction log backup frequency
- Backup distribution
- Backup compression

The next sections go into more detail on these suggestions.

## Incremental and Transaction Log Backup Frequency

For an OLTP database, it is mandatory that the database be backed up regularly so that, in case of a failure, the database can be restored on a different server. For large databases, the full database backup usually takes a long time, so full backups cannot be performed often. Consequently, full backups are performed at widespread time intervals, with incremental backups and transaction log backups scheduled more frequently between two consecutive full backups. With the frequent incremental and transaction log backups set in place, if a database fails completely, the database can be restored up to a point in time.

Differential backups can be used to reduce the overhead of a full backup by backing up only the data that has changed since the last full backup. Because this is potentially much faster, it will cause less of a slowdown on the production system. Each situation is unique, so you need to find the method that works best for you. As a general rule, I recommend taking a weekly full backup and then daily differential backups. From there, you can determine the needs of your transaction log backups.

Frequently backing up of the transaction log adds a small amount of overhead to the server, especially during peak hours.

For most businesses, the acceptable amount of data loss (in terms of time) usually takes precedence over conserving the log-disk space or providing ideal database performance. Therefore, you must take into account the acceptable amount of data loss when scheduling the transaction log backup, as opposed to randomly setting the backup schedule to a low-time interval.

## Backup Scheduling Distribution

When multiple databases need to be backed up, you must ensure that all full backups are not scheduled at the same time so that the hardware resources are not hit at the same time. If the backup process involves backing up the databases to a central SAN disk array, then the full backups from all the database servers must be distributed across the backup time window so that the central backup infrastructure doesn't get slammed by too many backup requests at the same time. Flooding the central infrastructure with a great deal of backup requests at the same time forces the components of the infrastructure to spend a significant part of their resources just managing the excessive number of requests. This mismanaged use of the resources increases the backup durations significantly, causing the full backups to continue during peak hours and thus affecting the performance of the user requests.

To minimize the impact of the full backup process on database performance, you must first determine the nonpeak hours when full backups can be scheduled and then distribute the full backups across the nonpeak time window, as follows:

1. Identify the number of databases that must be backed up.
2. Prioritize the databases in order of their importance to the business.
3. Determine the nonpeak hours when the full database backups can be scheduled.
4. Calculate the time interval between two consecutive full backups as follows: Time interval = (Total backup time window) / (Number of full backups).

5. Schedule the full backups in order of the database priorities, with the first backup starting at the start time of the backup window and subsequent backups spread uniformly at the time intervals calculated in the preceding equation.

This uniform distribution of the full backups will ensure that the backup infrastructure is not flooded with too many backup requests at the same time, thereby reducing the impact of the full backups on the database performance.

## Backup Compression

For relatively large databases, the backup durations and backup file sizes usually become an issue. Long backup durations make it difficult to complete the backups within the administrative time windows and thus start affecting the end user's experience. The large size of the backup files makes space management for the backup files quite challenging, and it increases the pressure on the network when the backups are performed across the network to a central backup infrastructure. Compression also acts to speed up the backup process since fewer writes to the disk are needed.

The recommended way to optimize the backup duration, the backup file size, and the resultant network pressure is to use *backup compression*.

# Query Design

Here's a list of the performance-related best practices you should follow when designing the database queries:

- Use the command `SET NOCOUNT ON`.
- Explicitly define the owner of an object.
- Avoid *nonsargable* search conditions.
- Avoid arithmetic operators and functions on `WHERE` clause columns.
- Avoid optimizer hints.
- Stay away from nesting views.
- Ensure there are no implicit data type conversions.
- Minimize logging overhead.

- Adopt best practices for reusing execution plans.
- Adopt best practices for database transactions.
- Eliminate or reduce the overhead of database cursors.
- Use natively compile stored procedures.
- Take advantage of columnstore indexes for analytical queries

I further detail each best practice in the following sections.

## Use the Command SET NOCOUNT ON

As a rule, always use the command `SET NOCOUNT ON` as the first statement in stored procedures, triggers, and other batch queries. This enables you to avoid the network overhead associated with the return of the number of rows affected after every execution of a SQL statement. The command `SET NOCOUNT` is explained in detail in the "Use SET NOCOUNT" section of Chapter 20.

## Explicitly Define the Owner of an Object

As a performance best practice, always qualify a database object with its owner to avoid the runtime cost required to verify the owner of the object. The performance benefit of explicitly qualifying the owner of a database object is explained in detail in the "Do Not Allow Implicit Resolution of Objects in Queries" section of Chapter 16.

## Avoid Nonsargable Search Conditions

Be vigilant when defining the search conditions in your query. If the search condition on a column used in the `WHERE` clause prevents the optimizer from effectively using the index on that column, then the execution cost for the query will be high in spite of the presence of the correct index. The performance impact of nonsargable search conditions is explained in detail in the corresponding section of Chapter 19.

Additionally, please be careful about providing too much flexibility on search capabilities. If you define an application feature such as "retrieve all products with product name ending in caps," then you will have queries scanning the complete table (or the clustered index). As you know, scanning a multimillion-row table will hurt your database performance. Unless you use an index hint, you won't be able to benefit from