You can use the Performance Monitor that is built into Windows to create a baseline for SQL Server's hardware and software resource utilization. You can also get snapshots of this information by using dynamic management views and dynamic management functions. Similarly, you may baseline the SQL Server query workload using Extended Events, which can help you understand the average resource utilization and execution time of SQL queries when conditions are stable. You will learn in detail how to use these tools and queries in Chapters 2–5. A platform system may have different measures such as the Database Transaction Unit (DTU) of the Azure SQL Database.

Another option is to take advantage of one of the many tools that can generate an artificial load on a given server or database. Numerous third-party tools are available. Microsoft offers Distributed Replay, which is covered at length in Chapter 25.

Where to Focus Efforts

When you tune a particular system, pay special attention to the data access layer (the database queries and stored procedures executed by your code or through your object-relational mapping engine that are used to access the database). You will usually find that you can positively affect performance in the data access layer far more than if you spend an equal amount of time figuring out how to tune the hardware, operating system, or SQL Server configuration. Although a proper configuration of the hardware, operating system, and SQL Server instance is essential for the best performance of a database application, these areas of expertise have standardized so much that you usually need to spend only a limited amount of time configuring the systems properly for performance. Application design issues such as query design and indexing strategies, on the other hand, are unique to your code and data set. Consequently, there is usually more to optimize in the data access layer than in the hardware, operating system, SQL Server configuration, or platform. Figure 1-3 shows the results of a survey of 346 data professionals (with permission from Paul Randal: `http://bit.ly/1gRANRy`).

What were the root causes of the last few SQL Server performance problems you debugged?
(Vote multiple times if you want!)

CPU power saving	2%	6
Other hardware or OS issue	2%	7
Virtualization	2%	7
SQL Server/database configuration	3%	10
Out-of-date/missing statistics	9%	31
Database/table structure/schema design	10%	38
Application code	12%	43
I/O subsystem problem	16%	60
Poor indexing strategy	19%	68
T-SQL code	26%	94
	Total: 364 responses	

Figure 1-3. *Root causes of performance problems*

As you can see, the two most common issues are T-SQL code and poor indexing. Four of the six most common issues are all directly related to the T-SQL, indexes, code, and data structure. My experience matches that of the other respondents. You can obtain the greatest improvement in database application performance by looking first at the area of data access, including logical/physical database design, query design, and index design.

Sure, if you concentrate on hardware configuration and upgrades, you may obtain a satisfactory performance gain. However, a bad SQL query sent by the application can consume all the hardware resources available, no matter how much you have. Therefore, a poor application design can make hardware upgrade requirements very high, even beyond your cost limits. In the presence of a heavy SQL workload, concentrating on hardware configurations and upgrades usually produces a poor return on investment.

You should analyze the stress created by an application on a SQL Server database at two levels.

- *High level*: Analyze how much stress the database application is creating on individual hardware resources and the overall behavior of the SQL Server installation. The best measures for this are the various wait states and the DTUs of a platform like Azure. This information can help you in two ways. First, it helps you identify the area to concentrate on within a SQL Server application where there is poor performance. Second, it helps you identify any lack of proper configuration at the higher levels. You can then decide which hardware resource may be upgraded.
- *Low level*: Identify the exact culprits within the application—in other words, the SQL queries that are creating most of the pressure visible at the overall higher level. This can be done using the Extended Events tool and various dynamic management views, as explained in Chapter 6.

SQL Server Performance Killers

Let's now consider the major problem areas that can degrade SQL Server performance. By being aware of the main performance killers in SQL Server in advance, you will be able to focus your tuning efforts on the likely causes.

Once you have optimized the hardware, operating system, and SQL Server settings, the main performance killers in SQL Server are as follows, in a rough order (with the worst appearing first):

- Insufficient or inaccurate indexing
- Inaccurate statistics
- Improper query design
- Poorly generated execution plans
- Excessive blocking and deadlocks
- Non-set-based operations, usually T-SQL cursors
- Inappropriate database design

- Recompiling execution plans
- Frequent recompilation of queries
- Improper use of cursors
- Excessive index fragmentation

Let's take a quick look at each of these issues.

Insufficient or Inaccurate Indexing

Insufficient indexing is usually one of the biggest performance killers in SQL Server. As bad, and sometimes worse, is having the wrong indexes. In the absence of proper indexing for a query, SQL Server has to retrieve and process much more data while executing the query. This causes high amounts of stress on the disk, memory, and CPU, increasing the query execution time significantly. Increased query execution time then can lead to excessive blocking and deadlocks in SQL Server. You will learn how to determine indexing strategies and resolve indexing problems in Chapters 8–12.

Generally, indexes are considered to be the responsibility of the database administrator (DBA). However, the DBA can't proactively define how to use the indexes since the use of indexes is determined by the database queries and stored procedures written by the developers. Therefore, defining the indexes must be a shared responsibility since the developers usually have more knowledge of the data to be retrieved and the DBAs have a better understanding of how indexes work. Indexes created without the knowledge of the queries serve little purpose.

Too many or just the wrong indexes cause just as many problems. Lots of indexes will slow down data manipulation through INSERTs, UPDATEs, and DELETEs since the indexes have to be maintained. Slower performance leads to excessive blocking and once again deadlocks. Incorrect indexes just aren't used by the optimizer but still must be maintained, paying that cost in processing power, disk storage, and memory.

Note Because indexes created without the knowledge of the queries serve little purpose, database developers need to understand indexes at least as well as they know T-SQL.

Inaccurate Statistics

SQL Server relies heavily on cost-based optimization, so accurate data distribution statistics are extremely important for the effective use of indexes. Without accurate statistics, SQL Server's query optimizer can't accurately estimate the number of rows affected by a query. Because the amount of data to be retrieved from a table is highly important in deciding how to optimize the query execution, the query optimizer is much less effective if the data distribution statistics are not maintained accurately. Statistics can age without being updated. You can also see issues around data being distributed in a skewed fashion hurting statistics. Statistics on columns that auto-increment a value, such as a date, can be out-of-date as new data gets added. You will look at how to analyze statistics in Chapter 13.

Improper Query Design

The effectiveness of indexes depends in large part on the way you write SQL queries. Retrieving excessively large numbers of rows from a table or specifying a filter criterion that returns a larger result set from a table than is required can render the indexes ineffective. To improve performance, you must ensure that the SQL queries are written to make the best use of new or existing indexes. Failing to write cost-effective SQL queries may prevent the optimizer from choosing proper indexes, which increases query execution time and database blocking. Chapter 19 covers how to write effective queries in specific detail.

Query design covers not only single queries but also sets of queries often used to implement database functionalities such as a queue management among queue readers and writers. Even when the performance of individual queries used in the design is fine, the overall performance of the database can be very poor. Resolving this kind of bottleneck requires a broad understanding of different characteristics of SQL Server, which can affect the performance of database functionalities. You will see how to design effective database functionality using SQL queries throughout the book.

Poorly Generated Execution Plans

The same mechanisms that allow SQL Server to establish an efficient execution plan and reuse that plan again and again instead of recompiling can, in some cases, work against you. A bad execution plan can be a real performance killer. Inaccurate and poorly

performing plans are frequently caused when a process called *parameter sniffing* goes bad. Parameter sniffing is a process that comes from the mechanisms that the query optimizer uses to determine the best plan based on sampled or specific values from the statistics. It's important to understand how statistics and parameters combine to create execution plans and what you can do to control them. Statistics are covered in Chapter 13, and execution plan analysis is covered in Chapters 15 and 16. Chapter 17 focuses only on bad parameter sniffing and how best to deal with it (along with some of the details from Chapter 11 on the Query Store and Plan Forcing).

Excessive Blocking and Deadlocks

Because SQL Server is fully atomicity, consistency, isolation, and durability (ACID) compliant, the database engine ensures that modifications made by concurrent transactions are properly isolated from one another. By default, a transaction sees the data either in the state before another concurrent transaction modified the data or after the other transaction completed—it does not see an intermediate state.

Because of this isolation, when multiple transactions try to access a common resource concurrently in a noncompatible way, *blocking* occurs in the database. Two processes can't update the same piece of data the same time. Further, since all the updates within SQL Server are founded on a page of data, 8KB worth of rows, you can see blocking occurring even when two processes aren't updating the same row. Blocking is a good thing in terms of ensuring proper data storage and retrieval, but too much of it in the wrong place can slow you down.

Related to blocking but actually a separate issue, a *deadlock* occurs when two resources attempt to escalate or expand locked resources and conflict with one another. The query engine determines which process is the least costly to roll back and chooses it as the *deadlock victim.* This requires that the database request be resubmitted for successful execution. Deadlocks are a fundamental performance problem even though many people think of them as a structural issue. The execution time of a query is adversely affected by the amount of blocking and deadlocks, if any, it faces.

For scalable performance of a multiuser database application, properly controlling the isolation levels and transaction scopes of the queries to minimize blocking and deadlocks is critical; otherwise, the execution time of the queries will increase significantly, even though the hardware resources may be highly underutilized. I cover this problem in depth in Chapters 21 and 22.

Non-Set-Based Operations

Transact-SQL is a set-based language, which means it operates on sets of data. This forces you to think in terms of columns rather than in terms of rows. Non-set-based thinking leads to excessive use of cursors and loops rather than exploring more efficient joins and subqueries. The T-SQL language offers rich mechanisms for manipulating sets of data. For performance to shine, you need to take advantage of these mechanisms rather than force a row-by-row approach to your code, which will kill performance. Examples of how to do this are available throughout the book; also, I address T-SQL best practices in Chapter 19 and cursors in Chapter 23.

Inappropriate Database Design

A database should be adequately normalized to increase the performance of data retrieval and reduce blocking. For example, if you have an undernormalized database with customer and order information in the same table, then the customer information will be repeated in all the order rows of the customer. This repetition of information in every row will increase the number of page reads required to fetch all the orders placed by a customer. At the same time, a data writer working on a customer's order will reserve all the rows that include the customer information and thus could block all other data writers/data readers trying to access the customer profile.

Overnormalization of a database can be as bad as undernormalization. Overnormalization increases the number and complexity of joins required to retrieve data. An overnormalized database contains a large number of tables with a small number of columns. Overnormalization is not a problem I've run into a lot, but when I've seen it, it seriously impacts performance. It's much more common to be dealing with undernormalization or improper normalization of your structures.

Having too many joins in a query may also be because database entities have not been partitioned distinctly or the query is serving a complex set of requirements that could perhaps be better served by creating a new stored procedure.

Another issue with database design is actually implementing primary keys, unique constraints, and enforced foreign keys. Not only do these mechanisms ensure data consistency and accuracy, but the query optimizer can take advantage of them when making decisions about how to resolve a particular query. All too often though people ignore creating a primary key or disable their foreign keys, either directly or through the use of `WITH NO_CHECK`. Without these tools, the optimizer has no choice but to create suboptimal plans.

Database design is a large subject. I will provide a few pointers in Chapter 19 and throughout the rest of the book. Because of the size of the topic, I won't be able to treat it in the complete manner it requires. However, if you want to read a book on database design with an emphasis on introducing the subject, I recommend reading *Pro SQL Server 2012 Relational Database Design and Implementation* by Louis Davidson et al. (Apress, 2012).

Recompiling Execution Plans

To execute a query in an efficient way, SQL Server's query optimizer spends a fair amount of CPU cycles creating a cost-effective execution plan. The good news is that the plan is cached in memory, so you can reuse it once created. However, if the plan is designed so that you can't plug parameter values into it, SQL Server creates a new execution plan every time the same query is resubmitted with different values. So, for better performance, it is extremely important to submit SQL queries in forms that help SQL Server cache and reuse the execution plans. I will also address topics such as plan freezing, forcing query plans, and using "optimize for ad hoc workloads." You will see in detail how to improve the reusability of execution plans in Chapter 16.

Frequent Recompilation of Queries

One of the standard ways of ensuring a reusable execution plan, independent of values used in a query, is to use a stored procedure or a parameterized query. Using a stored procedure to execute a set of SQL queries allows SQL Server to create a parameterized execution plan.

A *parameterized execution plan* is independent of the parameter values supplied during the execution of the stored procedure or parameterized query, and it is consequently highly reusable. Frequent recompilation of queries increases pressure on the CPU and the query execution time. I will discuss in detail the various causes and resolutions of stored procedure, and statement, recompilation in Chapter 18.

Improper Use of Cursors

By preferring a cursor-based (row-at-a-time) result set—or as Jeff Moden has so aptly termed it, Row By Agonizing Row (RBAR; pronounced "ree-bar")—instead of a regular set-based SQL query, you add a large amount of overhead to SQL Server. Use set-based queries whenever possible, but if you are forced to deal with cursors, be sure to use efficient cursor types such as fast-forward only. Excessive use of inefficient cursors increases stress on SQL Server resources, slowing down system performance. I discuss how to work with cursors properly, if you must, in Chapter 23.

Excessive Index Fragmentation

While analyzing data retrieval operations, you can usually assume that the data is organized in an orderly way, as indicated by the index used by the data retrieval operation. However, if the pages containing the data are fragmented in a nonorderly fashion or if they contain a small amount of data because of frequent page splits, then the number of read operations required by the data retrieval operation will be much higher than might otherwise be required. The increase in the number of read operations caused by fragmentation hurts query performance. In Chapter 14, you will learn how to analyze and remove fragmentation. However, it doesn't hurt to mention that there is a lot of new thought around index fragmentation that it may not be a problem at all. You'll need to evaluate your system to check whether this is a problem.

Summary

In this introductory chapter, you saw that SQL Server performance tuning is an iterative process, consisting of identifying performance bottlenecks, troubleshooting their cause, applying different resolutions, quantifying performance improvements, and then repeating these steps until your required performance level is reached.
To assist in this process, you should create a system baseline to compare with your modifications. Throughout the performance tuning process, you need to be objective about the amount of tuning you want to perform—you can always make a query run a little bit faster, but is the effort worth the cost? Finally, since performance depends on the pattern of user activity and data, you must reevaluate the database server performance on a regular basis.

To derive the optimal performance from a SQL Server database system, it is extremely important that you understand the stresses on the server created by the database application. In the next three chapters, I discuss how to analyze these stresses, both at a higher system level and at a lower SQL Server activities level. Then I show how to combine the two.

In the rest of the book, you will examine in depth the biggest SQL Server performance killers, as mentioned earlier in the chapter. You will learn how these individual factors can affect performance if used incorrectly and how to resolve or avoid these traps.

CHAPTER 2

Memory Performance Analysis

A system can directly impact SQL Server and the queries running on it in three primary places: memory, disk, and CPU. You're going to explore each of these in turn starting, in this chapter, with memory. Queries retrieving data in SQL Server must first load that data into memory. Any changes to data are first loaded into memory where the modifications are made, prior to writing them to disk. Many other operations take advantage of the speed of memory in the system, such as sorting data using an ORDER BY clause in a query, performing calculations to create hash tables when joining two tables, and putting the tables in memory through the in-memory OLTP table functions. Because all this work is being done within the memory of the system, it's important that you understand how memory is being managed.

In this chapter, I cover the following topics:

- The basics of the Performance Monitor tool
- Some of the dynamic management objects used to observe system behavior
- How and why hardware resources can be bottlenecks
- Methods of observing and measuring memory use within SQL Server and Windows
- Methods of observing and measuring memory use in Linux
- Possible resolutions to memory bottlenecks

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_2

Performance Monitor Tool

Windows Server 2016 provides a tool called Performance Monitor, which collects detailed information about the utilization of operating system resources. It allows you to track nearly every aspect of system performance, including memory, disk, processor, and the network. In addition, SQL Server 2017 provides extensions to the Performance Monitor tool that track a variety of functional areas within SQL Server.

Performance Monitor tracks resource behavior by capturing performance data generated by hardware and software components of the system, such as a processor, a process, a thread, and so on. The performance data generated by a system component is represented by a performance object. The performance object provides counters that represent specific aspects of a component, such as % Processor Time for a Processor object. Just remember, when running these counters within a virtual machine (VM), the performance measured for the counters in many instances, depending on the type of counter, is for the VM, not the physical server. That means some values collected on a VM are not going to accurately reflect physical reality.

There can be multiple instances of a system component. For instance, the Processor object in a computer with two processors will have two instances, represented as instances 0 and 1. Performance objects with multiple instances may also have an instance called Total to represent the total value for all the instances. For example, the processor usage of a computer with two processors can be determined using the following performance object, counter, and instance (as shown in Figure 2-1):

- *Performance object*: Processor
- *Counter*: **%** Processor Time
- *Instance*: _Total

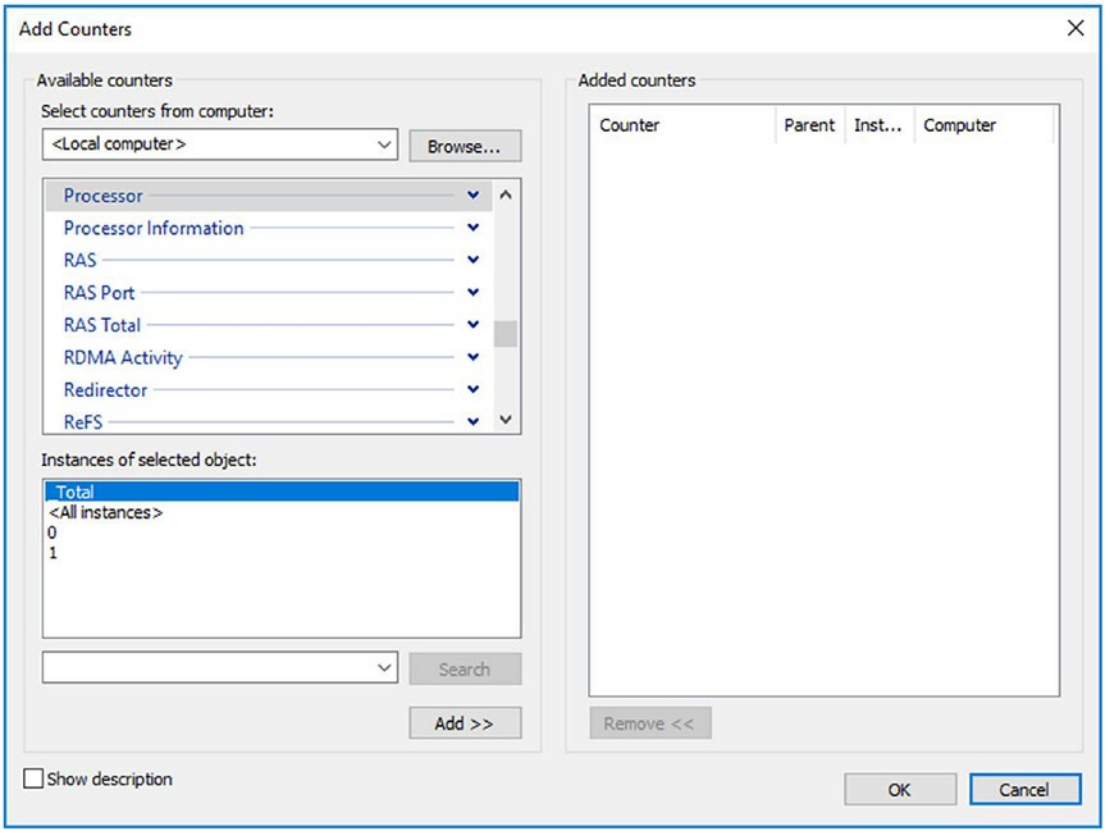

Figure 2-1. *Adding a Performance Monitor counter*

System behavior can be either tracked in real time in the form of graphs or captured as a file (called a *data collector set*) for offline analysis. The preferred mechanism on production servers is to use the file. You'll want to collect the information in a file to store it and transmit it as needed over time. Plus, writing the collection to a file takes up fewer resources than collecting it on the screen in active memory.

To run the Performance Monitor tool, execute `perfmon` from a command prompt, which will open the Performance Monitor suite. You can also right-click the Computer icon on the desktop or the Start menu, expand Diagnostics, and then expand the Performance Monitor. You can also go to the Start screen and start typing **Performance Monitor**; you'll see the icon for launching the application. Any of these methods will allow you to open the Performance Monitor utility.

You will learn how to set up the individual counters in Chapter 5. Now that I've introduced the concept of the Performance Monitor, I'll introduce another metric-gathering interface, dynamic management views.

Dynamic Management Views

To get an immediate snapshot of a large amount of data that was formerly available only in Performance Monitor, SQL Server offers some of the same data, plus a lot of different information, internally through a set of dynamic management views (DMVs) and dynamic management functions (DMFs), collectively referred to as *dynamic management views* (documentation used to refer to *objects*, but that has changed). These are extremely useful mechanisms for capturing a snapshot of the current performance of your system. I'll introduce several DMVs throughout the book, but for now I'll focus on a few that are the most important for monitoring server performance and for establishing a baseline.

The `sys.dm_os_performance_counters` view displays the SQL Server counters within a query, allowing you to apply the full strength of T-SQL to the data immediately. For example, this simple query will return the current value for Logins/sec:

```
SELECT  dopc.cntr_value,
        dopc.cntr_type
FROM    sys.dm_os_performance_counters AS dopc
WHERE   dopc.object_name =  'SQLServer:General Statistics'
        AND dopc.counter_name =  'Logins/sec';
```

This returns the value of 46 for my test server. For your server, you'll need to substitute the appropriate server name in the `object_name` comparison if you have a named instance, for example `MSSQL$SQL1-General Statistics`. Worth noting is the `cntr_type` column. This column tells you what type of counter you're reading (documented by Microsoft at `http://bit.ly/1mmcRaN`). For example, the previous counter returns the value 272696576, which means that this counter is an average value. There are values that are moments-in-time snapshots, accumulations since the server started, and others. Knowing what the measure represents is an important part of understanding these metrics.

There are a large number of DMVs that can be used to gather information about the server. I'll introduce one more here that you will find yourself accessing on a regular basis, `sys.dm_os_wait_stats`. This DMV shows aggregated wait times within SQL Server on various resources, collected since the last time SQL Server was started, the last time it failed over, or the counters were reset. The wait times are recorded after the work is completed, so these numbers don't reflect any active threads. Identifying the types of

waits that are occurring within your system is one of the easiest mechanisms to begin identifying the source of your bottlenecks. You can sort the data in various ways; this first example looks at the waits that have the longest current count using this simple query:

```
SELECT TOP(10)
    dows.*
FROM sys.dm_os_wait_stats AS dows
ORDER BY dows.wait_time_ms DESC;
```

Figure 2-2 displays the output.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	SLEEP_TASK	13146	10527380	8749	42129
2	DIRTY_PAGE_POLL	35844	4155158	5108	593
3	LOGMGR_QUEUE	30214	4154589	4515	1588
4	HADR_FILESTREAM_IOMGR_IOCOMPLETION	8012	4154564	5011	12197
5	LAZYWRITER_SLEEP	4121	4154119	7795	23135
6	SQLTRACE_INCREMENTAL_FLUSH_SLEEP	1034	4153952	7891	13
7	REQUEST_FOR_DEADLOCK_SEARCH	829	4152916	5414	4152916
8	XE_TIMER_EVENT	1190	4152398	6110	4112775
9	QDS_PERSIST_TASK_MAIN_LOOP_SLEEP	70	4146281	64541	667
10	XE_DISPATCHER_WAIT	47	4080524	120445	0

Figure 2-2. *Output from sys.dm_os_wait_stats*

You can see not only the cumulative time that particular waits have accumulated but also a count of how often they have occurred and the maximum time that something had to wait. From here, you can identify the wait type and begin troubleshooting. One of the most common types of waits is I/O. If you see ASYNC_IO_C0MPLETI0N, IO_C0MPLETION, LOGMGR, WRITELOG, or PAGEIOLATCH in your top ten wait types, you may be experiencing I/O contention, and you now know where to start working. The previous list includes quite a few waits that basically qualify as noise. A common practice is to eliminate them. However, there are a lot of them. The easiest method for dealing with that is to lean on Paul Randals scripts from this article: "Wait statistics, or please tell me where it hurts" (http://bit.ly/2wsQHQE). Also, you can now see aggregated wait statistics for individual queries in the information captured by the Query Store, which we'll cover in Chapter 11. You can always find information about more obscure wait types by going directly to Microsoft through MSDN support (http://bit.ly/2vAWAfP). Finally, Paul Randal also maintains a library of wait types (collected at http://bit.ly/2ePzYO2).

Hardware Resource Bottlenecks

Typically, SQL Server database performance is affected by stress on the following hardware resources:

- Memory
- Disk I/O
- Processor
- Network

Stress beyond the capacity of a hardware resource forms a bottleneck. To address the overall performance of a system, you need to identify these bottlenecks because they form the limit on overall system performance. Further, when you clear one bottleneck, you may find that you have others since one set of bad behaviors masks or limits other sets.

Identifying Bottlenecks

There is usually a relationship between resource bottlenecks. For example, a processor bottleneck may be a symptom of excessive paging (memory bottleneck) or a slow disk (disk bottleneck) caused by bad execution plans. If a system is low on memory, causing excessive paging, and has a slow disk, then one of the end results will be a processor with high utilization since the processor has to spend a significant number of CPU cycles to swap pages in and out of the memory and to manage the resultant high number of I/O requests. Replacing the processors with faster ones may help a little, but it is not the best overall solution. In a case like this, increasing memory is a more appropriate solution because it will decrease pressure on the disk and processor. In fact, upgrading the disk is probably a better solution than upgrading the processor. If you can, decreasing the workload could also help, and, of course, tuning the queries to ensure maximum efficiency is also an option.

One of the best ways of locating a bottleneck is to identify resources that are waiting for some other resource to complete its operation. You can use Performance Monitor counters or DMVs such as `sys.dm_os_wait_stats` to gather that information. The response time of a request served by a resource includes the time the request had to wait in the resource queue as well as the time taken to execute the request, so end user response time is directly proportional to the amount of queuing in a system.

Another way to identify a bottleneck is to reference the response time and capacity of the system. The amount of throughput, for example, to your disks should normally be something approaching what the vendor suggests the disk is capable of. So, measuring information such as disk sec/transfer will indicate when disks are slowing down because of excessive load.

Not all resources have specific counters that show queuing levels, but most resources have some counters that represent an overcommittal of that resource. For example, memory has no such counter, but a large number of hard page faults represents the overcommittal of physical memory (hard page faults are explained later in the chapter in the section "Pages/Sec and Page Faults/Sec"). Other resources, such as the processor and disk, have specific counters to indicate the level of queuing. For example, the counter Page Life Expectancy indicates how long a page will stay in the buffer pool without being referenced. This indicates how well SQL Server is able to manage its memory since a longer life means that a piece of data in the buffer will be there, available, waiting for the next reference. However, a shorter life means that SQL Server is moving pages in and out of the buffer quickly, possibly suggesting a memory bottleneck.

You will see which counters to use in analyzing each type of bottleneck shortly.

Bottleneck Resolution

Once you have identified bottlenecks, you can resolve them in two ways.

- You can increase resource capacity.
- You can decrease the arrival rate of requests to the resource.

Increasing the capacity usually requires extra resources such as memory, disks, processors, or network adapters. You can decrease the arrival rate by being more selective about the requests to a resource. For example, when you have a disk subsystem bottleneck, you can either increase the capacity of the disk subsystem or decrease the number of I/O requests.

Increasing the capacity means adding more disks or upgrading to faster disks. Decreasing the arrival rate means identifying the cause of high I/O requests to the disk subsystem and applying resolutions to decrease their number. You may be able to decrease the I/O requests, for example, by adding appropriate indexes on a table to limit the amount of data accessed or by writing the T-SQL statement to include more or better filters in the WHERE clause.

Memory Bottleneck Analysis

Memory can be a problematic bottleneck because a bottleneck in memory will manifest on other resources, too. This is particularly true for a system running SQL Server. When SQL Server runs out of cache (or memory), a process within SQL Server (called *lazy writer*) has to work extensively to maintain enough free internal memory pages within SQL Server. This consumes extra CPU cycles and performs additional physical disk I/O to write memory pages back to disk.

SQL Server Memory Management

SQL Server manages memory for databases, including memory requirements for data and query execution plans, in a large pool of memory called the *buffer pool*. The memory pool used to consist of a collection of 8KB buffers to manage memory. Now there are multiple page allocations for data pages and plan cache pages, free pages, and so forth. The buffer pool is usually the largest portion of SQL Server memory. SQL Server manages memory by growing or shrinking its memory pool size dynamically.

You can configure SQL Server for dynamic memory management in SQL Server Management Studio (SSMS). Go to the Memory folder of the Server Properties dialog box, as shown in Figure 2-3.

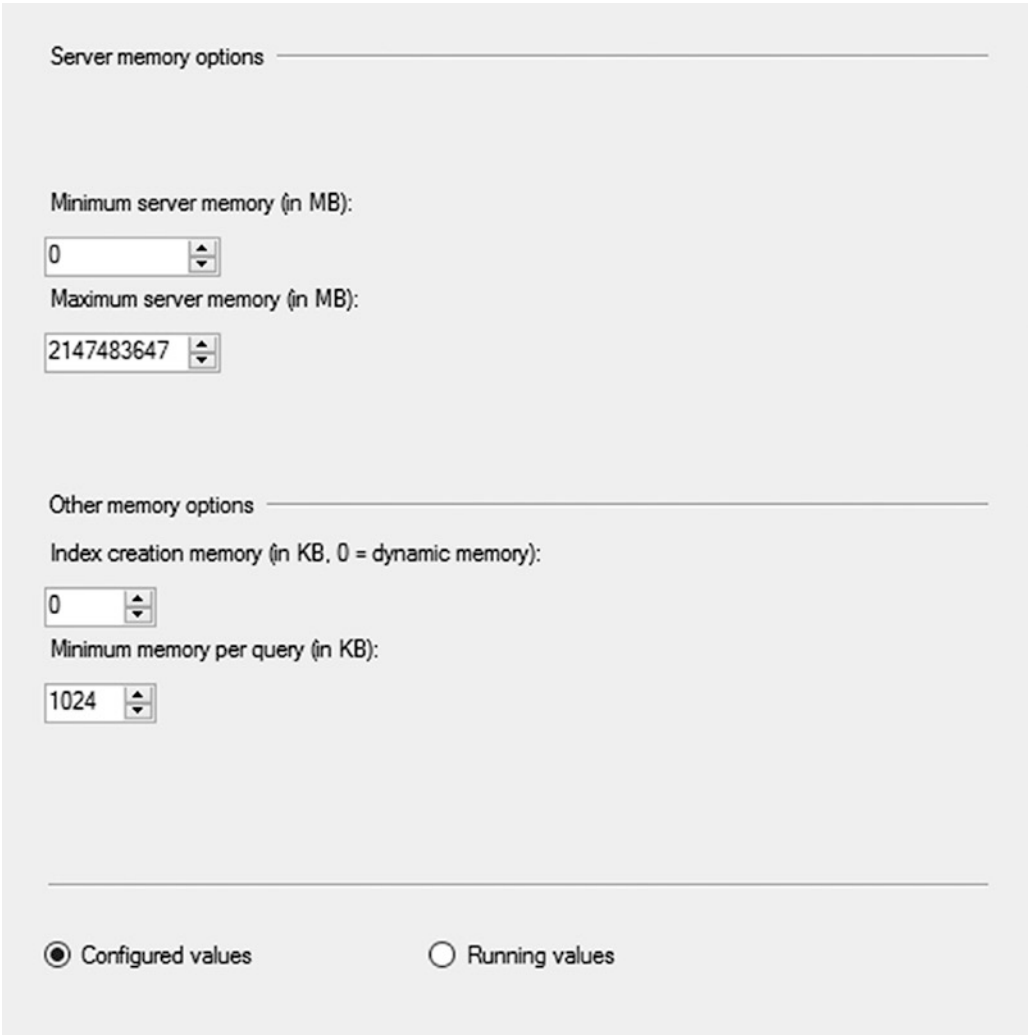

Figure 2-3. *SQL Server memory configuration*

The dynamic memory range is controlled through two configuration properties: Minimum(MB) and Maximum(MB).

- Minimum(MB), also known as *min server memory*, works as a floor value for the memory pool. Once the memory pool reaches the same size as the floor value, SQL Server can continue committing pages in the memory pool, but it can't be shrunk to less than the floor value. Note that SQL Server does not start with the min server memory configuration value but commits memory dynamically, as needed.

- Maximum(MB), also known as *max server memory*, serves as a ceiling value to limit the maximum growth of the memory pool. These configuration settings take effect immediately and do not require a restart. In SQL Server 2017 the lowest maximum memory is 512MB for Express Edition and 1GB for all others when running on Windows. The memory requirement on Linux is 3.5GB.

Microsoft recommends that you use dynamic memory configuration for SQL Server, where min server memory is 0 and max server memory is set to allow some memory for the operating system, assuming a single instance on the machine. The amount of memory for the operating system depends first on the type of OS and then on the size of the server being configured.

In Windows, for small systems with 8GB to 16GB of memory, you should leave about 2GB to 4GB for the OS. As the amount of memory in your server increases, you'll need to allocate more memory for the OS. A common recommendation is 4GB for every 16GB beyond 32GB of total system memory. You'll need to adjust this depending on your own system's needs and memory allocations. You should not run other memory-intensive applications on the same server as SQL Server, but if you must, I recommend you first get estimates on how much memory is needed by other applications and then configure SQL Server with a max server memory value set to prevent the other applications from starving SQL Server of memory. On a server with multiple SQL Server instances, you'll need to adjust these memory settings to ensure each instance has an adequate value. Just make sure you've left enough memory for the operating system and external processes.

In Linux, the general guidance is to leave about 20 percent of memory on the system for the operating system. The same types of processing needs are going to apply as the OS needs memory to manage its various resources in support of SQL Server.

Memory within SQL Server, regardless of the OS, can be roughly divided into buffer pool memory, which represents data pages and free pages, and nonbuffer memory, which consists of threads, DLLs, linked servers, and others. Most of the memory used by SQL Server goes into the buffer pool. But you can get allocations beyond the buffer pool, known as *private bytes*, which can cause memory pressure not evident in the normal process of monitoring the buffer pool. Check Process: sqlservr: Private Bytes in comparison to SQL Server: Buffer Manager: Total pages if you suspect this issue on your system.