About the Author

Grant Fritchey, Microsoft Data Platform MVP, has more than 20 years of experience in IT. That time was spent in technical support, development, and database administration. He currently works as a product evangelist at Red Gate Software. Grant writes articles for publication at SQL Server Central and Simple-Talk. He has published books, including *SQL Server Execution Plans* and *SQL Server 2012 Query Performance Tuning* (Apress). He has written chapters for *Beginning SQL Server 2012 Administration* (Apress)*, SQL Server Team-based Development, SQL Server MVP Deep Dives Volume 2, Pro SQL Server 2012 Practices* (Apress)*,* and *Expert Performance Indexing in SQL Server* (Apress). Grant currently serves as the president on the board of directors of the PASS organization, the leading source of educational content and training on the Microsoft data platform.

About the Technical Reviewer

Joseph Sack is a principal program manager at Microsoft, focusing on query processing for Azure SQL Database and SQL Server. He has worked as a SQL Server professional since 1997 and has supported and developed for SQL Server environments in financial services, IT consulting, manufacturing, retail, and the real estate industry.

Joe joined Microsoft in 2006 and was a SQL Server premier field engineer for large retail customers in Minneapolis, Minnesota. He was responsible for providing deep SQL Server advisory services, training, troubleshooting, and ongoing solutions guidance. In 2006 Joe earned the Microsoft Certified Master: SQL Server 2005 certification, and in 2008 he earned the Microsoft Certified Master: SQL Server 2008 certification. In 2009 he took over responsibility for the entire SQL Server Microsoft Certified Master program and held that post until 2011.

He left Microsoft in late 2011 to join SQLskills, working as a principal consultant. During that time, he co-instructed for various training events and was a consultant for customer performance tuning engagements. He recorded 13 Pluralsight courses, including *SQL Server: Troubleshooting Query Plan Quality Issues*, *SQL Server: Transact-SQL Basic Data Retrieval*, and *SQL Server: Common Query Tuning Problems and Solutions*. He returned to Microsoft in 2015.

Over the years Joe has published and edited several SQL Server books and white papers. His first book, *SQL Server 2000 Fast Answers for DBAs and Developers*, was published in 2003. He also started and maintained the T-SQL Recipe series, including *SQL Server 2005 T-SQL Recipes* and *SQL Server 2008 Transact-SQL Recipes*.

His most popular white papers include “Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator” and “AlwaysOn Architecture Guide: Building a High Availability and Disaster Recovery Solution by Using Failover Cluster Instances and

Availability Groups." Currently he writes (along with his colleagues) on the SQL Server Engine Blog. His classic posts can still be found at `https://www.sqlskills.com/blogs/joe/` and `https://blogs.msdn.microsoft.com/joesack/`.

His Twitter handle is `@JoeSackMSFT`, and you can find Joe speaking at most major SQL Server conferences. He spends half his time between Minneapolis and Seattle, meaning that he is either cold or wet at any given point in time.

Acknowledgments

The poor editors at Apress have to put up with me and my bad scheduling, so first and foremost, thanks to Jill Balzano and Jonathan Gennick. To say I couldn’t have done it without you doesn’t begin to cover it. I’ve said it before and I’ll say it again here, publicly and forever in print, Joe Sack is my hero. Thanks for everything, Joe.

Introduction

Technology is changing all the time, faster and faster. New functionality is introduced in Azure SQL Database on an almost weekly schedule, and SQL Server itself has gone through two releases since the last edition of this book was published. New styles of databases are introduced all the time. With all this change, the question immediately in front of you should be, do we still need to do query tuning?

The answer is a very short and resounding, yes.

With all the functionality and capability built into SQL Server and the Azure Data Platform, not only is query tuning still an important skill, it actually becomes a way to save your organization money. Knowing how to make a query run faster so that fewer resources are needed literally becomes a way to reduce costs within a platform-as-a-service offering such as Azure SQL Database.

However, it's not just about money. The code generated by object-relational mapping tools such as Entity Framework can be fantastic, until it isn't. Then, you'll be working on creating custom scripts and generating data structures and indexes and all the rest of traditional query performance tuning.

While technology has certainly moved fast and far, there is still a fundamental need to get queries to run faster and do more with less overhead on your servers. That's where this book comes into play. This is a resource that you can use to ensure that you're using all the tools in your hands to ensure that the databases you build, develop, and maintain will continue to run faster.

Who Is This Book For?

If you write or generate T-SQL, you're going to need to make it run faster. So, this book is for data analysts, developers, coders, database designers, database developers, and that last bastion of protection for the company's information, the database administrator. You'll all need at one point or another to understand how indexes work, where to track down performance metrics, and methods and mechanisms to ensure that your queries run as fast as they can.

The code for the book is available from Apress.com. If you have questions, want suggestions, or just need a little help, you can get in touch with me at `grant@scarydba.com`.

CHAPTER 1

SQL Query Performance Tuning

Query performance tuning continues to be a fundamental aspect of modern database maintenance and development. Yes, hardware performance is constantly improving. Upgrades to SQL Server—especially to the optimizer, which helps determine how a query is executed, and the query engine, which executes the query—lead to better performance all on their own. Further, automation within SQL Server will do some aspects of query tuning for you. At the same time, SQL Server instances are being put on virtual machines, either locally or in hosted environments, where the hardware behavior is not guaranteed. Databases are going to platform-as-a-service systems such as Amazon RDS and Azure SQL Database. Object-relational mapping software such as Entity Framework will generate most queries for you. Despite all this, you still have to deal with fundamental database design and code generation. In short, query performance tuning remains a vital mechanism for improving the performance of your database management systems. The beauty of query performance tuning is that, in many cases, a small change to an index or a SQL query can result in a far more efficient application at a very low cost. In those cases, the increase in performance can be orders of magnitude better than that offered by an incrementally faster CPU or a slightly better optimizer.

There are, however, many pitfalls for the unwary. As a result, a proven process is required to ensure that you correctly identify and resolve performance bottlenecks. To whet your appetite for the types of topics essential to honing your query optimization skills, the following is a quick list of the query optimization aspects I cover in this book:

- Identifying problematic SQL queries
- Analyzing a query execution plan
- Evaluating the effectiveness of the current indexes

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_1

- Taking advantage of the Query Store to monitor and fix queries
- Evaluating the effectiveness of the current statistics
- Understanding parameter sniffing and fixing it when it breaks
- Optimizing execution plan caching
- Analyzing and minimizing statement recompilation
- Minimizing blocking and deadlocks
- Taking advantage of the storage mechanism Columnstore
- Applying in-memory table storage and procedure execution
- Applying performance-tuning processes, tools, and optimization techniques to optimize SQL workloads

Before jumping straight into these topics, let’s first examine why we go about performance tuning the way we do. In this chapter, I discuss the basic concepts of performance tuning for a SQL Server database system. It’s important to have a process you follow to be able to find and identify performance problems, fix those problems, and document the improvements you’ve made. Without a well-structured process, you’re going to be stabbing in the dark, hoping to hit a target. I detail the main performance bottlenecks and show just how important it is to design a database-friendly application, which is the consumer of the data, as well as how to optimize the database. Specifically, I cover the following topics:

- The performance tuning process
- Performance versus price
- The performance baseline
- Where to focus efforts in tuning
- The top 13 SQL Server performance killers

What I don’t cover within these pages could fill a number of other books. The focus of this book is on T-SQL query performance tuning, as the title says. But, just so you’re clear, there will be no coverage of the following:

- Hardware choices
- Application coding methodologies

- Server configuration (except where it impacts query tuning)
- SQL Server Integration Services
- SQL Server Analysis Services
- SQL Server Reporting Services
- PowerShell
- Virtual machines, whether in Azure or local
- Details of SQL Server on Linux (although a small amount of information is provided)

The Performance Tuning Process

The performance tuning process consists of identifying performance bottlenecks, prioritizing the identified issues, troubleshooting their causes, applying different resolutions, and quantifying performance improvements—and then repeating the whole process again and again. It is necessary to be a little creative since most of the time there is no one silver bullet to improve performance. The challenge is to narrow down the list of possible causes and evaluate the effects of different resolutions. You may even undo previous modifications as you iterate through the tuning process.

The Core Process

During the tuning process, you must examine various hardware and software factors that can affect the performance of a SQL Server–based application. You should be asking yourself the following general questions during the performance analysis:

- Is any other resource-intensive application running on the same server?
- Is the capacity of the hardware subsystem capable of withstanding the maximum workload?
- Is SQL Server configured properly?
- Does the SQL Server environment, whether physical server, VM, or platform, have adequate resources, or am I dealing with a configuration issue or even resource contention from other services?

- Is the network connection between SQL Server and the application adequate?
- Does the database design support the fastest data retrieval (and modification for an updatable database)?
- Is the user workload, consisting of SQL queries, optimized to reduce the load on SQL Server?
- What processes are causing the system to slow down as reflected in the measurement of various wait states, performance counters, and other measurement sources?
- Does the workload support the required level of concurrency?

If any of these factors is not configured properly, then the overall system performance may suffer. Let's briefly examine these factors.

Having another resource-intensive application on the same server can limit the resources available to SQL Server. Even an application running as a service can consume a good part of the system resources and limit the resources available to SQL Server. When SQL Server has to wait on resources from the other service, then your queries will also be waiting on those resources before you can retrieve or update your data.

Improperly configuring the hardware can prevent SQL Server from gaining the maximum benefit from the available resources. The main hardware resources to be considered are processor, memory, disk, and network. If the capacity of a particular hardware resource is small, then it can soon become a performance bottleneck for SQL Server. While I'm not covering hardware choices, as a part of tuning queries, you do need to understand how and where you may see performance bottlenecks because of the hardware you have. Chapters 2, 3, and 4 cover some of these hardware bottlenecks in detail.

You should also look at the configuration of SQL Server since proper configuration is essential for an optimized application. There is a long list of SQL Server configurations that defines the generic behavior of a SQL Server installation. These configurations can be viewed and modified using a system stored procedure, `sp_configure`, and viewed directly through a system view, `sys.configurations`. Many of these configurations can also be managed interactively through SQL Server Management Studio.

Since the SQL Server configurations are applicable for the complete SQL Server installation, a standard configuration is usually preferred. The good news is that, generally, you need not modify the majority of these configurations; the default settings work best for most situations. In fact, the general recommendation is to keep most SQL Server configurations at the default values. I discuss some of the configuration parameters in detail throughout this book and make some recommendations for changing a few of them.

The same thing applies to database options. The default settings on the model database are adequate for most systems. You should probably adjust autogrowth settings from the defaults, but many of the other properties, such as autoclose or autoshrink, should be left off, while others, such as the automatic creation of statistics, should be left on in most circumstances.

If you're running inside of some hosted environment, you might be sharing a server with a number of other virtual machines or databases. In some cases, you can work with the vendor or your local administrators to adjust the settings of these virtual environments to help your SQL Server instance perform better. But, in many circumstances, you'll have little to no control over the behavior of the systems at all. You'll need to work with the individual platform to determine when you're hitting limits on that platform that could also be causing performance issues.

Poor connectivity between SQL Server and the database application can hurt application performance. One of the questions you should ask yourself is, how good is the network connection? For example, the query executed by the application may be highly optimized, but the network connection used to submit this query may add considerable overhead to the overall performance. Ensuring that you have an optimal network configuration with appropriate bandwidth will be a fundamental part of your system setup. This is especially true if you're hosting your environments on the cloud.

The design of the database should also be analyzed while troubleshooting performance. This helps you understand not only the entity-relationship model of the database but also why a query may be written in a certain way. Although it may not always be possible to modify an in-use database design because of wider implications on the database application, a good understanding of the database design helps you focus in the right direction and understand the impact of a resolution. This is especially true of the primary and foreign keys and the clustered indexes used in the tables.

The application may be slow because of poorly built queries, the queries might not be able to use the indexes, or perhaps even the indexes themselves are inefficient or missing. If any of the queries are not optimized sufficiently, they can seriously impact other queries' performance. I cover index optimization in depth in Chapters 8, 9, 12, 13, and 14. The next question at this stage should be, is a query slow because it is resource intensive or because of concurrency issues with other queries? You can find in-depth information on blocking analysis in Chapter 21.

When processes run on a server, even one with multiple processors, at times one process will be waiting on another to complete. You can get a fundamental understanding of the root cause of slowdowns by identifying what is waiting and what is causing it to wait. You can realize this through operating system counters that you access through dynamic management views within SQL Server and through Performance Monitor. I cover this information in Chapters 2–4 and in Chapter 21.

The challenge is to find out which factor is causing the performance bottleneck. For example, with slow-running SQL queries and high pressure on the hardware resources, you may find that both poor database design and a nonoptimized query workload are to blame. In such a case, you must diagnose the symptoms further and correlate the findings with possible causes. Because performance tuning can be time-consuming and costly, you should ideally take a preventive approach by designing the system for optimum performance from the outset.

To strengthen the preventive approach, every lesson that you learn during the optimization of poor performance should be considered an optimization guideline when implementing new database applications. There are also proven best practices that you should consider while implementing database applications. I present these best practices in detail throughout the book, and Chapter 27 is dedicated to outlining many of the optimization best practices.

Please ensure that you take the performance optimization techniques into consideration at the early stages of your database application development. Doing so will help you roll out your database projects without big surprises later.

Unfortunately, we rarely live up to this ideal and often find database applications needing performance tuning. Therefore, it is important to understand not only how to improve the performance of a SQL Server–based application but also how to diagnose the causes of poor performance.

Iterating the Process

Performance tuning is an iterative process where you identify major bottlenecks, attempt to resolve them, measure the impact of your changes, and return to the first step until performance is acceptable. When applying your solutions, you should follow the golden rule of making only one change at a time where possible. Any change usually affects other parts of the system, so you must reevaluate the effect of each change on the performance of the overall system.

As an example, adding an index may fix the performance of a specific query, but it could cause other queries to run more slowly, as explained in Chapters 8 and 9. Consequently, it is preferable to conduct a performance analysis in a test environment to shield users from your diagnosis attempts and intermediate optimization steps. In such a case, evaluating one change at a time also helps in prioritizing the implementation order of the changes on the production server based on their relative contributions. Chapter 26 explains how to automate testing your database and query performance to help with this process.

You can keep on chipping away at the performance bottlenecks you've determined are the most painful and thus improve the system performance gradually. Initially, you will be able to resolve big performance bottlenecks and achieve significant performance improvements, but as you proceed through the iterations, your returns will gradually diminish. Therefore, to use your time efficiently, it is worthwhile to quantify the performance objectives first (for example, an 80 percent reduction in the time taken for a certain query, with no adverse effect anywhere else on the server) and then work toward them.

The performance of a SQL Server application is highly dependent on the amount and distribution of user activity (or workload) and data. Both the amount and distribution of workload and data usually change over time, and differing data can cause SQL Server to execute SQL queries differently. The performance resolution applicable for a certain workload and data may lose its effectiveness over a period of time. Therefore, to ensure optimum system performance on a continuing basis, you need to analyze system and application performance at regular intervals. Performance tuning is a never-ending process, as shown in Figure 1-1.

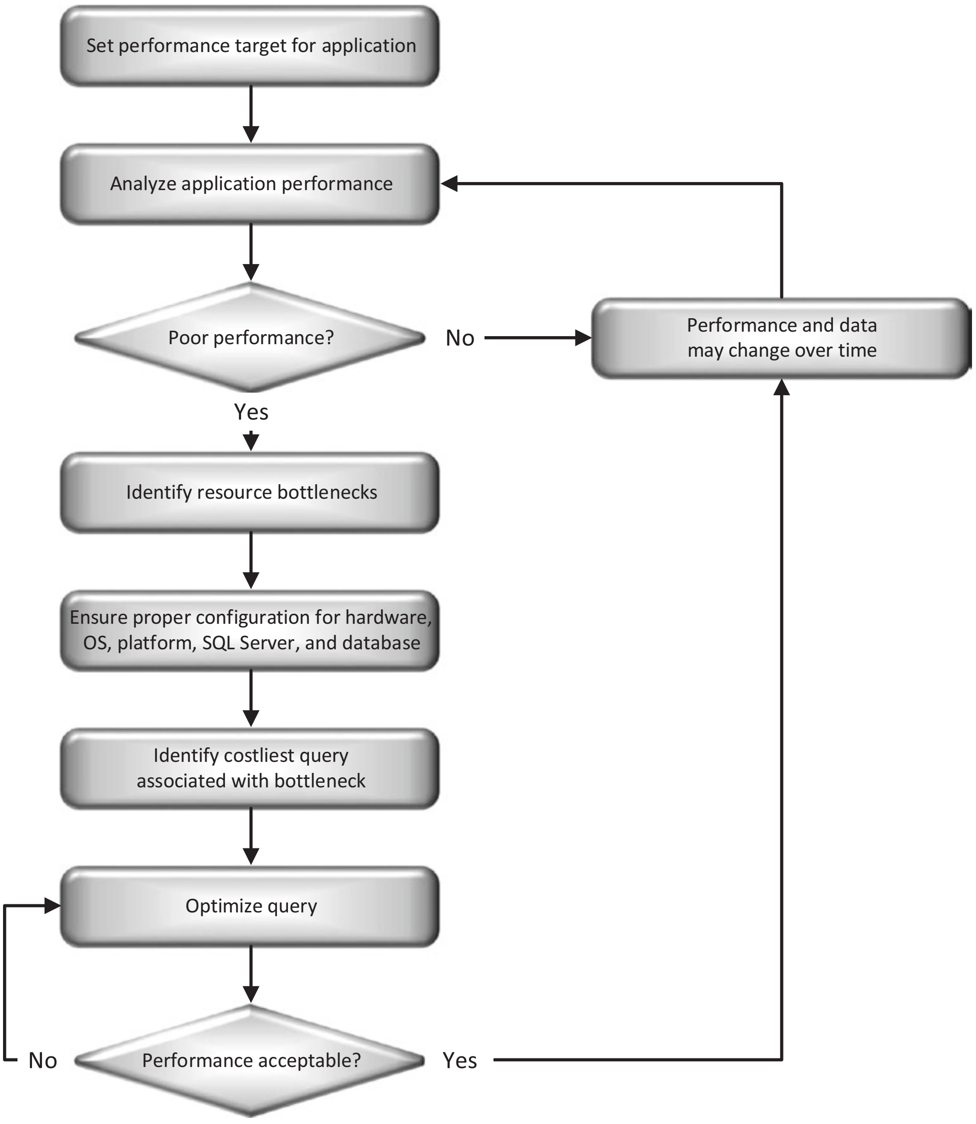

Set performance target for application
Analyze application performance
Poor performance?
No
Performance and data may change over time
Yes
Identify resource bottlenecks
Ensure proper configuration for hardware, OS, platform, SQL Server, and database
Identify costliest query associated with bottleneck
Optimize query
No
Performance acceptable?
Yes

Figure 1-1. *Performance tuning process*

You can see that the steps to optimize the costliest query make for a complex process, which also requires multiple iterations to troubleshoot the performance issues within the query and apply one change at a time. Figure 1-2 shows the steps involved in the optimization of the costliest query.

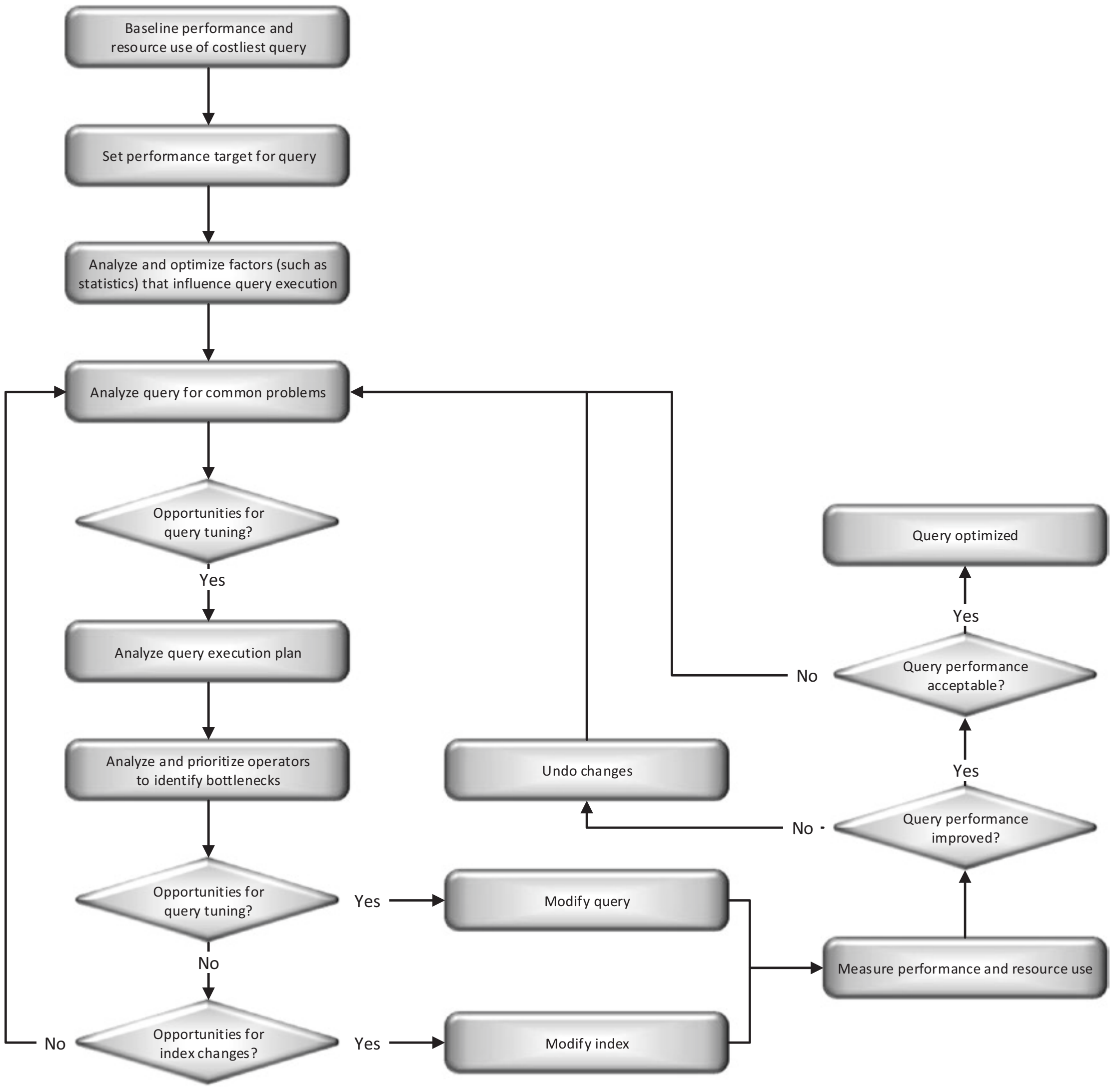

Baseline performance and resource use of costliest query
Set performance target for query
Analyze and optimize factors (such as statistics) that influence query execution
Analyze query for common problems
Opportunities for query tuning?
Yes
Analyze query execution plan
Analyze and prioritize operators to identify bottlenecks
Opportunities for query tuning?
Yes
No
Opportunities for index changes?
No
Yes
Modify query
Modify index
Undo changes
Measure performance and resource use
Query performance improved?
No
Yes
Query performance acceptable?
No
Yes
Query optimized

Figure 1-2. *Optimization of the costliest query*

As you can see from this process, there is quite a lot to do to ensure that you correctly tune the performance of a given query. It is important to use a solid process like this in performance tuning to focus on the main identified issues.

Having said this, it also helps to keep a broader perspective about the problem as a whole since you may believe one aspect is causing the performance bottleneck when in reality something else is causing the problem. At times you may have to go back to the business to identify potential changes in the requirements to find a way to make things run faster.

Performance vs. Price

One of the points I touched on earlier is that to gain increasingly small performance increments, you need to spend increasingly large amounts of time and money. Therefore, to ensure the best return on your investment, you should be objective while optimizing performance. Always consider the following two aspects:

- What is the acceptable performance for your application?
- Is the investment worth the performance gain?

Performance Targets

To derive maximum efficiency, you must realistically estimate your performance requirements. You can follow many best practices to improve performance. For example, you can have your database files on the most high-performance disk subsystem. However, before applying a best practice, you should consider how much you may gain from it and whether the gain will be worth the investment. Those performance requirements are usually set by someone else, either the application developers or the business consumers of the data. A fundamental part of query tuning will involve talking to these parties to determine a good enough and realistic set of requirements.

Sometimes it is really difficult to estimate the performance gain without actually making the enhancement. That makes properly identifying the source of your performance bottlenecks even more important. Are you CPU, memory, or disk bound? Is the cause code, data structure, or indexing, or are you simply at the limit of your hardware? Do you have a bad router, a poorly configured I/O path, or an improperly applied patch causing the network to perform slowly? Is your service tier on your platform set to the appropriate level? Be sure you can make these possibly costly decisions from a known point rather than guessing. One practical approach is to increase a resource in increments and analyze the application's scalability with the added resource. A scalable application will proportionately benefit from an incremental increase of the resource, if the resource was truly causing the scalability bottleneck. If the results appear to be satisfactory, then you can commit to the full enhancement. Experience also plays an important role here.

However, sometimes you're in pain from a performance perspective, and you need to do whatever you can to alleviate that pain. It's possible that a full root-cause analysis just won't always be possible. It's still the preferred path to provide the most protection for your production systems, but it's acknowledged that you won't always be able to do it.

"Good Enough" Tuning

Instead of tuning a system to the theoretical maximum performance, the goal should be to tune until the system performance is "good enough." This is a commonly adopted performance tuning approach. The cost investment after such a point usually increases exponentially in comparison to the performance gain. The 80:20 rule works very well: by investing 20 percent of your resources, you may get 80 percent of the possible performance enhancement, but for the remaining 20 percent possible performance gain, you may have to invest an additional 80 percent of resources. It is therefore important to be realistic when setting your performance objectives. Just remember that "good enough" is defined by you, your customers, and the businesspeople you're working with. There is no standard to which everyone adheres.

A business benefits not by considering pure performance but by considering the price of performance. However, if the target is to find the scalability limit of your application (for various reasons, including marketing the product against its competitors), then it may be worthwhile to invest as much as you can. Even in such cases, using a third-party stress test lab may be a better investment decision.

While there is a need in some cases to drill down to find every possible microsecond of performance enhancement, for most of us, most of the time, it's just not necessary. Instead, focusing on ensuring that we're doing the standard best practices appropriately will get us where we need to be. You may find yourself in an exceptional situation, but generally, this won't be the case. Focus first on the right standards.

Performance Baseline

One of the main objectives of performance analysis is to understand the underlying level of system use or pressure on different hardware and software subsystems. This knowledge helps you in the following ways:

- Allows you to analyze resource bottlenecks.
- Enables you to troubleshoot by comparing system utilization patterns with a preestablished baseline.
- Assists you in making accurate estimates in capacity planning and scheduling hardware upgrades.
- Aids you in identifying low-utilization periods when the database administrative activities can best be executed.

- Helps you estimate the nature of possible hardware downsizing or server consolidation. Why would a company downsize? Well, the company may have leased a very high-end system expecting strong growth, but because of poor growth, they now want to downsize their systems. And consolidation? Companies sometimes buy too many servers or realize that the maintenance and licensing costs are too high. This would make using fewer servers very attractive.
- Some metrics make sense only when compared to previously recorded values. Without that previous measure you won't be able to make sense of the information.

Therefore, to better understand your application's resource requirements, you should create a baseline for your application's hardware and software usage. A *baseline* serves as a statistic of your system's current usage pattern and as a reference with which to compare future statistics. Baseline analysis helps you understand your application's behavior during a stable period, how hardware resources are used during such periods, and the characteristics of the software. With a baseline in place, you can do the following:

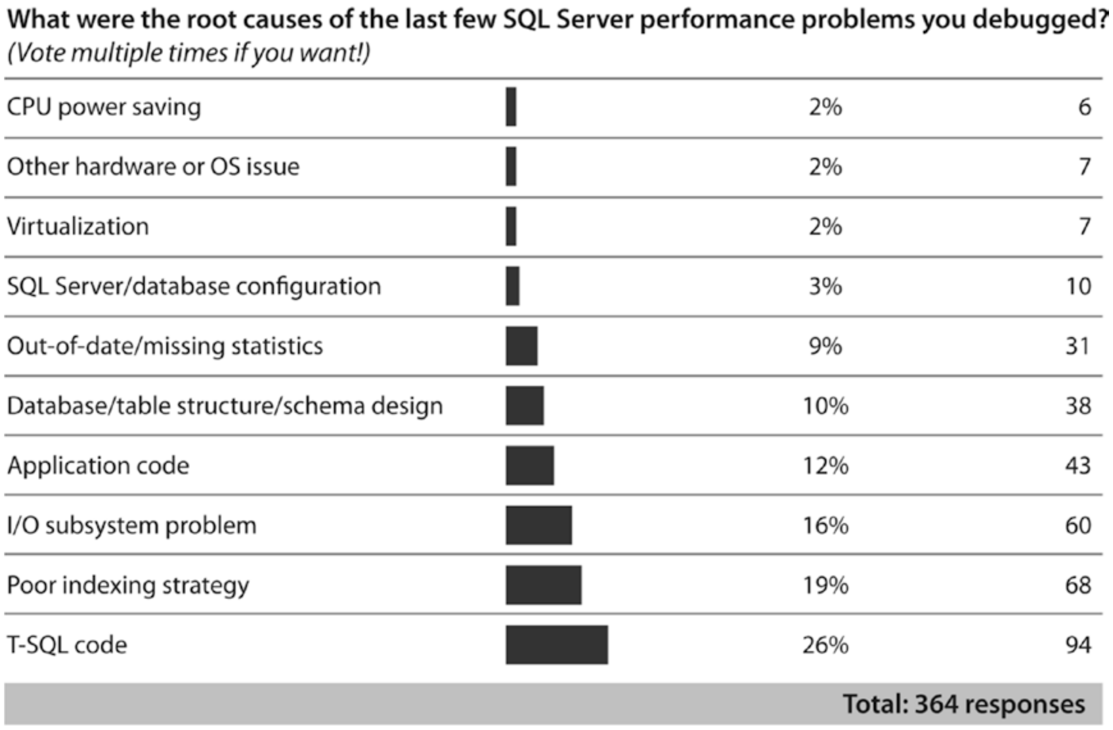
- Measure current performance and express your application's performance goals.
- Compare other hardware and software combinations, or compare platform service tiers against the baseline.
- Measure how the workload and/or data changes over time. This includes know about business cycles such as annual renewals or a sales event.
- Ensure that you understand what "normal" is on your server so that an arbitrary number isn't misinterpreted as an issue.
- Evaluate the peak and nonpeak usage pattern of the application. This information can be used to effectively distribute database administration activities, such as full database backup and database defragmentation during nonpeak hours.

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

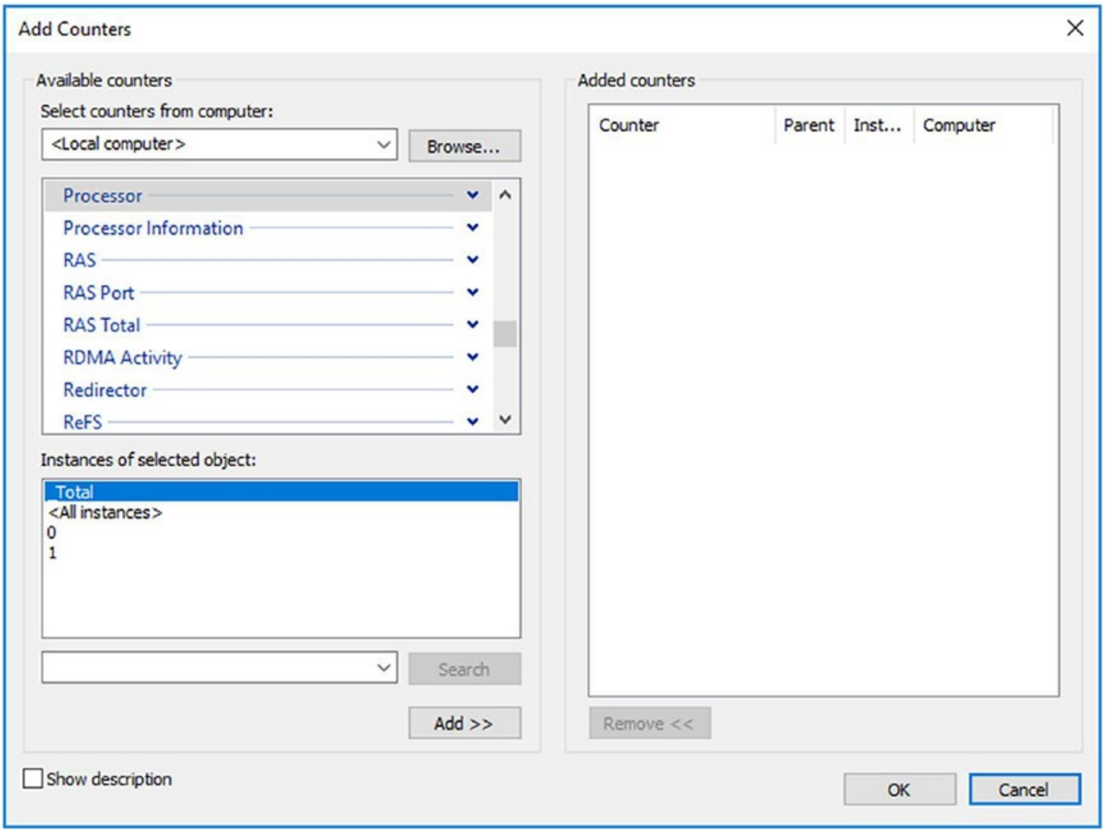


Figure 2-1. *Adding a Performance Monitor counter*

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:

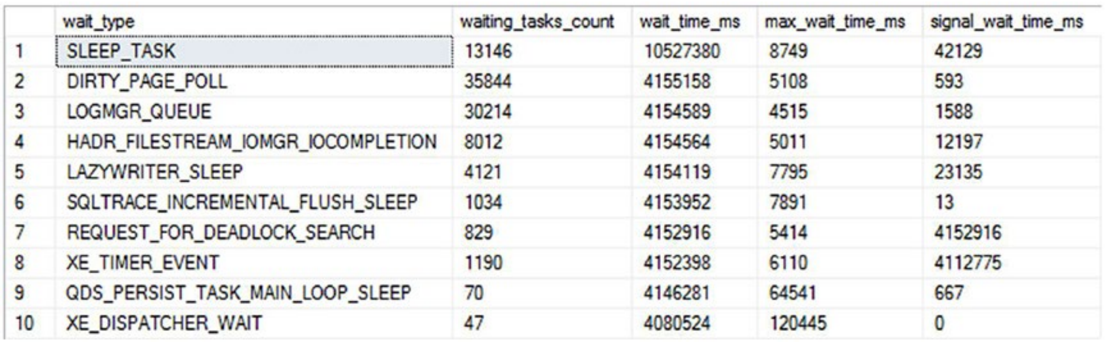
```
SELECT  dopc.cntr_value,
        dopc.cntr_type
FROM    sys.dm_os_performance_counters AS dopc
WHERE   dopc.object_name =  'SQLServer:General Statistics'
        AND dopc.counter_name =  'Logins/sec';
```

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.

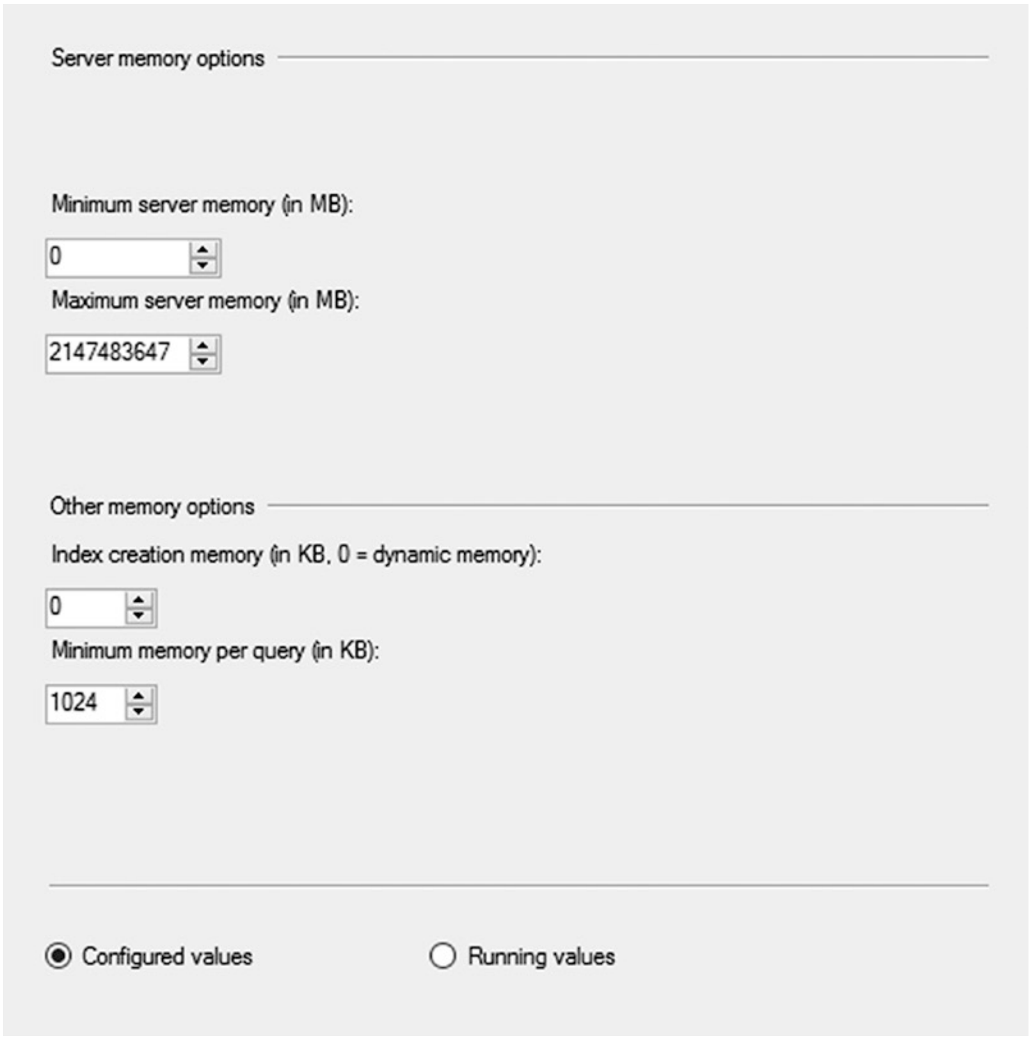
Memory Bottleneck Analysis

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.


Server memory options

Minimum server memory (in MB):

0

Maximum server memory (in MB):

2147483647

Other memory options

Index creation memory (in KB, 0 = dynamic memory):

0

Minimum memory per query (in KB):

1024

[✓ Configured values] Running values

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.

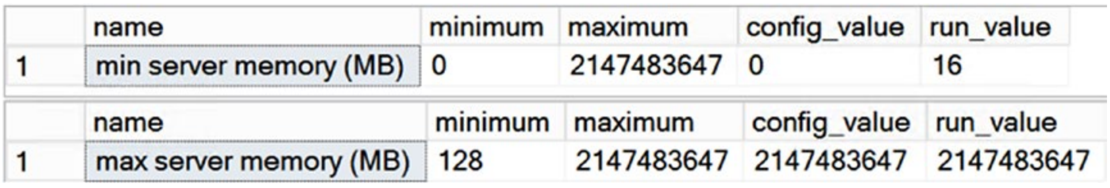
In Linux, the general guidance is to leave about 20 percent of memory on the system for the operating system. The same types of processing needs are going to apply as the OS needs memory to manage its various resources in support of SQL Server.

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.

You can also manage the configuration values for min server memory and max server memory by using the sp_configure system stored procedure. To see the configuration values for these parameters, execute the sp_configure stored procedure as follows:

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
EXEC sp_configure  'min server memory';
EXEC sp_configure  'max server memory';
```

Figure 2-4 shows the result of running these commands.



	name	minimum	maximum	config_value	run_value
1	min server memory (MB)	0	2147483647	0	16

	name	minimum	maximum	config_value	run_value
1	max server memory (MB)	128	2147483647	2147483647	2147483647

Figure 2-4. *SQL Server memory configuration properties*

Note that the default value for the min server memory setting is 0MB and for the max server memory setting is 2147483647MB.

You can also modify these configuration values using the sp_configure stored procedure. For example, to set max server memory to 10GB and min server memory to 5GB, execute the following set of statements (setmemory.sql in the download):

```
USE master;
EXEC sp_configure  'show advanced option',   1;
RECONFIGURE;
exec sp_configure  'min server memory (MB)',  5120;
exec sp_configure  'max server memory (MB)',  10240;
RECONFIGURE WITH OVERRIDE;
```

The min server memory and max server memory configurations are classified as advanced options. By default, the sp_configure stored procedure does not affect/display the advanced options. Setting show advanced option to 1 as shown previously enables the sp_configure stored procedure to affect/display the advanced options.

The RECONFIGURE statement updates the memory configuration values set by `sp_configure`. Since ad hoc updates to the system catalog containing the memory configuration values are not recommended, the OVERRIDE flag is used with the RECONFIGURE statement to force the memory configuration. If you do the memory configuration through Management Studio, Management Studio automatically executes the RECONFIGURE WITH OVERRIDE statement after the configuration setting.

Another way to see the settings but not to manipulate them is to use the `sys.configurations` system view. You can select from `sys.configurations` using standard T-SQL rather than having to execute a command.

You may need to allow for SQL Server sharing a system's memory. To elaborate, consider a computer with SQL Server and SharePoint running on it. Both servers are heavy users of memory and thus keep pushing each other for memory. The dynamic memory behavior of SQL Server allows it to release memory to SharePoint at one instance and grab it back as SharePoint releases it. You can avoid this dynamic memory management overhead by configuring SQL Server for a fixed memory size. However, please keep in mind that since SQL Server is an extremely resource-intensive process, it is highly recommended that you have a dedicated SQL Server production machine.

Now that you understand SQL Server memory management at a very high level, let's consider the performance counters you can use to analyze stress on memory, as shown in Table 2-1.

Table 2-1. *Performance Monitor Counters to Analyze Memory Pressure*

Object(Instance [,InstanceN])	Counter	Description	Values
Memory	Available Bytes	Free physical memory	System dependent
	Pages/sec	Rate of hard page faults	Compare with baseline
	Page Faults/sec	Rate of total page faults	Compare with its baseline value for trend analysis
	Pages Input/sec	Rate of input page faults	
	Pages Output/sec	Rate of output page faults	
	Paging File %Usage Peak	Peak values in the memory paging file	
	Paging File: %Usage	Rate of usage of the memory paging file	
SQLServer: Buffer Manager	Buffer cache hit ratio	Percentage of requests served out of buffer cache	Compare with its baseline value for trend analysis
	Page Life Expectancy	Time page spends in buffer cache	Compare with its baseline value for trend analysis
	Checkpoint Pages/sec	Pages written to disk by checkpoint	Compare with baseline
	Lazy writes/sec	Dirty aged pages flushed from buffer	Compare with baseline
SQLServer: Memory Manager	Memory Grants Pending	Number of processes waiting for memory grant	Average value = 0
	Target Server Memory (KB)	Maximum physical memory SQL Server can have on the box	Close to size of physical memory
	Total Server Memory (KB)	Physical memory currently assigned to SQL	Close to target server memory (KB)
Process	Private Bytes	Size, in bytes, of memory that this process has allocated that can't be shared with other processes	

Memory and disk I/O are closely related. Even if you think you have a problem that is directly memory related, you should also gather I/O metrics to understand how the system is behaving between the two resources. I'll now walk you through these counters to give you a better idea of possible uses.

Available Bytes

The Available Bytes counter represents free physical memory in the system. You can also look at Available Kbytes and Available Mbytes for the same data but with less granularity. For good performance, this counter value should not be too low. If SQL Server is configured for dynamic memory usage, then this value will be controlled by calls to a Windows API that determines when and how much memory to release. Extended periods of time with this value very low and SQL Server memory not changing indicates that the server is under severe memory stress.

Pages/Sec and Page Faults/Sec

To understand the importance of the Pages/sec and Page Faults/sec counters, you first need to learn about page faults. A *page fault* occurs when a process requires code or data that is not in its *working set* (its space in physical memory). It may lead to a soft page fault or a hard page fault. If the faulted page is found elsewhere in physical memory, then it is called a *soft page fault*. A *hard page fault* occurs when a process requires code or data that is not in its working set or elsewhere in physical memory and must be retrieved from disk.

The speed of a disk access is in the order of milliseconds for mechanical drives or as low as .1 milliseconds for a solid-state drive (SSD), whereas a memory access is in the order of nanoseconds. This huge difference in the speed between a disk access and a memory access makes the effect of hard page faults significant compared to that of soft page faults.

The Pages/sec counter represents the number of pages read from or written to disk per second to resolve hard page faults. The Page Faults/sec performance counter indicates the total page faults per second—soft page faults plus hard page faults—handled by the system. These are primarily measures of load and are not direct indicators of performance issues.

Hard page faults, indicated by Pages/sec, should not be consistently higher than normal. There are no hard-and-fast numbers for what indicates a problem because these numbers will vary widely between systems based on the amount and type of memory as well as the speed of disk access on the system.

If the Pages/sec counter is high, you can break it up into Pages Input/sec and Pages Output/sec.

- *Pages Input/sec*: An application will wait only on an input page, not on an output page.
- *Pages Output/sec*: Page output will stress the system, but an application usually does not see this stress. Pages output are usually represented by the application's dirty pages that need to be backed out to the disk. Pages Output/sec is an issue only when disk load become an issue.

Also, check Process:Page Faults/sec to find out which process is causing excessive paging in case of high Pages/sec. The Process object is the system component that provides performance data for the processes running on the system, which are individually represented by their corresponding instance name.

For example, the SQL Server process is represented by the sqlservr instance of the Process object. High numbers for this counter usually do not mean much unless Pages/sec is high. Page Faults/sec can range all over the spectrum with normal application behavior, with values from 0 to 1,000 per second being acceptable. This entire data set means a baseline is essential to determine the expected normal behavior.

Paging File %Usage and Page File %Usage

All memory in the Windows system is not the physical memory of the physical machine. Windows will swap memory that isn't immediately active in and out of the physical memory space to a paging file. These counters are used to understand how often this is occurring on your system. As a general measure of system performance, these counters are applicable only to the Windows OS and not to SQL Server. However, the impact of not enough virtual memory will affect SQL Server. These counters are collected to understand whether the memory pressures on SQL Server are internal or external. If they are external memory pressures, you will need to go into the Windows OS to determine what the problems might be.

Buffer Cache Hit Ratio

The *buffer cache* is the pool of buffer pages into which data pages are read, and it is often the biggest part of the SQL Server memory pool. This counter value should be as high as possible, especially for OLTP systems that should have fairly regimented data access, unlike a warehouse or reporting system. It is extremely common to find this counter value as 99 percent or more for most production servers. A low Buffer cache hit ratio value indicates that few requests could be served out of the buffer cache, with the rest of the requests being served from disk.

When this happens, either SQL Server is still warming up or the memory requirement of the buffer cache is more than the maximum memory available for its growth. If the cache hit ratio is consistently low, you might consider getting more memory for the system or reducing memory requirements through the use of good indexes and other query tuning mechanisms, that is, unless you're dealing with reporting systems with lots of ad hoc queries. It's possible when working with reporting systems to consistently see the cache hit ratio become extremely low.

This makes the buffer cache hit ratio an interesting number for understanding aspects of system behavior, but it is not a value that would suggest, by itself, potential performance problems. While this number represents an interesting behavior within the system, it's not a great measure for precise problems but instead shows a type of behavior. For more details on this topic, please read the "Great SQL Server Debates: Buffer Cache Hit Ratio" article on Simple-Talk (https://bit.ly/2rzWJvO).

Page Life Expectancy

Page Life Expectancy indicates how long a page will stay in the buffer pool without being referenced. Generally, a low number for this counter means that pages are being removed from the buffer, lowering the efficiency of the cache and indicating the possibility of memory pressure. On reporting systems, as opposed to OLTP systems, this number may remain at a lower value since more data is accessed from reporting systems. It's also common to see Page Life Expectancy fall to very low levels during nightly loads. Since this is dependent on the amount of memory you have available and the types of queries running on your system, there are no hard-and-fast numbers that will satisfy a wide audience. Therefore, you will need to establish a baseline for your system and monitor it over time.

If you are on a machine with nonuniform memory access (NUMA) , you need to know that the standard Page Life Expectancy counter is an average. To see specific measures, you'll need to use the Buffer Node:Page Life Expectancy counter.

Checkpoint Pages/Sec

The Checkpoint Pages/sec counter represents the number of pages that are moved to disk by a checkpoint operation. These numbers should be relatively low, for example, less than 30 per second for most systems. A higher number means more pages are being marked as dirty in the cache. A *dirty page* is one that is modified while in the buffer. When it's modified, it's marked as dirty and will get written back to the disk during the next checkpoint. Higher values on this counter indicate a larger number of writes occurring within the system, possibly indicative of I/O problems. But, if you are taking advantage of indirect checkpoints, which allow you to control when checkpoints occur in order to reduce recovery intervals, you might see different numbers here. Take that into account when monitoring databases with the indirect checkpoint configured. For more information about checkpoints on SQL Server 2016 and better, I suggest you read the "Changes in SQL Server 2016 Checkpoint Behavior" article on MSDN (`https://bit.ly/2pdggk3`).

Lazy Writes/Sec

The Lazy writes/sec counter records the number of buffers written each second by the buffer manager's lazy write process. This process is where the dirty, aged buffers are removed from the buffer by a system process that frees up the memory for other uses. A dirty, aged buffer is one that has changes and needs to be written to the disk. Higher values on this counter possibly indicate I/O issues or even memory problems. The Lazy writes/sec values should consistently be less than 20 for the average system. However, as with all other counters, you must compare your values to a baseline measure.

Memory Grants Pending

The Memory Grants Pending counter represents the number of processes pending for a memory grant within SQL Server memory. If this counter value is high, then SQL Server is short of buffer memory, which can be caused not simply by a lack of memory but by issues such as oversized memory grants caused by incorrect row counts because your statistics are out-of-date. Under normal conditions, this counter value should consistently be 0 for most production servers.

Another way to retrieve this value, on the fly, is to run queries against the DMV `sys.dm_exec_query_memory_grants`. A `null` value in the column `grant_time` indicates that the process is still waiting for a memory grant. This is one method you can use to troubleshoot query timeouts by identifying that a query (or queries) is waiting on memory in order to execute.

Target Server Memory (KB) and Total Server Memory (KB)

Target Server Memory (KB) indicates the total amount of dynamic memory SQL Server is willing to consume. Total Server Memory (KB) indicates the amount of memory currently assigned to SQL Server. The Total Server Memory (KB) counter value can be very high if the system is dedicated to SQL Server. If Total Server Memory (KB) is much less than Target Server Memory (KB), then either the SQL Server memory requirement is low, the max server memory configuration parameter of SQL Server is set at too low a value, or the system is in warm-up phase. The *warm-up phase* is the period after SQL Server is started when the database server is in the process of expanding its memory allocation dynamically as more data sets are accessed, bringing more data pages into memory.

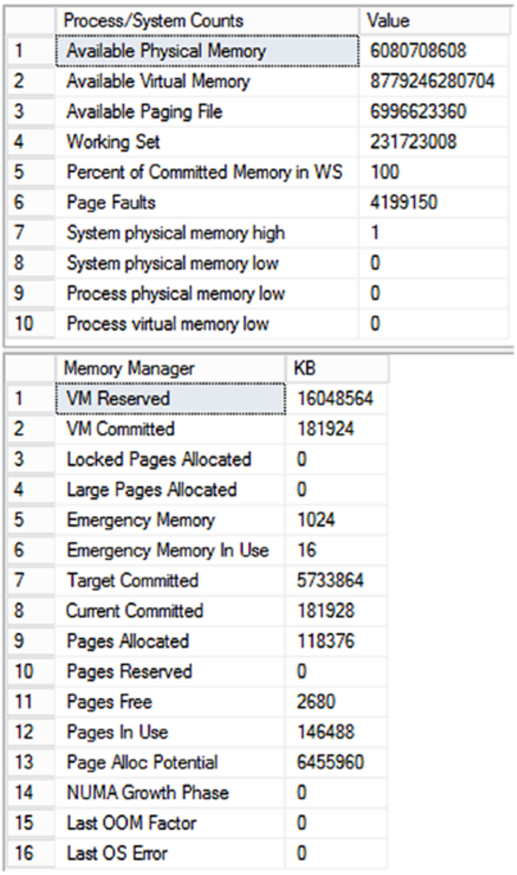
You can confirm a low memory requirement from SQL Server by the presence of a large number of free pages, usually 5,000 or more. Also, you can directly check the status of memory by querying the DMV `sys.dm_os_ring_buffers`, which returns information about memory allocation within SQL Server. I cover `sys.dm_os_ring_buffers` in more detail in the following section.

Additional Memory Monitoring Tools

While you can get the basis for the behavior of memory within SQL Server from the Performance Monitor counters, once you know that you need to spend time looking at your memory usage, you'll need to take advantage of other tools and tool sets. The following are some of the commonly used reference points for identifying memory issues on a SQL Server system. A few of these tools are only of use for in-memory OLTP management. Some of these tools, while actively used by large numbers of the SQL Server community, are not documented within SQL Server Books Online. This means they are absolutely subject to change or removal.

DBCC MEMORYSTATUS

This command goes into the SQL Server memory and reads out the current allocations. It's a moment-in-time measurement, a snapshot. It gives you a set of measures of where memory is currently allocated. The results from running the command come back as two basic result sets, as you can see in Figure 2-5.



	Process/System Counts	Value
1	Available Physical Memory	6080708608
2	Available Virtual Memory	8779246280704
3	Available Paging File	6996623360
4	Working Set	231723008
5	Percent of Committed Memory in WS	100
6	Page Faults	4199150
7	System physical memory high	1
8	System physical memory low	0
9	Process physical memory low	0
10	Process virtual memory low	0

	Memory Manager	KB
1	VM Reserved	16048564
2	VM Committed	181924
3	Locked Pages Allocated	0
4	Large Pages Allocated	0
5	Emergency Memory	1024
6	Emergency Memory In Use	16
7	Target Committed	5733864
8	Current Committed	181928
9	Pages Allocated	118376
10	Pages Reserved	0
11	Pages Free	2680
12	Pages In Use	146488
13	Page Alloc Potential	6455960
14	NUMA Growth Phase	0
15	Last OOM Factor	0
16	Last OS Error	0

Figure 2-5. *Output of DBCC MEMORYSTATUS*

The first data set shows basic allocations of memory and counts of occurrences. For example, Available Physical Memory is a measure of the memory available on the system, whereas Page Faults is just a count of the number of page faults that have occurred.

The second data set shows different memory managers within SQL Server and the amount of memory they have consumed at the moment that the `MEMORYSTATUS` command was called.

Each of these can be used to understand where memory allocation is occurring within the system. For example, in most systems, most of the time the primary consumer of memory is the buffer pool. You can compare the Target Committed value to the Current Committed value to understand if you're seeing pressure on the buffer pool. When Target Committed is higher than Current Committed, you might be seeing buffer cache problems and need to figure out which process within your currently executing SQL Server processes is using the most memory. This can be done using a dynamic management object, `sys.dm_os_performance_counters`.

The remaining data sets are various memory managers, memory clerks, and other memory stores from the full dump of memory that DBCC MEMORYSTATUS produces. They're only going to be interesting in narrow circumstances when dealing with particular aspects of SQL Server management, and they fall far outside the scope of this book to document them all. You can read more in the MSDN article "How to use the DBCC MEMORYSTATUS command" (`http://bit.ly/1eJ2M2f`).

Dynamic Management Views

There are a large number of memory-related DMVs within SQL Server. Several of them have been updated with SQL Server 2017, and some new ones have been added. Reviewing all of them is outside the scope of this book. There are three that are the most frequently used when determining whether you have memory bottlenecks within SQL Server. There are also another two that are useful when you need to monitor your in-memory OLTP memory usage.

Sys.dm_os_memory_brokers

While most of the memory within SQL Server is allocated to the buffer cache, there are a number of processes within SQL Server that also can, and will, consume memory. These processes expose their memory allocations through this DMV. You can use this to see what processes might be taking resources away from the buffer cache in the event you have other indications of a memory bottleneck.

Sys.dm_os_memory_clerks

A memory clerk is the process that allocates memory within SQL Server. Looking at what these processes are up to allows you to understand whether there are internal memory allocation issues going on within SQL Server that might rob the procedure cache of needed memory. If the Performance Monitor counter for Private Bytes is high, you can determine which parts of the system are being consumed through the DMV.

If you have a database using in-memory OLTP storage, you can use `sys.dm_db_xtp_table_memory_stats` to look at the individual database objects. But if you want to look at the allocations of these objects across the entire instance, you'll need to use `sys.dm_os_memory_clerks`.

Sys.dm_os_ring_buffers

This DMV is not documented within Books Online, so it is subject to change or removal. It changed between SQL Server 2008R2 and SQL Server 2012. The queries I normally run against it still seem to work for SQL Server 2017, but you can't count on that. This DMV outputs as XML. You can usually read the output by eye, but you may need to implement XQuery to get really sophisticated reads from the ring buffers.

A ring buffer is nothing more than a recorded response to a notification. Ring buffers are kept within this DMV, and accessing `sys.dm_os_ring_buffers` allows you to see things changing within your memory. Table 2-2 describes the main ring buffers associated with memory.

***Table 2-2.** Main Ring Buffers Associated with Memory*

Ring Buffer	Ring_buffer_type	Use
Resource Monitor	RING_BUFFER_ RESOURCE_ MONITOR	As memory allocation changes, notifications of this change are recorded here. This information can be useful for identifying external memory pressure.
Out Of Memory	RING_BUFFER_ OOM	When you get out-of-memory issues, they are recorded here so you can tell what kind of memory action failed.
Memory Broker	RING_BUFFER_ MEMORY_BROKER	As the memory internal to SQL Server drops, a low memory notification will force processes to release memory for the buffer. These notifications are recorded here, making this a useful measure for when internal memory pressure occurs.
Buffer Pool	RING_BUFFER_ BUFFER_POOL	Notifications of when the buffer pool itself is running out of memory are recorded here. This is just a general indication of memory pressure.

There are other ring buffers available, but they are not applicable to memory allocation issues.

Sys.dm_db_xtp_table_memory_stats

To see the memory in use by the tables and indexes that you created in-memory, you can query this DMV. The output measures the memory allocated and memory used for the tables and indexes. It outputs only the `object_id`, so you'll need to also query the system view `sys.objects` to get the names of tables or indexes. This DMV outputs for the database you are currently connected to when querying.

Sys.dm_xtp_system_memory_consumers

This DMV shows system structures that are used to manage the internals of the in-memory engine. It's not something you should normally have to deal with, but when troubleshooting memory issues, it's good to understand if you're dealing directly with something occurring within the system or just the amount of data that you've loaded into memory. The principal measures you'd be looking for here are the allocated and used bytes shown for each of the management structures.

Monitoring Memory in Linux

You won't have Perfmon within the Linux operating system. However, this doesn't mean you can't observe memory behavior on the server to understand how the system is behaving. You can query the DMVs `sys.dm_os_performance_counters` and `sys.dm_os_wait_stats` within a SQL Server 2017 instance running on Linux to observe memory behavior in that way.

Additional monitoring of the Linux OS can be done through native Linux tools. There are a large number of them, but a commonly used one is Grafana. It's open source with lots of documentation available online. The SQL Server Customer Advisory Team has a documented method for monitoring Linux that I can recommend: `http://bit.ly/2wi73bA`.

Memory Bottleneck Resolutions

When there is high stress on memory, indicated by a large number of hard page faults, you can resolve a memory bottleneck using the flowchart shown in Figure 2-6.

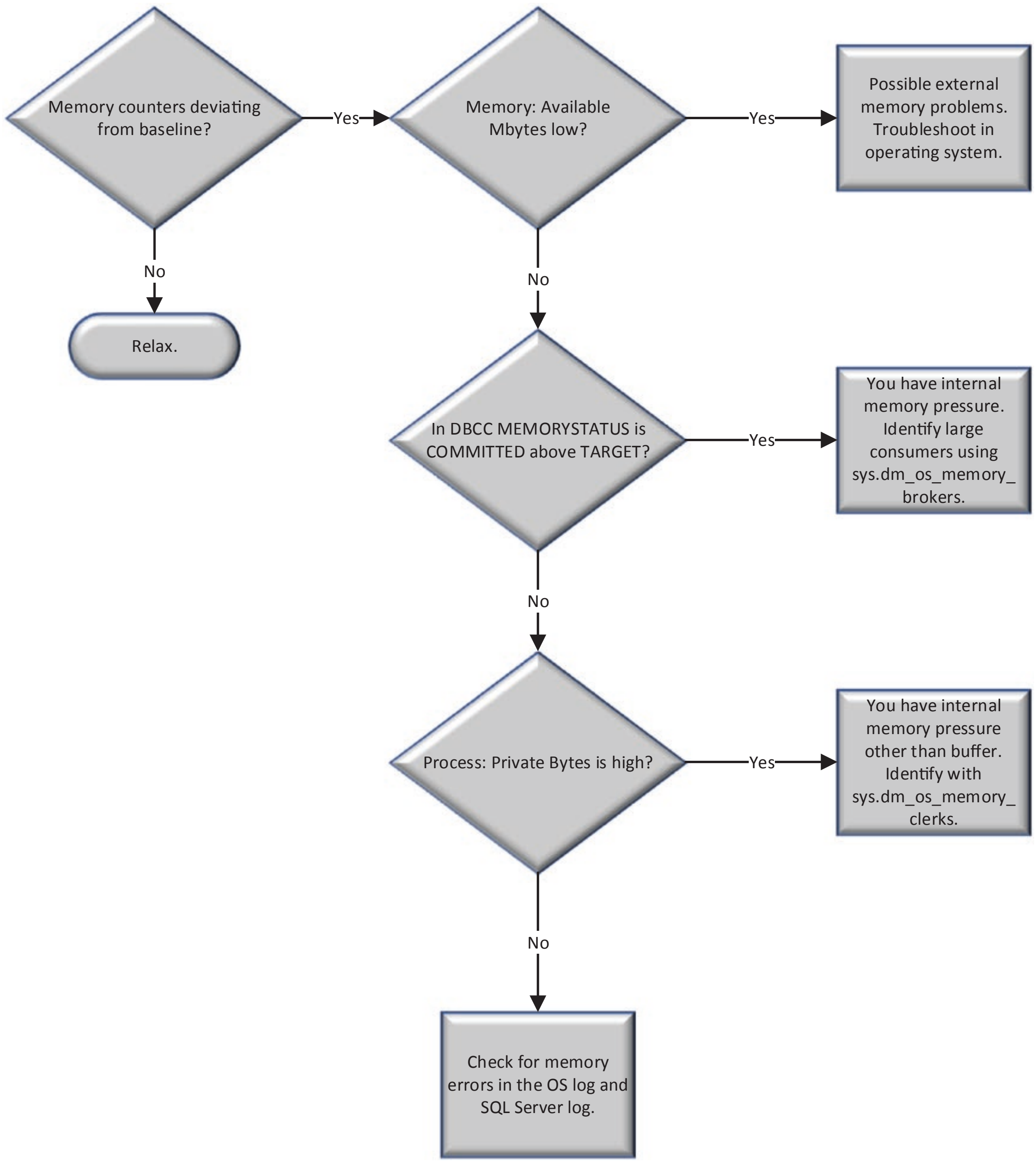

Memory counters deviating from baseline?
Yes
No
Relax.
Memory: Available Mbytes low?
Yes
Possible external memory problems. Troubleshoot in operating system.
No
In DBCC MEMORYSTATUS is COMMITTED above TARGET?
Yes
You have internal memory pressure. Identify large consumers using sys.dm_os_memory_brokers.
No
Process: Private Bytes is high?
Yes
You have internal memory pressure other than buffer. Identify with sys.dm_os_memory_clerks.
No
Check for memory errors in the OS log and SQL Server log.

***Figure 2-6.** Memory bottleneck resolution chart*

A few of the common resolutions for memory bottlenecks are as follows:

- Optimizing application workload
- Allocating more memory to SQL Server

- Moving in-memory tables back to standard storage
- Increasing system memory
- Changing from a 32-bit to a 64-bit processor
- Enabling 3GB of process space
- Compressing data
- Addressing fragmentation

Of course, fixing any of the query issues that can lead to excessive memory use is always an option. Let's take a look at each of these in turn.

Optimizing Application Workload

Optimizing application workload is the most effective resolution most of the time, but because of the complexity and challenges involved in this process, it is usually considered last. To identify the memory-intensive queries, capture all the SQL queries using Extended Events (which you will learn how to use in Chapter 6) or use Query Store (which we'll cover in Chapter 11) and then group the output on the `Reads` column. The queries with the highest number of logical reads contribute most often to memory stress, but there is not a linear correlation between the two. You can also use `sys.dm_exec_query_stats`, a DMV that collects query metrics for queries that are actively in cache to identify the same thing. But, since this DMV is based on cache, it may not be as accurate as capturing metrics using Extended Events, although it will be quicker and easier. You will see how to optimize those queries in more detail throughout this book.

Allocating More Memory to SQL Server

As you learned in the "SQL Server Memory Management" section, the max server memory configuration can limit the maximum size of the SQL Server buffer memory pool. If the memory requirement of SQL Server is more than the max server memory value, which you can tell through the number of hard page faults, then increasing the value will allow the memory pool to grow. To benefit from increasing the max server memory value, ensure that enough physical memory is available in the system.

If you are using in-memory OLTP storage, you may need to adjust the memory percentages allocated to the resource pools you have defined for your in-memory objects. But, that will take memory from other parts of your SQL Server instance.

Moving In-Memory Tables Back to Standard Storage

Introduced in SQL Server 2014, a new table type was introduced called the *in-memory* table. This moves the storage of tables from the disk to memory, radically improving the performance. But, not all tables or all workloads will benefit from this new functionality. You need to keep an eye on your general query performance metrics for in-memory tables and take advantage of the specific DMVs that let you monitor the in-memory tables. I'll be covering all this in detail in Chapter 24. If your workload doesn't work well with in-memory tables or you just don't have enough memory in the system, you may need to move those in-memory tables back to disk storage.

Increasing System Memory

The memory requirement of SQL Server depends on the total amount of data processed by SQL activities. It is not directly correlated to the size of the database or the number of incoming SQL queries. For example, if a memory-intensive query performs a cross join between two small tables without any filter criteria to narrow down the result set, it can cause high stress on the system memory.

One of the easiest and quickest resolutions is to simply increase system memory by purchasing and installing more. However, it is still important to find out what is consuming the physical memory because if the application workload is extremely memory intensive, you could soon be limited by the maximum amount of memory a system can access. To identify which queries are using more memory, query the `sys.dm_exec_query_memory_grants` DMV and collect metrics on queries and their I/O use. Just be careful when running queries against this DMV using a JOIN or ORDER BY statement; if your system is already under memory stress, these actions can lead to your query needing its own memory grant.

Changing from a 32-Bit to a 64-Bit Processor

Switching the physical server from a 32-bit processor to a 64-bit processor (and the attendant Windows Server software upgrade) radically changes the memory management capabilities of SQL Server. The limitations on SQL Server for memory go from 3GB to a limit of up to 24TB depending on the version of the operating system and the specific processor type.

Prior to SQL Server 2012, it was possible to add up to 64GB of data cache to a SQL Server instance through the use of Address Windowing Extensions. These were removed from SQL Server 2012, so a 32-bit instance of SQL Server is limited to accessing only 3GB of memory. Only small systems should be running 32-bit versions of SQL Server prior to 2017 because of this limitation.

SQL Server 2017 does not support the x86 chip set. You must move on to a 64-bit processor to use 2017.

Compressing Data

Data compression has a number of excellent benefits for storing and retrieving information. It has an added benefit that many people aren't aware of: while compressed information is stored in memory, it remains compressed. This means more information can be moved while using less system memory, increasing your overall memory throughput. All this does come at some cost to the CPU, so you'll need to keep an eye on that to be sure you're not just transferring stress. Sometimes you may not see much compression because of the nature of your data.

Enabling 3GB of Process Address Space

Standard 32-bit addresses can map a maximum of 4GB of memory. The standard address spaces of 32-bit Windows operating system processes are therefore limited to 4GB. Out of this 4GB process space, by default the upper 2GB is reserved for the operating system, and the lower 2GB is made available to the application. If you specify a /3GB switch in the `boot.ini` file of the 32-bit OS, the operating system reserves only 1GB of the address space, and the application can access up to 3GB. This is also called *4-gig tuning* (4GT). No new APIs are required for this purpose.

Therefore, on a machine with 4GB of physical memory and the default Windows configuration, you will find available memory of about 2GB or more. To let SQL Server use up to 3GB of the available memory, you can add the /3GB switch in the `boot.ini` file as follows:

```
[boot loader]
timeout=30
default=multi(o)disk(o)rdisk(o)partition(l)\WINNT
[operating systems]
```

```
multi(o)disk(o)rdisk(o)partition(l)\WINNT=
"Microsoft Windows Server 2016 Advanced Server"
/fastdetect /3GB
```

The /3GB switch should not be used for systems with more than 16GB of physical memory, as explained in the following section, or for systems that require a higher amount of kernel memory.

SQL Server 2017 on 64-bit systems can support up to 24TB on an x64 platform. It no longer makes much sense to put production systems, especially enterprise-level production systems, on 32-bit architecture, and you can't with SQL Server 2017.

Addressing Fragmentation

While fragmentation of storage may not sound like a performance issue because of how SQL Server retrieves information from disk and into memory, a page of information is accessed. If you have a high level of fragmentation, that will translate itself straight to your memory management since you have to store the pages retrieved from disk in memory as they are, empty space and all. So, while fragmentation may affect storage, it also can affect memory. I address fragmentation in Chapter 17.

Summary

In this chapter, you were introduced to the Performance Monitor and DMVs. You explored different methods of gathering metrics on memory and memory behavior within SQL Server. Understanding how memory behaves will help you understand how your system is performing. You also saw a number of possible resolutions to memory issues, other than simply buying more memory. SQL Server will make use of as much memory as you can supply it, so manage this resource well.

In the next chapter, you will be introduced to the next system bottleneck, the disk and the disk subsystems.

CHAPTER 3

Disk Performance Analysis

The disks and the disk subsystem, which includes the controllers and connectors and management software, are one of the single slowest parts of any computing system. Over the years, memory has become faster and faster. The same can be said of CPUs. But disks, except for some of the radical improvements we've seen recently with technologies such as solid-state disks (SSDs), have not changed that much; disks are still one of the slowest parts of most systems. This means you're going to want to be able to monitor your disks to understand their behavior. In this chapter, you'll explore areas such as the following:

- Using system counters to gather disk performance metrics
- Using other mechanisms of gathering disk behavior
- Resolving disk performance issues
- Differences when dealing with Linux OS and disk I/O

Disk Bottleneck Analysis

SQL Server can have demanding I/O requirements, and since disk speeds are comparatively much slower than memory and processor speeds, a contention in I/O resources can significantly degrade SQL Server performance. Analysis and resolution of any I/O path bottleneck can improve SQL Server performance significantly. As with any performance metrics, taking a single counter or a single measure and basing your determination of good or bad performance based on that measure will lead to problems. This is even more true when it comes to modern disk and I/O management systems between old-school RAID systems and modern disk virtualization because measuring

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_3

I/O is a complex topic. Plan on using multiple metrics to understand how the I/O subsystem within your environment is behaving. With all the information here, this chapter covers only the basics.

There are other mechanisms in modern systems that are also going to make measuring I/O more difficult. A lot more systems are running virtually and sharing resources including disks. This will lead to a lot more random I/O, so you'll have to take that into account when looking at the measures throughout this chapter. Antivirus programs are a frequent problem when it comes to I/O, so be sure you validate if you're dealing with that prior to using the I/O metrics we're getting ready to talk about. You may also see issues with filter drivers acting as a bottleneck in your I/O paths, so this is another thing to look at.

One thing you need to know about before we talk about metrics and resolutions is how the checkpoint process works. When SQL Server writes data, it first writes it all to memory (and we'll talk about memory issues in Chapter 4). Any pages in memory that have changes in them are known as *dirty pages*. The checkpoint process occurs periodically based on internal measures and your recovery interval settings. The checkpoint process writes the dirty pages to disk and records all the changes to the transaction log. The checkpoint process is the primary driver of the write I/O activity you'll see within SQL Server.

Let's see how we can measure the behavior of the I/O subsystem.

Disk Counters

To analyze disk performance, you can use the counters shown in Table 3-1.

Table 3-1. *Performance Monitor Counters to Analyze I/O Pressure*

Object (Instance[,InstanceN])	Counter	Description	Value
PhysicalDisk (Data-disk, Log-disk)	Disk Transfers/sec	Rate of read/write operations on disk	Maximum value dependent on I/O subsystem
	Disk Bytes/sec	Amount of data transfer to/from per disk per second	Maximum value dependent on I/O subsystem
	Avg. Disk Sec/Read	Average time in ms to read from disk	Average value < 10 ms, but compare to baseline
	Avg. Disk Sec/Write	Average time in ms to write to disk	Average value < 10 ms, but compare to baseline
SQLServer: Buffer Manager	Page reads/sec	Number of pages being read into the buffer manager	Compare to baseline
	Page writes/sec	Number of pages being written out of the buffer manager	Compare to baseline

The PhysicalDisk counters represent the activities on a physical disk. LogicalDisk counters represent logical subunits (or partitions) created on a physical disk. If you create two partitions, say R: and S:, on a physical disk, then you can monitor the disk activities of the individual logical disks using logical disk counters. However, because a disk bottleneck ultimately occurs on the physical disk, not on the logical disk, it is usually preferable to use the PhysicalDisk counters.

Note that for a hardware redundant array of independent disks (RAID) subsystem (see the "Using a RAID Array" section for more on RAID), the counters treat the array as a single physical disk. For example, even if you have ten disks in a RAID configuration, they will all be represented as one physical disk to the operating system, and subsequently you will have only one set of PhysicalDisk counters for that RAID subsystem. The same point applies to storage area network (SAN) disks (see the "Using a SAN System" section for specifics). You'll also see this in many of the more modern disk systems and virtual disks. Because of this, some of the numbers represented in Table 3-1 may be radically lower (or higher) than what your system can support.

Take all these numbers as general guidelines for monitoring your disks and adjust the numbers to account for the fact that technology is constantly shifting, and you may see different performance as the hardware improves. We're moving into more and more solid-state drives and even SSD arrays that make disk I/O operations orders of magnitude faster. Where we're not moving in SSD, we're taking advantage of iSCSI interfaces. As you work with these types of hardware, keep in mind that these numbers are more in line with platter-style disk drives and that those are fast becoming obsolete.

Disk Transfers/Sec

Disk Transfers/sec monitors the rate of read and write operations on the disk. A typical hard disk drive today can do about 180 disk transfers per second for sequential I/O (IOPS) and 100 disk transfers per second for random I/O. In the case of random I/O, Disk Transfers/sec is lower because more disk arm and head movements are involved. OLTP workloads, which are workloads for doing mainly singleton operations, small operations, and random access, are typically constrained by disk transfers per second. So, in the case of an OLTP workload, you are more constrained by the fact that a disk can do only 100 disk transfers per second than by its throughput specification of 1000MB per second.

Note An SSD can be anywhere from around 5,000 IOPS to as much as 500,000 IOPS for some high-end SSD systems. Your monitoring of Disk Transfers/sec will need to scale accordingly. See your vendor for details on this measure.

Because of the inherent slowness of a disk, it is recommended that you keep disk transfers per second as low as possible.

Disk Bytes/Sec

The Disk Bytes/sec counter monitors the rate at which bytes are transferred to or from the disk during read or write operations. A typical disk spinning at 7200RPM can transfer about 1000MB per second. Generally, OLTP applications are not constrained by the disk transfer capacity of the disk subsystem since the OLTP applications access small amounts of data in individual database requests. If the amount of data transfer exceeds the capacity of the disk subsystem, then a backlog starts developing on the disk subsystem, as reflected by the Disk Queue Length counters.

Again, these numbers may be much higher for SSD access since it's largely limited only by the latency caused by the drive to host interface.

Avg. Disk Sec/Read and Avg. Disk Sec/Write

Avg. Disk Sec/Read and Avg. Disk Sec/Write track the average amount of time it takes in milliseconds to read from or write to a disk. Having an understanding of just how well the disks are handling the writes and reads that they receive can be a strong indicator of where problems are. If it's taking more than about 10ms to move the data from or to your disk, you may need to take a look at the hardware and configuration to be sure everything is working correctly. You'll need to get even better response times for the transaction log to perform well.

In terms of measuring performance of your I/O system, these are the single best measure. Sec/Read or Write may not tell you which query or queries are causing problems. These measures will tell you absolutely how your I/O system is behaving, so I would include them along with any other set of metrics you gather.

Buffer Manager Page Reads/Writes

While measuring the I/O system is important, as mentioned earlier, you need to have more than one measure to show how the I/O system is behaving. Knowing the pages being moved into and out of your buffer manager gives you a great indication as to whether the I/O you are seeing is within SQL Server. It's one of the measures you'll want to add to any others when trying to demonstrate an I/O issue.

Additional I/O Monitoring Tools

Just like with all the other tools, you'll need to supplement the information you gather from Performance Monitor with data available in other sources. The really good information for I/O and disk issues are all in DMOs.

Sys.dm_io_virtual_file_stats

This is a function that returns information about the files that make up a database. You call it something like the following:

```
SELECT  *
FROM    sys.dm_io_virtual_file_stats(DB_ID('AdventureWorks2017'), 2) AS
divfs;
```

It returns several interesting columns of information about the file. The most interesting things are the stall data, which is the time that users are waiting on different I/O operations. First, `io_stall_read_ms` represents the amount of time in milliseconds that users are waiting for reads. Then there is `io_stall_write_ms`, which shows you the amount of time that write operations have had to wait on this file within the database. You can also look at the general number, `io_stall`, which represents all waits on I/O for the file. To make these numbers meaningful, you get one more value, `sample_ms`, which shows the amount of time measured. You can compare this value to the others to get a sense of the degree that I/O issues are holding up your system. Further, you can narrow this down to a particular file so you know what's slowing things down in the log or in a particular data file. This is an extremely useful measure for determining the existence of an I/O bottleneck. It doesn't help that much to identify the particular bottleneck. Combine this with wait statistics and the Perfmon metrics mentioned earlier.

Sys.dm_os_wait_stats

This is a useful DMO that shows aggregate information about waits on the system. To determine whether you have an I/O bottleneck, you can take advantage of this DMO by querying it like this:

```
SELECT  *
FROM    sys.dm_os_wait_stats AS dows
WHERE   wait_type LIKE 'PAGEIOLATCH%';
```

What you're looking at are the various I/O latch operations that are causing waits to occur. Like with `sys.dm_io_virtual_status`, you don't get a specific query from this DMO, but it does identify whether you have a bottleneck in I/O. Like many of the performance counters, you can't simply look for a value here. You need to compare the current values to a baseline value to arrive at your current situation.

The WHERE clause shown earlier uses PAGEIOLATCH%, but you should also look for waits related to other I/O processes such as WRITELOG, LOGBUFFER, and ASYNC_IO_COMPLETION.

When you run this query, you get a count of the waits that have occurred as well as an aggregation of the total wait time. You also get a max value for these waits so you know what the longest one was since it's possible that a single wait could have caused the majority of the wait time.

Don’t forget that you can see wait statistics in the Query Store. We’ll cover those in detail in Chapter 11.

Monitoring Linux I/O

For I/O monitoring, you’ll be limited either to SQL Server internals or to taking advantage of Linux-specific monitoring tools such as were mentioned in Chapter 2. The fundamentals of input and output within the Linux system are not very different from those within the Windows OS. The principal difference is just in how you capture disk behavior at the OS level.

Disk Bottleneck Resolutions

A few of the common disk bottleneck resolutions are as follows:

- Optimizing application workload
- Using a faster I/O path
- Using a RAID array
- Using a SAN system
- Using solid-state drives
- Aligning disks properly
- Adding system memory
- Creating multiple files and filegroups
- Moving the log files to a separate physical drive
- Using partitioned tables

I’ll now walk you through each of these resolutions in turn.

Optimizing Application Workload

I cannot stress enough how important it is to optimize an application’s workload in resolving a performance issue. The queries with the highest number of reads or writes will be the ones that cause a great deal of disk I/O. I’ll cover the strategies for optimizing those queries in more detail throughout the rest of this book.

Using a Faster I/O Path

One of the most efficient resolutions, and one that you will adopt any time you can, is to use drives, controllers, and other architecture with faster disk transfers per second. However, you should not just upgrade disk drives without further investigation; you need to find out what is causing the stress on the disk.

Using a RAID Array

One way of obtaining disk I/O parallelism is to create a single pool of drives to serve all SQL Server database files, excluding transaction log files. The pool can be a single RAID array, which is represented in Windows Server 2016 as a single physical disk drive. The effectiveness of a drive pool depends on the configuration of the RAID disks.

Out of all available RAID configurations, the most commonly used RAID configurations are the following (also shown in Figure 3-1):

- *RAID 0*: Striping with no fault tolerance
- *RAID 1:* Mirroring
- *RAID 5:* Striping with parity
- *RAID 1+0*: Striping with mirroring

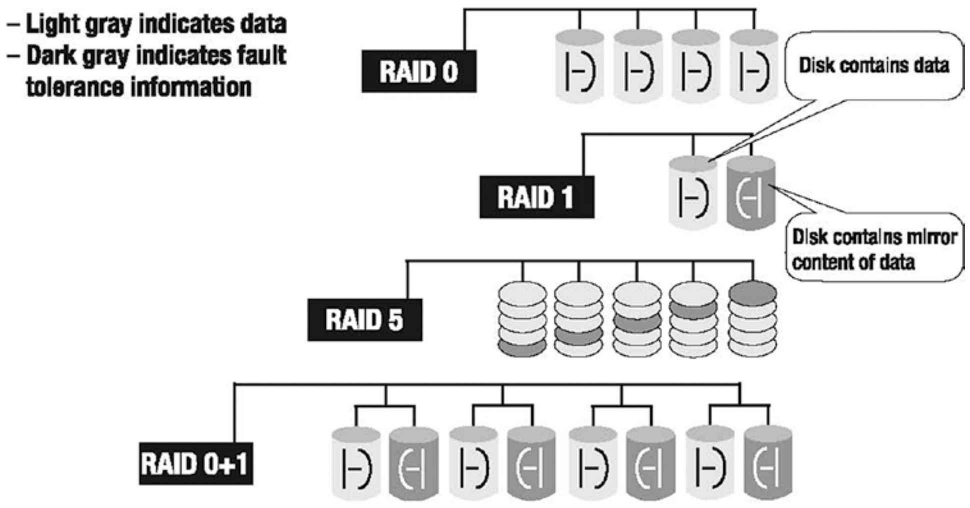

– Light gray indicates data
– Dark gray indicates fault tolerance information
RAID 0
Disk contains data
RAID 1
Disk contains mirror content of data
RAID 5
RAID 0+1

Figure 3-1. *RAID configurations*

RAID 0

Since this RAID configuration has no fault tolerance, you can use it only in situations where the reliability of data is not a concern. The failure of any disk in the array will cause complete data loss in the disk subsystem. Therefore, you shouldn't use it for any data file or transaction log file that constitutes a database, except, possibly, for the system temporary database called `tempdb`. The number of I/Os per disk in RAID 0 is represented by the following equation:

```
I/Os per disk = (Reads + Writes) / Number of disks in the array
```

In this equation, `Reads` is the number of read requests to the disk subsystem, and `Writes` is the number of write requests to the disk subsystem.

RAID 1

RAID 1 provides high fault tolerance for critical data by mirroring the data disk onto a separate disk. It can be used where the complete data can be accommodated in one disk only. Database transaction log files for user databases, operating system files, and SQL Server system databases (`master` and `msdb`) are usually small enough to use RAID 1.

The number of I/Os per disk in RAID 1 is represented by the following equation:

```
I/Os per disk =(Reads + 2 X Writes) / 2
```

RAID 5

RAID 5 is an acceptable option in many cases. It provides reasonable fault tolerance by effectively using only one extra disk to save the computed parity of the data in other disks, as shown in Figure 3-1. When there is a disk failure in RAID 5 configuration, I/O performance becomes terrible, although the system does remain usable while operating with the failed drive.

Any data where writes make up more than 10 percent of the total disk requests is not a good candidate for RAID 5. Thus, use RAID 5 on read-only volumes or volumes with a low percentage of disk writes.

The number of I/Os per disk in RAID 5 is represented by the following equation:

```
I/Os per disk = (Reads + 4 X Writes) / Number of disks in the array
```

As shown in this equation, the write operations on the RAID 5 disk subsystem are magnified four times. For each incoming write request, the following are the four corresponding I/O requests on the disk subsystem:

- One read I/O to read existing data from the data disk whose content is to be modified
- One read I/O to read existing parity information from the corresponding parity disk
- One write I/O to write the new data to the data disk whose content is to be modified
- One write I/O to write the new parity information to the corresponding parity disk

Therefore, the four I/Os for each write request consist of two read I/Os and two write I/Os.

In an OLTP database, all the data modifications are immediately written to the transaction log file as part of the database transaction, but the data in the data file itself is synchronized with the transaction log file content asynchronously in batch operations. This operation is managed by the internal process of SQL Server called the *checkpoint process.* The frequency of this operation can be controlled by using the recovery interval (min) configuration parameter of SQL Server. Just remember that the timing of checkpoints can be controlled through the use of indirect checkpoints introduced in SQL Server 2012.

Because of the continuous write operation in the transaction log file for a highly transactional OLTP database, placing transaction log files on a RAID 5 array will degrade the array's performance. Although, where possible, you should not place the transactional log files on a RAID 5 array, the data files may be placed on RAID 5 since the write operations to the data files are intermittent and batched together to improve the efficiency of the write operation.

RAID 6

RAID 6 is an added layer on top of RAID 5. An extra parity block is added to the storage of RAID 5. This doesn't negatively affect reads in any way. This means that, for reads, performance is the same as RAID 5. There is an added overhead for the additional write, but it's not that large. This extra parity block was added because RAID arrays are becoming so large these days that data loss is inevitable. The extra parity block acts as a check against this to better ensure that your data is safe.

RAID 1+0 (RAID 10)

RAID 1+0 (also referred to RAID 10) configuration offers a high degree of fault tolerance by mirroring every data disk in the array. It is a much more expensive solution than RAID 5, since double the number of data disks are required to provide fault tolerance. This RAID configuration should be used where a large volume is required to save data and more than 10 percent of disk requests are writes. Since RAID 1+0 supports *split seeks* (the ability to distribute the read operations onto the data disk and the mirror disk and then converge the two data streams), read performance is also very good. Thus, use RAID 1+0 wherever performance is critical.

The number of I/Os per disk in RAID 1+0 is represented by the following equation:

```
I/Os per disk = (Reads + 2 X Writes) / Number of disks in the array
```

Using a SAN System

SANs remain largely the domain of large-scale enterprise systems, although the cost has dropped. A SAN can be used to increase the performance of a storage subsystem by simply providing more spindles and disk drives to read from and write to. Because of their size, complexity, and cost, SANs are not necessarily a good solution in all cases. Also, depending on the amount of data, direct-attached storage (DAS) can be configured to run faster. The principal strength of SAN systems is not reflected in performance but rather in the areas of scalability, availability, and maintenance.

Another area where SANs are growing are SAN devices that use Internet Small Computing System Interface (iSCSI) to connect a device to the network. Because of how the iSCSI interface works, you can make a network device appear to be locally attached storage. In fact, it can work nearly as fast as locally attached storage, but you get to consolidate your storage systems.

Conversely, you may achieve performance gains by going to local disks and getting rid of the SAN. SAN systems are extremely redundant by design. But, that redundancy adds a lot of overhead to disk operations, especially the type typically performed by SQL Server: lots of small writes done rapidly. While moving from a single local disk to a SAN can be an improvement, depending on your systems and the disk subsystem you put together, you could achieve even better performance outside the SAN.

Using Solid-State Drives

Solid-state drives are taking the disk performance world by storm. These drives use memory instead of spinning disks to store information. They're quiet, lower power, and supremely fast. However, they're also quite expensive when compared to hard disk drives (HDDs). At this writing, it costs approximately $.03/GB for an HDD and $.90/GB for an SSD. But that cost is offset by an increase in speed from approximately 100 operations per second to 5,000 operations per second and up. You can also put SSDs into arrays through a SAN or RAID, further increasing the performance benefits. There are a limited number of write operations possible on an SSD drive, but the failure rate is no higher than that from HDDs so far. There are also hybrid solutions with varying price points and performance metrics. For a hardware-only solution, implementing SSDs is probably the best operation you can do for a system that is I/O bound.

Aligning Disks Properly

Windows Server 2016 aligns disks as part of the install process, so modern servers should not be running into this issue. However, if you have an older server, this can still be a concern. You'll also need to worry about this if you're moving volumes from a pre-Windows Server 2008 system. You will need to reformat these in order to get the alignment set appropriately. The way data is stored on a disk is in a series of *sectors* (also referred to as *blocks*) that are stored on tracks. A disk is out of alignment when the size of the track, determined by the vendor, consists of a number of sectors different from the default size you're writing to. This means that one sector will be written correctly, but the next one will have to cross two tracks. This can more than double the amount of I/O required to write or read from the disk. The key is to align the partition so that you're storing the correct number of sectors for the track.

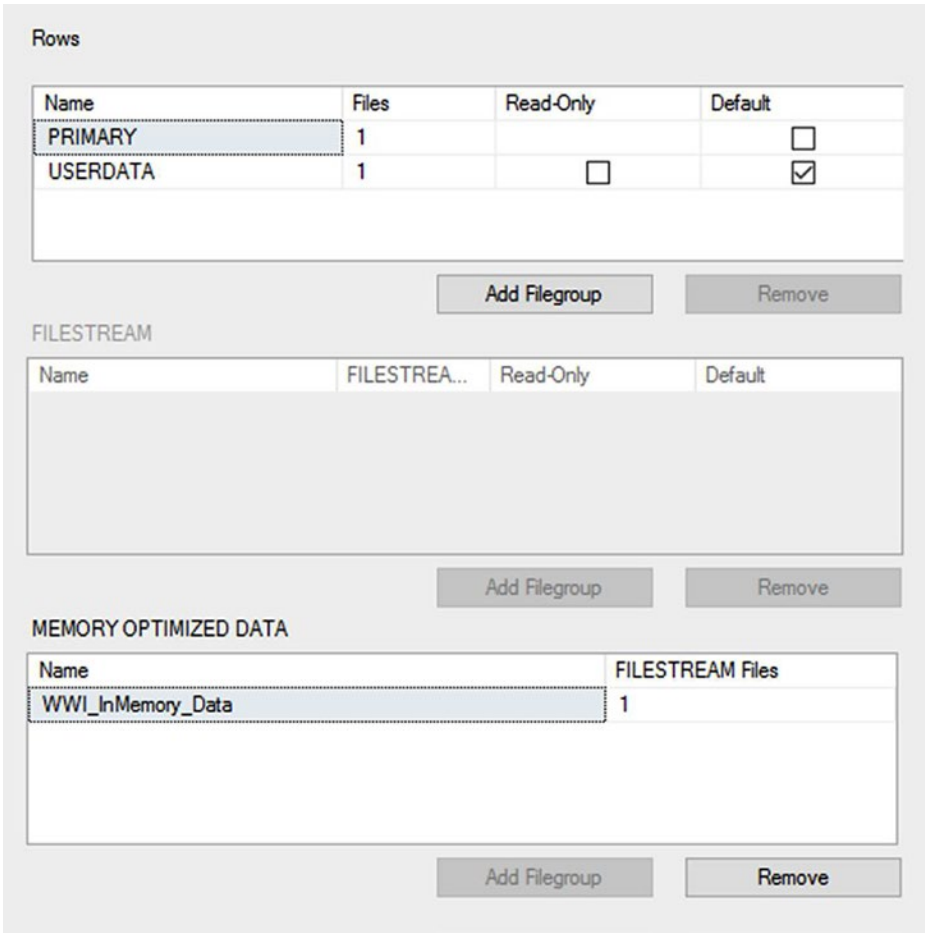
Adding System Memory

When physical memory is scarce, the system starts writing the contents of memory back to disk and reading smaller blocks of data more frequently, or reading large blocks, both of which cause a lot of paging. The less memory the system has, the more the disk subsystem is used. This can be resolved using the memory bottleneck resolutions enumerated in the previous section.

Creating Multiple Files and Filegroups

In SQL Server, each user database consists of one or more data files and usually one transaction log file. The data files belonging to a database can be grouped together in one or more filegroups for administrative and data allocation/placement purposes. For example, if a data file is placed in a separate filegroup, then write access to all the tables in the filegroup can be controlled collectively by making the filegroup read-only (transaction log files do not belong to any filegroup).

You can create a filegroup for a database from SQL Server Management Studio, as shown in Figure 3-2. The filegroups of a database are presented in the Filegroups pane of the Database Properties dialog box.



Rows

Name	Files	Read-Only	Default
PRIMARY	1		☐
USERDATA	1	☐	☑

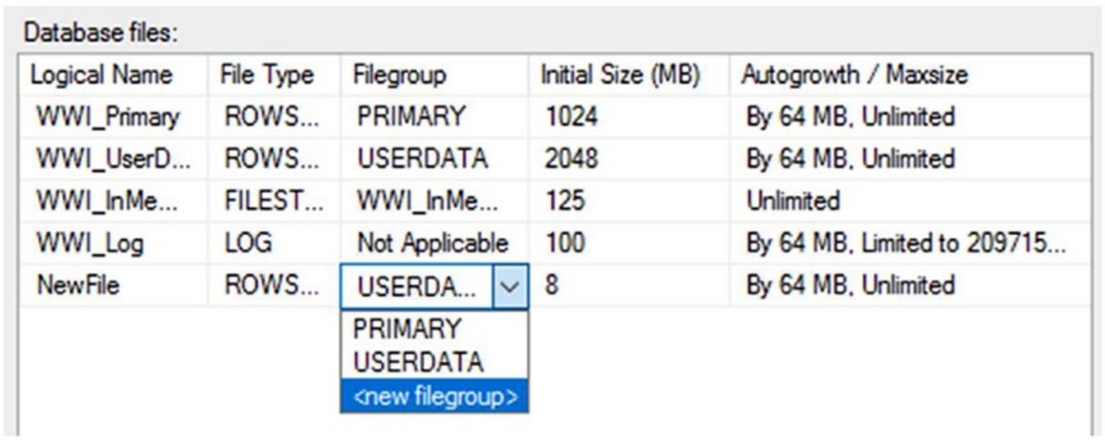
Name	FILESTREA...	Read-Only	Default

Name	FILESTREAM Files
WWI_InMemory_Data	1

Figure 3-2. *Filegroups configuration*

In Figure 3-2, you can see that there are three filegroups defined for the WideWorldImporters database. You can add multiple files to multiple filegroups distributed across multiple I/O paths so that work can be done in parallel across the groups and distributed storage after you also move your database objects into those different groups, literally putting multiple spindles and multiple I/O paths to work. But, simply throwing lots of files, even on different disks, through a single disk controller may result in worse performance, not better.

You can add a data file to a filegroup in the Database Properties dialog box in the Files window by selecting from the drop-down list, as shown in Figure 3-3.



Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize
WWI_Primary	ROWS...	PRIMARY	1024	By 64 MB, Unlimited
WWI_UserD...	ROWS...	USERDATA	2048	By 64 MB, Unlimited
WWI_InMe...	FILEST...	WWI_InMe...	125	Unlimited
WWI_Log	LOG	Not Applicable	100	By 64 MB, Limited to 209715...
NewFile	ROWS...	USERDA...	8	By 64 MB, Unlimited

Figure 3-3. *Data files configuration*

You can also do this programmatically, as follows:

```
ALTER DATABASE WideWorldImporters
ADD FILEGROUP Indexes;
ALTER DATABASE WideWorldImporters
ADD FILE
    (
        NAME = AdventureWorks2017_Data2,
        FILENAME = 'c:\DATA\WWI_Index.ndf',
        SIZE = 20GB,
        FILEGROWTH = 10%
    )
TO FILEGROUP Indexes;
```

By separating tables that are frequently joined into separate filegroups and then putting files within the filegroups on separate disks or LUNS, the separated I/O paths can result in improved performance, assuming of course the paths to those disks are properly configured and not overloaded (do not mistake more disks for automatically more I/O; it just doesn't work that way). For example, consider the following query:

```
SELECT si.StockItemName,
       s.SupplierName
FROM Warehouse.StockItems AS si
JOIN Purchasing.Suppliers AS s
    ON si.SupplierID = s.SupplierID;
```

If the tables `Warehouse.StockItems` and `Purchasing.Suppliers` are placed in separate filegroups containing one file each, the disks can be read from multiple I/O paths, increasing performance.

It is recommended for performance and recovery purposes that, if multiple filegroups are to be used, the primary filegroup should be used only for system objects, and secondary filegroups should be used only for user objects. This approach improves the ability to recover from corruption. The recoverability of a database is higher if the primary data file and the log files are intact. Use the primary filegroup for system objects only, and store all user-related objects on one or more secondary filegroups.

Spreading a database into multiple files, even on the same drive, makes it easy to move the database files onto separate drives, making future disk upgrades easier. For example, to move a user database file (`WWI_Index.ndf`) to a new disk subsystem (F:), you can follow these steps:

1. Detach the user database as follows:

   ```
   USE master;
   GO
   EXEC sp_detach_db 'WideWorldImporters';
   GO
   ```

2. Copy the data file `WWI_Index.ndf` to a folder `F:\Data\` on the new disk subsystem.

3. Reattach the user database by referring files at appropriate locations, as shown here:

```
USE master;
GO
sp_attach_db 'WideWorldImporters',
 'R:\DATA\WWI_Primary.mdf',
’R:\DATA\WWI_UserData.ndf’,
 'F:\DATA\WWI_Indexes.ndf',
’R:\DATA\WWI_InMemory.ndf’,
 'S:\LOG\WWI_Log.1df ';
GO
```

4. To verify the files belonging to a database, execute the following commands:

```
USE WideWorldImporters;
GO
SELECT * FROM sys.database_files;
GO
```

Moving the Log Files to a Separate Physical Disk

SQL Server transaction log files should always, when possible, be located on a separate hard disk drive from all other SQL Server database files. Transaction log activity primarily consists of sequential write I/O, unlike the nonsequential (or random) I/O required for the data files. Separating transaction log activity from other nonsequential disk I/O activity can result in I/O performance improvements because it allows the hard disk drives containing log files to concentrate on sequential I/O. But, remember, there are random transaction log reads, and the data reads and writes can be sequential as much as the transaction log. There is just a strong tendency of transaction log writes to be sequential.

However, creating a single disk for all your log files just brings you back to random I/O again. If this particular log file is mission critical, it may need its own storage and path to maximize performance.

The major portion of time required to access data from a hard disk is spent on the physical movement of the disk spindle head to locate the data. Once the data is located, the data is read electronically, which is much faster than the physical movement of the head. With only sequential I/O operations on the log disk, the spindle head of the log disk can write to the log disk with a minimum of physical movement. If the same disk is used for data files, however, the spindle head has to move to the correct location before writing to the log file. This increases the time required to write to the log file and thereby hurts performance.

Even with an SSD disk, isolating the data from the transaction log means the work will be distributed to multiple locations, improving the performance.

Furthermore, for SQL Server with multiple OLTP databases, the transaction log files should be physically separated from each other on different physical drives to improve performance. An exception to this requirement is a read-only database or a database with few database changes. Since no online changes are made to the read-only database, no write operations are performed on the log file. Therefore, having the log file on a separate disk is not required for read-only databases.

As a general rule of thumb, you should try, where possible, to isolate files with the highest I/O from other files with high I/O. This will reduce contention on the disks and possibly improve performance. To identify those files using the most I/O, reference `sys.dm_io_virtual_file_stats`.

Using Partitioned Tables

In addition to simply adding files to filegroups and letting SQL Server distribute the data between them, it's possible to define a horizontal segmentation of data called a *partition* so that data is divided between multiple files by the partition. A filtered set of data is a segment; for example, if the partition is by month, the segment of data is any given month. Creating a partition moves the segment of data to a particular filegroup and only that filegroup. While partitioning is primarily a tool for making data management easier, you can see an increase in speed in some situations because when querying against well-defined partitions, only the files with the partitions of data you're interested in will be accessed during a given query through a process called *partition elimination*. If you assume for a moment that data is partitioned by month, then each month's data file can be set to read-only as each month ends. That read-only status means you'll recover the system faster, and you can compress the storage resulting in some performance improvements. Just remember that partitions are primarily a manageability feature. While you can see some performance benefits from them in certain situations, it shouldn't be counted on as part of partitioning the data. SQL Server 2017 supports up to 15,000 partitions (just remember, that's a limit, not a goal). Let me repeat, partitioning is absolutely not a performance enhancement tool.

Summary

This chapter focused on gathering and interpreting metrics about the behavior of your disks. Just remember that every set of hardware can be fundamentally different, so applying any hard-and-fast set of metrics around behavior can be problematic. You now have the tools to gather disk performance metrics using Performance Monitor and some T-SQL commands. The resolutions for disk bottlenecks are varied but must be explored if you are dealing with bottlenecks related to disk behavior.

The next chapter completes the examination of system bottlenecks with a discussion of the CPU.

CHAPTER 4

CPU Performance Analysis

This chapter concludes the book's exploration of the system, with a discussion about CPU, network, and general SQL Server metrics. The CPU is the work engine of a system and keeps everything running. All the different calculations required for gathering and delivering data, maintaining the system, and ordering access are performed by the CPU. Getting bottlenecked on the CPU can be a difficult process to work out of. Unlike memory, which you can sometimes easily install more of, or disks, which you can sometimes easily add more of or upgrade, CPUs are an integral part of the system you're running on and can frequently be upgraded only by buying newer machines. So, you'll want to keep an eye on CPU usage. Networks are seldom a major bottleneck for SQL Server, except of course when dealing with Azure SQL Database, but it's good to keep an eye on them too. Finally, there are some SQL Server internal processes that you'll need to gather metrics on. This chapter covers the following topics:

- How to gather metrics on the processor
- Additional metrics available through T-SQL queries
- Methods for resolving processor bottlenecks

Processor Bottleneck Analysis

SQL Server makes heavy use of any processor resource available. You're more likely to be bottlenecked on I/O or memory, but you can hit issues with your CPUs as well. The measures we're covering here are focused on the operating systems and SQL Server. However, in a virtualized environment, the measures we're looking at for CPU are much less likely to reflect reality. You'll need to deal with whatever hypervisor or system you're

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_4

using for virtualization to understand exactly how some of the OS-level CPU measures are actually reflecting reality. You may be experiencing external pressure or even external throttling, none of which will be visible with the counters outlined here.

You can use the Performance Monitor counters in Table 4-1 to analyze pressure on the processor resource.

Table 4-1. *Performance Monitor Counters to Analyze CPU Pressure*

Object (Instance[,InstanceN])	**Counter**	**Description**	**Value**
Processor(_Total)%	Processor Time	Percentage of time processor was busy	Average value < 80%, but compare to baseline
	% Privileged	Percentage of processor time spent in privileged mode	Average value < 10%, but compare to baseline
System	Processor Queue Length	Number of requests outstanding on the processor	Average value < 2, but compare to baseline
	Context Switches/sec	Rate at which processor is switched per processor from one thread to another	Average value < 5,000, but compare to baseline
SQL Server:SQL Statistics	Batch Requests/sec	SQL command batches received per second	Based on your standard workload
	SQL Compilations/sec	Number of times SQL is compiled	Based on your standard workload
	SQL Recompilations/sec	Number of recompiles	

Let's discuss these counters in more detail.

% Processor Time

% Processor Time should not be consistently high (greater than 80 percent). The effect of any sustained processor time greater than 90 percent is effectively the same as that of 100 percent. If % Processor Time is consistently high and disk and network counter values are low, your first priority must be to reduce the stress on the processor. Just remember that the numbers here are simply suggestions; people can disagree with these numbers for valid reasons. Use them as a starting point for evaluating your system, not as a specific recommendation.

For example, if % Processor Time is 85 percent and you are seeing excessive disk use by monitoring I/O counters, it is quite likely that a major part of the processor time is spent on managing the disk activities. This will be reflected in the % Privileged Time counter of the processor, as explained in the next section. In that case, it will be advantageous to optimize the disk bottleneck first. Further, remember that the disk bottleneck in turn can be because of a memory bottleneck, as explained earlier in the chapter.

You can track processor time as an aggregate of all the processors on the machine, or you can track the percentage utilization individually to particular processors. This allows you to segregate the data collection in the event that SQL Server runs on three processors of a four-processor machine. Remember, you might be seeing one processor maxed out while another processor has little load. The average value wouldn't reflect reality in that case. Use the average value as just an indicator and the individual values as more of a measure of actual load and processing on the system.

In a virtualized environment, the CPUs may be virtualized so that what you're seeing isn't accurate. So, for example, in a VMware system, if you install the VMware Tools, you'll be able to look at a VM Processor counter to see the processor usage of the host machine. Using this measure you can tell whether the CPU usage you're seeing in your OS is reflected in the hosting machine or whether you're actually just maxing out your virtual CPUs. On the other hand, running HyperV, you'd need to look to \Hyper-V Hypervisor Logical Processor(_Total)\% Total Run Time for the same measure. You'll have other measures depending on the hypervisor you're using.

% Privileged Time

Processing on a Windows server is done in two modes: *user mode* and *privileged* (or *kernel*) mode. All system-level activities, including disk access, are done in privileged mode. If you find that % Privileged Time on a dedicated SQL Server system is 20 to 25 percent or more, then the system is probably doing a lot of external processing. It could be I/O, a filter driver such as encryption services, defective I/O components, or even out-of-date drivers. The % Privileged Time counter on a dedicated SQL Server system should be at most 5 to 10 percent, but use your baseline to establish what looks like normal behavior on your systems.

Processor Queue Length

Processor Queue Length is the number of threads in the processor queue. (There is a single processor queue, even on computers with multiple processors.) Unlike the disk counters, the Processor Queue Length counter does not read threads that are already running. On systems with lower CPU utilization, the Processor Queue Length counter is typically 0 or 1.

A sustained Processor Queue Length counter of greater than 2 generally indicates processor congestion. Because of multiple processors, you may need to take into account the number of schedulers dealing with the processor queue length. A processor queue length more than two times the number of schedulers (usually 1:1 with processors) can also indicate a processor bottleneck. Although a high % Processor Time counter indicates a busy processor, a sustained high Processor Queue Length counter is a more certain indicator. If the recommended value is exceeded, this generally indicates that there are more threads ready to run than the current number of processors can service in an optimal way.

Context Switches/Sec

The Context Switches/sec counter monitors the combined rate at which all processors on the computer are switched from one thread to another. A context switch occurs when a running thread voluntarily relinquishes the processor, is preempted by a higher-priority ready thread, or switches between user mode and privileged mode to use an executive or subsystem service. It is the sum of Thread:Context Switches/sec for all threads running on all processors in the computer, and it is measured in numbers of switches.