realtimepublishers.com™

The Definitive Guide™ *To*

SQL Server Performance Optimization

VERITAS™

Don Jones

Introduction

By Sean Daily, Series Editor

Welcome to *The Definitive Guide to SQL Server Performance Optimization*!

The book you are about to read represents an entirely new modality of book publishing and a major first in the publishing industry. The founding concept behind Realtimepublishers.com is the idea of providing readers with high-quality books about today's most critical IT topics—at no cost to the reader. Although this may sound like a somewhat impossible feat to achieve, it is made possible through the vision and generosity of corporate sponsors such as VERITAS, who agree to bear the book's production expenses and host the book on its Web site for the benefit of its Web site visitors.

It should be pointed out that the free nature of these books does not in any way diminish their quality. Without reservation, I can tell you that this book is the equivalent of any similar printed book you might find at your local bookstore (with the notable exception that it won't cost you $30 to $80). In addition to the free nature of the books, this publishing model provides other significant benefits. For example, the electronic nature of this eBook makes events such as chapter updates and additions, or the release of a new edition of the book possible to achieve in a far shorter timeframe than is possible with printed books. Because we publish our titles in "real-time"—that is, as chapters are written or revised by the author—you benefit from receiving the information immediately rather than having to wait months or years to receive a complete product.

Finally, I'd like to note that although it is true that the sponsor's Web site is the exclusive online location of the book, this book is by no means a paid advertisement. Realtimepublishers is an independent publishing company and maintains, by written agreement with the sponsor, 100% editorial control over the content of our titles. However, by hosting this information, VERITAS has set itself apart from its competitors by providing real value to its customers and transforming its site into a true technical resource library—not just place to learn about its company and products. It is my opinion that this system of content delivery is not only of immeasurable value to readers, but represents the future of book publishing.

As series editor, it is my raison d'être to locate and work only with the industry's leading authors and editors, and publish books that help IT personnel, IT managers, and users to do their everyday jobs. To that end, I encourage and welcome your feedback on this or any other book in the Realtimepublishers.com series. If you would like to submit a comment, question, or suggestion, please do so by sending an email to feedback@realtimepublishers.com, leaving feedback on our Web site at www.realtimepublishers.com, or calling us at (707) 539-5280.

Thanks for reading, and enjoy!

Sean Daily

Series Editor

Foreword

I've been tuning SQL Server databases for the past decade, and it's a topic that I'm passionate about. Unfortunately, I've seen the same problem play out over and over: Database Administrators (DBAs) blame the application for performance problems and developers blame the DBAs. Who's right, who's wrong? The answer: It doesn't really matter who is at fault. We should be focused on delivering scalable applications that keep our users happy.

Why should we care about database performance tuning in the first place? There are many obvious answers, but when you boil it all down, our applications need to run fast enough to please the consumers of our applications. Applications and computer technology in general lose value, or become useless altogether, if information can't be served up in a timely manner. Performance tuning helps to keep the applications we build useable. But how fast is fast enough?

I firmly believe that there is no such thing as a perfectly tuned application—at least, a complex one that is used in the real world. You can always make an application run "just a little bit faster," so it's important to have some way to determine that it is finally fast enough. The correct answer depends on how your users plan to interact in addition to a myriad of other determining factors. Many people get wrapped around the performance tuning axle because they don't understand the important difference between server performance and server health. Performance is simply a hard cold number. My CPU is running at 87 percent utilization. Is that good or bad? That statistic is almost meaningless in a contextual vacuum. The real question becomes, Is 87 percent good enough to satisfy response time and throughput requirements? You might say that your server is healthy if 87 percent is good enough. This book starts by explaining important concepts behind performance and server health that lay the groundwork for the practical tuning tips and tricks that follow.

A second reason for caring about performance tuning is financially motivated: Too many people solve a database performance problem by throwing hardware at it. Users complain that the database is slow, so an IT staff may add more memory or a few extra CPUs, which can add up. What if you simply needed a new index on a large table? What if adding the extra hardware doesn't even solve the problem, which is the outcome more often than not? Firm understanding of database performance tuning principles can help you avoid costly server upgrades when a simple tweak to the application or database configuration is all that's really necessary. Don's book will give you insights into understanding where the real problem lies and what you can do about it. Performance tuning can seem like a dry topic, but saving your company from a five- or six-figure hardware upgrade might not look bad during your annual review.

Database performance tuning is an art form as much as it is a science. For better or worse, few people in the technical community have all the right pieces of knowledge to fully tune a complex database application, and application performance often suffers accordingly. What can you do about it? First and foremost, more people need to be educated about the intricacies of database performance tuning. In particular, DBAs need to learn a little bit more about the application space, and application developers need to learn a little bit more about the back end. Don Jones' book has compelling technical content, but it's different from many performance tuning books I've read and reviewed in an important way: I was impressed by how well Don was able to distill complex SQL Server tuning concepts into language that was easy to approach and understand. The book covers complex and meaty technical concepts but isn't weighed down by DBA jargon. Both DBAs and application developers who read this book will pick up new tips and tricks. And, hopefully, we will collectively build more efficient and more scalable SQL Server applications.

Brian Moran

July 2002

VERITAS™

Copyright Statement

realtimepublishers.com™

VERITAS™

Chapter 1: Introduction to SQL Server Performance

Welcome to *The Definitive Guide to SQL Server Performance.* This book is one that I've always wanted to write because I'm often called in on consulting jobs to help database developers and administrators narrow down the performance problems with their SQL Server-based applications. SQL Sever forms the backbone of many companies' mission-critical applications, and so SQL Server performance is always a hot topic for those companies' administrators. In the years that I've been teaching SQL Server and consulting in SQL Server-based shops, I've seen and heard about a number of SQL Server "worst practices," and this book is an opportunity to share those performance-related issues with you so that you can avoid them in your own environment.

Defining Performance and Health

What, exactly, is performance? The computer industry actually recognizes two related terms: *performance* and *health.* Performance represents exactly how a computer or software application is behaving, often in numeric terms. For example, you might express processor performance in terms of utilization percentage or query performance in terms of execution time. The important thing to remember about performance is that it's absolute: Performance doesn't care what numbers are good or bad. For example, a processor with a utilization of 100 percent isn't, from a performance standpoint, inherently *bad.* Health, however, wraps up performance data and applies real-world expectations and information. Health defines the performance that is considered good, or healthy, and performance that is conferred bad, or unhealthy. Health often applies ranges of performance values. For example, a processor with a utilization of 0 to 70 percent might be considered healthy, while utilization of 70 to 80 percent might be considered a warning zone, and utilization greater than 80 percent might be considered unhealthy.

> 🖉 Computer folks often talk about the difference between *data,* which is simply raw numbers, and *information,* which places data into a useful, real-world context. That pretty much describes the difference between performance and health, too. Performance is raw numbers, such as transactions per second or processor utilization. Health places performance into a real-world context and tells you how well your servers are doing or how badly you need an upgrade.

Computer administrators of all kinds are always seeking to improve their servers' performance, and SQL Server administrators—often referred to as database administrators (DBAs)—are no exception. The problem with most performance tuning is that it deals only with performance, which is essentially the same as working in a vacuum because performance by itself doesn't mean very much. Instead, you have to start thinking about performance as a way to improve server *health.*

Here's a SQL Server-specific example: Imagine that you've got a SQL Server computer that supports your company's primary line-of-business application. The average query on the server executes in just a couple of milliseconds, which is generally quite acceptable. The application's users, however, complain that the application seems too slow and that it seems to be getting slower every day. You're actually comparing performance—the absolute speed of the queries— with health—the users' relative feeling of the application's responsiveness. Like comparing apples and oranges, comparing performance and health won't help you. Instead, you need to create a common comparison by defining health criteria for your servers. For example, you

might create health ranges that define query execution speed of less than 1 second as healthy. In that case, your line-of-business SQL Server computer would be considered perfectly healthy, despite your users' complaints.

So what's the point? The point of health is to define acceptable levels of performance for each aspect of a computer or application. In this example, the query execution speed is healthy, but users are still complaining—meaning you need to look at some other aspect of the application to improve the application's overall performance. Defining health criteria, as opposed to looking at raw performance numbers, helps you quickly determine when a particular aspect of a server is outside acceptable ranges and focus on that aspect for performance improvements. At the same time, health allows you to recognize when a particular component is *within* its range of acceptable health, and quickly focus on *other* components for possible performance improvements.

So why the long lecture about health and performance? SQL Server is an incredibly complex system with dozens of various components contributing to a server's overall performance as a database server. If you focus strictly on performance numbers, you'll often find yourself fighting a losing battle, trying to squeeze a few extra performance points out of a server component that is simply running as fast as possible. By creating health standards for these various components, you'll be able to categorize each component's performance as acceptable or unacceptable, and quickly focus on the components of the server that need your attention. Throughout this book, I'll not only help you understand how SQL Server performance works, but also to understand which operational ranges are generally considered healthy and which should be considered unhealthy. These guidelines will help you quickly create health standards for the SQL Server computers in your environment and start applying performance tuning techniques to the server components that need the most help.

How SQL Server Works

If you're an experienced SQL Server developer or DBA, you probably know all that you need to know about how SQL Server works. But understanding SQL Server's basic architecture from an operational standpoint is very different than understanding it from a health and performance standpoint. With that in mind, I want to take a few paragraphs to reexamine what SQL Server does under the hood so that you can start to think about the product's architecture in terms of performance.

The Transaction Log

When you submit a change to SQL Server—using either an INSERT, UPDATE, or DELETE statement—the product doesn't actually run out to the database file on disk and make your changes. Instead, SQL Server makes a note of your changes in its *transaction log,* which is literally a list of all the operations (or transactions) that have been performed on the server. Next, SQL Server goes out to disk and loads the effected data pages into memory. SQL Server makes your change (or changes) to the data pages in memory, then leaves those pages in memory. The theory is that leaving the pages in memory will allow SQL Server to make additional changes to them more quickly because SQL Server can access memory more quickly than disk-based files. (For information about SQL Server stores information on disk, see the sidebar "Pages: Not Just for Books.")

Pages: Not Just for Books

SQL Server stores information on disk in *pages,* and each page consists of exactly 8KB of data. Pages define several of SQL Server's operational characteristics. For example, the data in a single table row cannot exceed the size of a single page. In other words, a single database row cannot exceed about 8KB in size (the actual number is a bit smaller because SQL Server uses a few bytes on each page to track the page's position in the file and for other tasks). Keep in mind that some data types—such as binary columns or large-text columns—aren't always stored within the row's data page. Instead, those data types are stored on their own, dedicated 8KB pages (more than one page, if necessary), and the main row's data page contains a *pointer* to the dedicated pages. You can think of data pages as Web pages, which are limited to 8KB in size. Each page contains all the main data for a single table's row, although pages can also contain hyperlinks (pointers) to other pages that include related information.

An inefficient table design will require just about 4KB, or slightly more, for a single database row. Keep in mind that the *entire row* must fit within a single page and cannot be split across pages. If a table design requires just more than 4KB, then only one row will be able to fit within the page, effectively wasting all the remaining space in the page. An efficient table will require less than 4KB per row, allowing multiple rows to be contained on each page.

This efficiency not only affects disk space utilization but also database performance. Remember, when SQL Server needs to modify a row's data, it loads the entire effected page into memory, including any other rows contained in that same page. Many SQL Server transactions modify many rows at once, so with a single read operation, SQL Server can load multiple rows of data into memory and modify them. If, however, data pages contain only a single row of data, SQL Server must load multiple pages into memory to modify multiple rows. Anytime SQL Server must resort to reading data from disk, performance will suffer.

Eventually, SQL Server will save the modified data pages back to disk. At that time, SQL Server knows that the data contained on those pages is fairly safe because disk storage is much less volatile than memory. So after the data is saved back to disk, SQL Server *checkpoints* the transactions in the transaction log that were associated with that data. The checkpoint lets SQL Server know that the transaction in question was successfully performed and saved to disk.

✎ SQL Server's transaction log maintains the order that the server received transactions. SQL Server always saves data pages back to disk in the order that they occurred. In other words, if you execute ten updates on the server, those updates will be executed, saved to disk, and checkpointed in the same order that you issued them.

The transaction log obviously plays an important role in disaster recovery. If SQL Server experiences a failure, such as a power outage or an outright crash, any modified data pages that are still in memory will be lost. When SQL Server restarts, it examines the transaction log for any uncheckpointed transactions, and immediately re-executes those transactions. This behavior ensures that SQL Server's technique of keeping data pages in memory for some time will not result in a data loss in the event of a server failure. Because the transaction log is used for nearly every transaction that SQL Server performs, the transaction log can become a performance bottleneck. For example, if the transaction log is located on a slower hard disk, that disk's speed will limit the speed at which SQL Server can accept incoming transactions.

📖 I'll discuss how to find and eliminate hardware bottlenecks and other hardware-related performance problems in Chapter 3.

VERITAS™

Indexes

Indexes play a key role in SQL Server performance. If you've ever tried to look up information in an especially large printed book, you'll appreciate the importance of indexes. Imagine, for example, that you have a 2000-page book about database design theory and that you need to find the first page that mentions database normalization. You have two options for finding the information. One way is to perform a *scan,* which simply means skimming through each page until you spot the term "database normalization." Scanning—especially in a large body of data—can be very time consuming. Another method is to use the book's index, which allows you to look up the term "database normalization" in an alphabetical list of other terms, and provides a cross-reference to the pages on which the term can be found. Using an index to locate terms in such a large book would obviously be much faster than scanning.

Indexes aren't, however, always a sure-fire way to improve performance. If you need to find a term in a 10-page brochure, using an index could actually be slower than simply scanning for the desired term. And, while indexes can definitely improve query performance in a large body of data, they *reduce* performance when updating the data. That's because the index must be updated every time the data is updated.

SQL Server allows you to build indexes on specific columns within a table. You can index a single column or index multiple columns (called a *compound index*). Query performance will generally improve if your query includes indexed columns. However, every index you add will reduce SQL Server's performance when updating data because every index is just one more thing that SQL Server has to update when making a change to the actual data. SQL Server actually supports two types of indexes: *clustered* and *nonclustered.*

Clustered Indexes

Clustered indexes physically change the order in which data is stored. Returning to the 2000-page book example, a clustered index would physically reorder all the words within the book so that they appeared in alphabetical order, as if the book's normal index were also the book's actual content. Clustered indexes are very fast because locating an item within the index means that you've located the actual item itself.

However, because clustered indexes affect the physical order of the data as stored on disk, each table can have only one clustered index. After all, you can't physically arrange people, for example, in order both by their age and their height, you have to pick one.

In fact, *every* SQL Server table has a clustered index, whether you know it or not. If you don't explicitly create a clustered index, SQL Server uses a *phantom* clustered index that simply places rows in order by their ordinal number, which is basically the order in which the rows were created. (This order isn't the same as the ROWID referred to in the SQL Server Books Online; this order is an internal ID number that you normally won't see at all.)

> Chapter 4 focuses on query performance tuning, including strategies for using clustered indexes to the best effect.

Nonclustered Indexes

Nonclustered indexes are closer in function to the index in that 2000-page book. A nonclustered index stores a copy of the indexed column (or columns) in order, along with a pointer to the actual data. That means nonclustered indexes make it very easy to quickly locate a particular term or other piece of data but require SQL Server to take an extra step to actually find the data. This extra step is called a *traversal,* and represents the main difference between clustered and nonclustered indexes: When SQL Server locates data in a clustered index, SQL Server is already on the actual page containing the full table row. When SQL Server locates data in a nonclustered index, all that SQL Server has is a pointer to the full table row and must still perform a traversal to look up that pointer and retrieve the actual data from the table row. Figure 1.1 shows the relationship between clustered and nonclustered indexes.

Nonclustered Index on person's first name	Clustered Index on person's ID number	Data page for the person's table row
Abel (10)	1	Ewen Olsen 723 Spruce Ave
Barney (5)	2	Charles Urwiler 1717 Landsdowne Ave
Charles (2)	3	Harry Johnson 882 Windy Beach Blvd
Don (8)	4	Martin McGregor 45 NE 56th Ave
Ewen (1)	5	Barney Sandusky 534 Witchduck Rd
Frank (6)	6	Frank Amundsen 889 Crystal Ave
George (11)	7	Samuel Hinz 1753 Coqui Dr
Harry (3)	8	Don Jones 2843 Mustang Ln
Jacob (9)	9	Jacob Gannon 9931 West Orange Rd
Martin (4)	10	Abel Smith 122 Elm St
Samuel (7)	11	George Harlen PO Box 1723

Seek to the correct spot in the index

Traverse to the correct spot in the clustered index to retrieve the data row

Figure 1.1: Searches in a nonclustered index must still be cross-referenced to the actual data pages. This cross-reference is actually made to the table's clustered index.

> ✎ SQL Server has to perform a traversal only when the nonclustered index in use doesn't contain the data that SQL Server needs. For example, if you were to execute the query
>
> ```
> SELECT * FROM Customers WHERE FirstName = 'Don'
> ```
>
> SQL Server would have to perform the traversal represented in Figure 1.1. However, if you executed the query
>
> ```
> SELECT FirstName FROM Customers WHERE FirstName = 'Don'
> ```
>
> SQL Server wouldn't need to perform a traversal because the data that you requested—the FirstName column—is contained within the nonclustered index. In this instance, the index is also called a *covered index* because the data in the index can satisfy the query. Of course, in this simplistic example, there's no point—why ask SQL Server to return a name when you already know the name? Covered indexes come into play primarily with compound indexes, which include multiple columns.
>
> For example, suppose you build a compound nonclustered index on the table's FirstName and LastName columns. You could then execute the query
>
> ```
> SELECT LastName FROM Customers WHERE FirstName = 'Don'
> ```
>
> In this case, SQL Server would be able to satisfy the query entirely from the compound index without having to traverse to the nonclustered index and the actual data pages because the compound index contains both the search term—FirstName—and the desired result—LastName.

Deciding which table columns should be indexed is definitely a complex, strategic process. Before you start slapping indexes on a table, you need to understand the queries that will be executed against the table. There are several "rules of thumb" that you can use to apply indexes to a table to begin with, and SQL Server contains some useful tools that can help identify the columns that need to be indexed or identify indexes that aren't being used effectively.

> 📖 Chapters 4 and 5 focus on query performance tuning, which is largely a matter of understanding where to place indexes. I'll cover indexing strategies and SQL Server's index tuning tools in those chapters.

The Query Engine and Optimizer

SQL Server's query engine is the heart of the product and is responsible for accepting and executing every query that you execute on the server. The query engine itself is comprised of two primary functions: the *query optimizer* and the actual query execution engine. Although the query engine is largely self-contained, the query optimizer is an important component in performance tuning and is open to manipulation by a skilled database developer or administrator.

The optimizer's job is to examine queries and determine how the execution engine should perform them. The analysis includes the indexes that the engine should utilize, the order in which the query's various clauses should be executed, and so forth. The optimizer uses statistics generated by SQL Server regarding the size and composition of tables and indexes, and plugs those statistics into a statistics model that Microsoft built into the product. The model produces estimated execution times for various query scenarios, such as the time required to execute a query without using an index, the time required if a particular index were utilized, and so forth. The model looks for the execution plan with the smallest estimated execution time and submits that execution plan to the query execution engine.

✎ "There are three kinds of lies: Lies, damn lies, and statistics." — *Benjamin Disraeli*

By default, SQL Server 2000 automatically generates and updates statistics for the databases on the server. You can disable this option by modifying the database's properties. If you do, you'll have to use SQL Server's internal commands to manually generate and update statistics from time to time. The purpose of database statistics is to tell the query optimizer vital information about the database. As I've already mentioned, it can sometimes be quicker to scan through data than to use an index, especially if the amount of data involved is fairly small. Statistics helps the optimizer make that decision, along with countless other decisions.

Unfortunately, statistics and the built-in optimization model don't always closely represent the real world. It's possible for the query optimizer to come up with an execution plan that is anything but optimal simply because the query it has considered doesn't fit well within the statistical model that the optimizer uses. I've worked with several consulting clients who have encountered situations in which the optimizer refuses to use available indexes even though doing so would, in fact, significantly improve query performance.

The moral is this: Always make sure you understand the execution plans that the optimizer generates and make sure that those plans make sense to you. You can always manually tweak the query to force the optimizer to use various indexes or other execution techniques.

📖 I'll explain how to read query execution plans and how to optimize them in Chapters 4 and 5.

The query optimizer is pretty intelligent about searching for fast execution plans. It starts by examining the query for any obvious execution plans. For example, simple queries, such as

```
SELECT FirstName FROM Customers
```

are often easy to optimize simply because they are so simple. The optimizer might consider the cost of several fairly obvious execution plans, and if one of those plans has a cost that is less than a predetermined threshold, the optimizer will decide that the plan is "cheap enough" and just go with it.

If none of the really obvious plans are "cheap enough," the optimizer will consider more complex queries based on its statistical model. This second phase requires a bit more time, but if it produces a plan that is "cheap enough," the optimizer will submit the plan for execution. If, however, the second phase of optimization doesn't produce an inexpensive-enough plan, the optimizer will begin using a *brute-force* optimization method in which it simply generates every conceivable execution plan until it finds one that has an estimated execution cost less than a predetermined threshold. The optimizer has a time limit for this third phase, and if it doesn't find a "cheap enough" execution plan by the end of the time limit, it will simply go with the cheapest plan that it has come up with by then.

The purpose of this entire optimization process is to produce the fastest execution plan possible in the shortest amount of time. The optimizer recognizes that each successive phase of optimization takes longer, so it will only perform those phases if doing so will take less time than the query plans it has considered up until then.

☞ The optimizer's execution plans are based partly upon your database's statistics. If those statistics are out of date (which can occur if you disable automatic statistics updates and forget to manually update them), the optimizer will generate and use execution plans that are no longer appropriate for your database.

📖 I'll show you how to update statistics in Chapter 4.

The optimizer doesn't simply rely on statistics, though. For example, when considering whether to use parallelism (which I'll discuss shortly), the optimizer considers the server's current level of processor utilization. For many of those resource-dependent calculations, the optimizer makes different decisions *every time the query is executed.* So, although the majority of the optimizer's decisions are based on statistical modeling, the optimizer does factor in the server's configuration and workload when determining the cost of a proposed execution plan.

Caches

Disk input/output (I/O) operations are the bane of a product such as SQL Server. Most often, the server's disk subsystem is the bottleneck, requiring the server to wait while information is retrieved from disk. SQL Server includes several features, such as read-ahead caching, that can reduce the negative impact of disk performance. However, in the end, it's almost always disk performance that creates poor server health. In fact, SQL Server's query optimizer factors in the estimated number of *physical reads* (the amount of information that must be read from disk) in all of its query execution plan costs.

To help alleviate the impact of disk I/O, SQL Server configures several *memory caches,* in which information read from disk can be saved in memory for future use. SQL Server always looks to its caches—through a process called a *logical read—*before attempting to read information from disk.

☞ RAM, wonderful RAM! In reality, SQL Server only reads data from its memory caches. In the event that a logical read results in a *cache miss* (meaning the data requested isn't in memory), SQL Server performs a physical read to pull the desired information from disk and into the cache, where it can then satisfy the logical read. This behavior means that SQL Server's best friend is memory. It's not uncommon for large-scale SQL Server implementations to use Windows 2000 (Win2K) Advanced Server (or Windows .NET Enterprise Server) simply to take advantage of the products' 8GB memory capability. Particularly large implementations might even need the 64GB memory support of Win2K Datacenter Server.

VERITAS™

SQL Server uses caches to avoid not only physical reads but also time-consuming query optimization. Whenever SQL Server creates a query execution plan, it caches that plan for future use. Any subsequent uses of the exact same query won't require a run through the query optimizer; instead, SQL Server will use the cached execution plan. SQL Server's caches can include:

- Stored procedures

- Prepared statements

- Ad-hoc queries

- Replication procedures

- Triggers

- Views

- Defaults

- User tables

- System tables

- Checks

- Rules

Each of these objects can be stored in SQL Server's caches for improved performance.

Understanding Hardware Performance Components

In the end, all performance comes down to hardware. Although database administrators often find themselves tweaking queries for better execution times, they do so only because it's easier than upgrading their server hardware. In fact, most of the performance tuning that you do as a database developer or administrator will be to make your applications run faster on the hardware that you have. That doesn't mean your applications or queries are poorly written; it simply means that you have to cut performance corners to make them run better on your hardware.

Because all performance problems eventually come down to hardware, you need to take the time to understand how SQL Server interacts with the hardware on which it runs. Doing so will help you select better hardware for new SQL Server computers and help you better utilize the hardware on existing SQL Server computers. From a performance point of view, there are four basic hardware components: disks, networks, processors, and memory.

Disk I/O

As I've already mentioned, disk throughput is often the bottleneck on SQL Server computers. Even if you're running 8GB of RAM on a server, your databases can easily consume one hundred times that much space, meaning SQL Server is doomed to constant disk access. There are, however, some tricks to improving disk I/O.

> 📖 I'll show you how to identify and solve disk I/O problems in Chapter 3. Plus, in Chapter 6, I'll explain how you can scale out SQL Server to help solve disk I/O problems.

VERITAS™

Imagine, for example, that you need to produce a handwritten copy of a 1000-page book. You can either use one helper so that the two of you will have to each copy 500 pages, or you can use nine helpers so that you'll each be responsible for only 100 pages. Because you can each copy a fixed number of pages in any given period of time, the more hands you have helping you, the better. The same applies to disks: The more, the merrier. Any single disk drive can transfer a fixed amount of data each second. If you want to speed your disk subsystem, simply increase the number of disks that are feeding data to SQL Server. One easy way to increase the number of disks is by using a redundant array of inexpensive (or independent) disks (RAID) array. In some RAID arrays, such as RAID 5, data is physically spread across the disks in the array, making each disk responsible for a smaller piece of the overall data load.

SQL Server also offers the ability to spread its disk workload across multiple disks (or even across multiple RAID arrays). For example, SQL Server stores its databases and their transaction logs in separate files. Moving those files to separate disks (or arrays) will let SQL Server access both more quickly. SQL Server can also split a database into multiple files, allowing you to spread those files across multiple disks (or arrays) to improve disk performance. Figure 1.2 shows a single database spread across three physical files. Each file, plus the database's transaction log, is located on a separate disk (or array), increasing the number of "hands" that are handling the data.

Disk 0	Disk 1	Disk 2	Disk 3
Database.mdf	Database2.ndf	Database3.ndf	Log.ldf
Contains the Customers table (frequently used)	Contains the FirstName column's index (frequently used)	Contains less-frequently used tables and indexes	Contains the database's transaction log
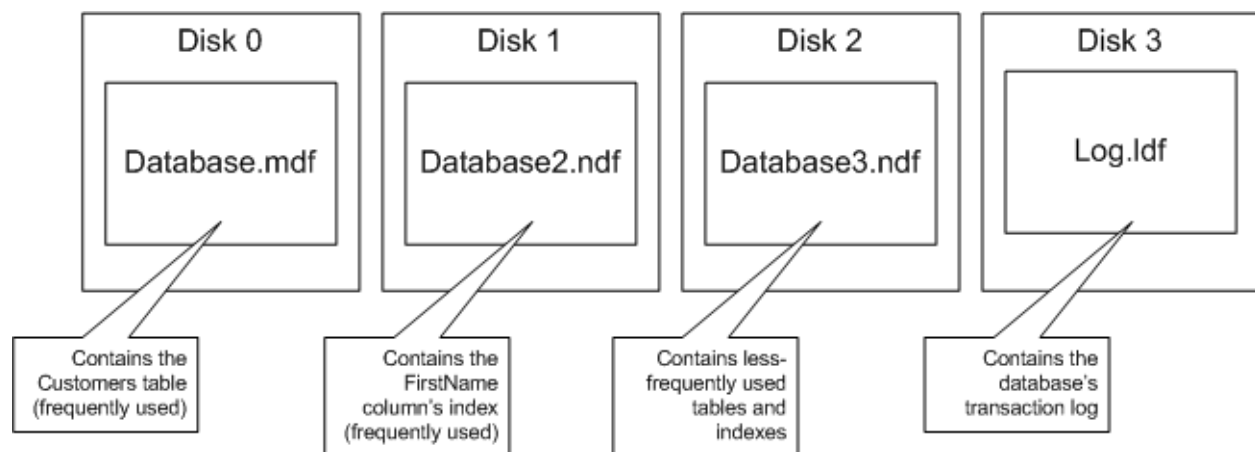
Figure 1.2: You can designate which portions of a database are stored on each file, letting you evenly distribute frequently accessed tables and indexes across the available files, which improves disk performance.

☞ If you really want high-speed disk performance, use a dedicated storage device from a vendor such as EMC. The company's standalone storage devices can attach via fiber-optic cables to multiple servers. Each server receives its own dedicated portion of the device's disk space, which the server sees as a single local drive letter. The devices themselves can contain dozens—or even hundreds— of physical drives, all helping to balance the device's overall throughput.

Network I/O

Network I/O—the speed at which SQL Server can pump data onto your network and get data from your network—isn't usually the first bottleneck SQL Server computers encounter. Most companies run 100Mbps or faster networks, and often install servers on even faster backbones running at 1Gbps. Still, depending upon your specific usage patterns, network congestion can make a SQL Server computer seem as if it's running slower than it should. Here are some tips for creating a network to support large-scale SQL Server usage:

- Connect SQL Server computers to a switch, and spread your client computers out across the switch's other ports. Ideally, use a multispeed switch that allows SQL Server to connect at a higher bandwidth than clients connect at. Doing so will allow the switch to multiplex multiple client conversations to SQL Server at the same time.

- Windows server operating systems (OSs) have an observed maximum throughput of about 500Mbps on today's server hardware (the primary limitation is the PCI bus used by network adapters, not Windows itself). Keep that limitation in mind, and as your server's utilization nears the limit, consider moving to multiple servers to handle your database workload.

- Use bus-mastering Peripheral Component Interconnect (PCI) network adapters, which are capable of accessing server memory without utilizing the server's processor. Ideally, install the network adapters on an otherwise empty PCI bus to give the adapter maximum bandwidth to the server's memory.

- Avoid running other services, such as IIS, on a SQL Server computer. Those services will compete for network bandwidth and might artificially restrict SQL Server's network throughput.

- Create queries that return a minimum amount of information, thereby minimizing network utilization. For example, executing

  ```
  SELECT * FROM Customers
  ```

  is overkill if you only need to retrieve a single customer's first name. Instead, execute a more specific query such as

  ```
  SELECT FirstName FROM Customers WHERE CustomerID = 9
  ```

📖 I'll show you how to identify and resolve common network utilization issues in Chapter 3.

Processors

Processors are the most commonly thought of causes of performance problems. SQL Server is definitely a heavy processor user—every query executed by the server imposes a certain load on the server's processor. SQL Server is optimized to take advantage of multiple processors, and large-scale SQL Server implementations often include 4-way or even 8-way processors. Microsoft has performed SQL Server benchmark testing on 32-way servers running Win2K Datacenter Server.

✎ A "way" in processor lingo means a processor. In other words, a 32-way server contains 32 processors all running in parallel. Win2K Advanced Server supports as many as 8 processors, and Win2K Server limits you to 4 processors.

Multiprocessor computers allow SQL Server 2000 to take advantage of *parallelism,* through which SQL Server uses multiple processors to execute a single query. Parallelism can often allow SQL Server to make some unobvious query execution decisions. For example, suppose you have a midsized table with several thousand rows. You execute a query that could use an index, but SQL Server instead performs a basic table scan. The decision to scan rather than using the index seems odd, until you remember parallelism. SQL Server might determine that it's faster to use 8 processors to perform a table scan—each processor effectively taking an eighth of the table—than to spend the time opening the index, finding the appropriate row, and traversing to the physical data page.

In addition to disk I/O, processor utilization is often the first performance bottleneck you'll run into. Unfortunately, it's also usually the first bottleneck in which you'll "hit a wall" and be unable to change. That's because, unlike disks or network adapters, servers can accept only a fixed number of processors, and generally can accept only specific processor speeds. After you've installed that many, you can't upgrade the processor subsystem any further. Most of the time this limitation means you'll need to start fine-tuning your database operations to use the processors less heavily or more efficiently.

Memory

Memory is one of SQL Server's most important attributes. Thanks to SQL Server's caches, memory can help solve disk I/O problems and even help alleviate some symptoms of processor over-utilization. By default, SQL Server asks the OS for memory as needed and releases memory back to the OS when that memory is no longer required. In theory, this behavior means that SQL Server should work well with other installed applications, because SQL Server doesn't automatically try to "grab" all available memory for itself. In practice, you'll find that many production-class SQL Server implementations need all the memory that you can provide. SQL Server, Standard Edition supports a maximum of 2GB of application memory. This limitation is actually imposed by Win2K Server, which automatically allocates 2GB of memory—whether physical RAM or virtual memory from the system swapfile—to each running application.

✎ SQL Server dynamically manages its memory caches so that they are not swapped to the system swapfile. Allowing the caches to be swapped to disk would be pointless, because the whole point of the caches is to avoid physical disk reads. SQL Server automatically recognizes when it is running on Windows 9x, NT, or Win2K, and adjusts its cache sizes accordingly to prevent the OS' swapping algorithms from swapping the cache memory to disk.

SQL Server, Enterprise Edition, however, can take advantage of the Address Windowing Extensions (AWE) memory capabilities of Win2K Advanced Server and Datacenter Server (or the corresponding .NET Server editions). AWE allows SQL Server to take advantage of the full amount of physical RAM supported by the OS (8GB for Advanced Server and 64GB for Datacenter Server). When running under AWE support, SQL Server does not use dynamic memory management. Instead, it allocates as much memory for itself as possible, by default. You can modify the maximum amount of memory that SQL Server uses by modifying the server's configuration options.

VERITAS™

📖 In Chapter 3, I'll show you how to identify memory problems and configure the server's memory options.

SQL Server Performance Audit Methodology

Measuring server performance and health is a precise science. I've visited too many shops that take a haphazard approach to measuring their servers' health and performance; perhaps only looking at an inconsistent set of performance counters or looking at data only periodically. This haphazard approach isn't uncommon because there really aren't any formal methodologies for measuring server health and performance. With that in mind, I think the best use of this introductory chapter is to provide you with the beginnings of a documented, repeatable methodology for measuring server health and performance. Throughout the rest of this book, I'll build on this basic framework to help you create a methodology that works for you.

Of course, the two most important components of any methodology regarding performance are *consistency* and *regularity*. You have to *regularly* monitor server performance to accurately ascertain health (for the same reasons that your doctor bugs you to come in for those yearly physicals). And, when you do monitor performance, you have to do so in a *consistent* fashion, measuring the same things every time. If your doctor measured your lung capacity one year and your hearing the year after, he wouldn't have a good idea of the health of either your lungs or ears; only by regularly monitoring the same things can you build an accurate picture of health.

Performance Tools

You'll likely use several tools when measuring server performance, but the most important of them will probably be the System Monitor (which is Win2K's name for the NT Performance Monitor, and is contained in the Performance console on the Start menu, under Administrative Tools).As Figure 1.3 shows, System Monitor displays performance information in an easy-to-read graph. You can also use System Monitor to produce performance log files, which you can review later (as opposed to trying to read the performance data in real time on the graph).
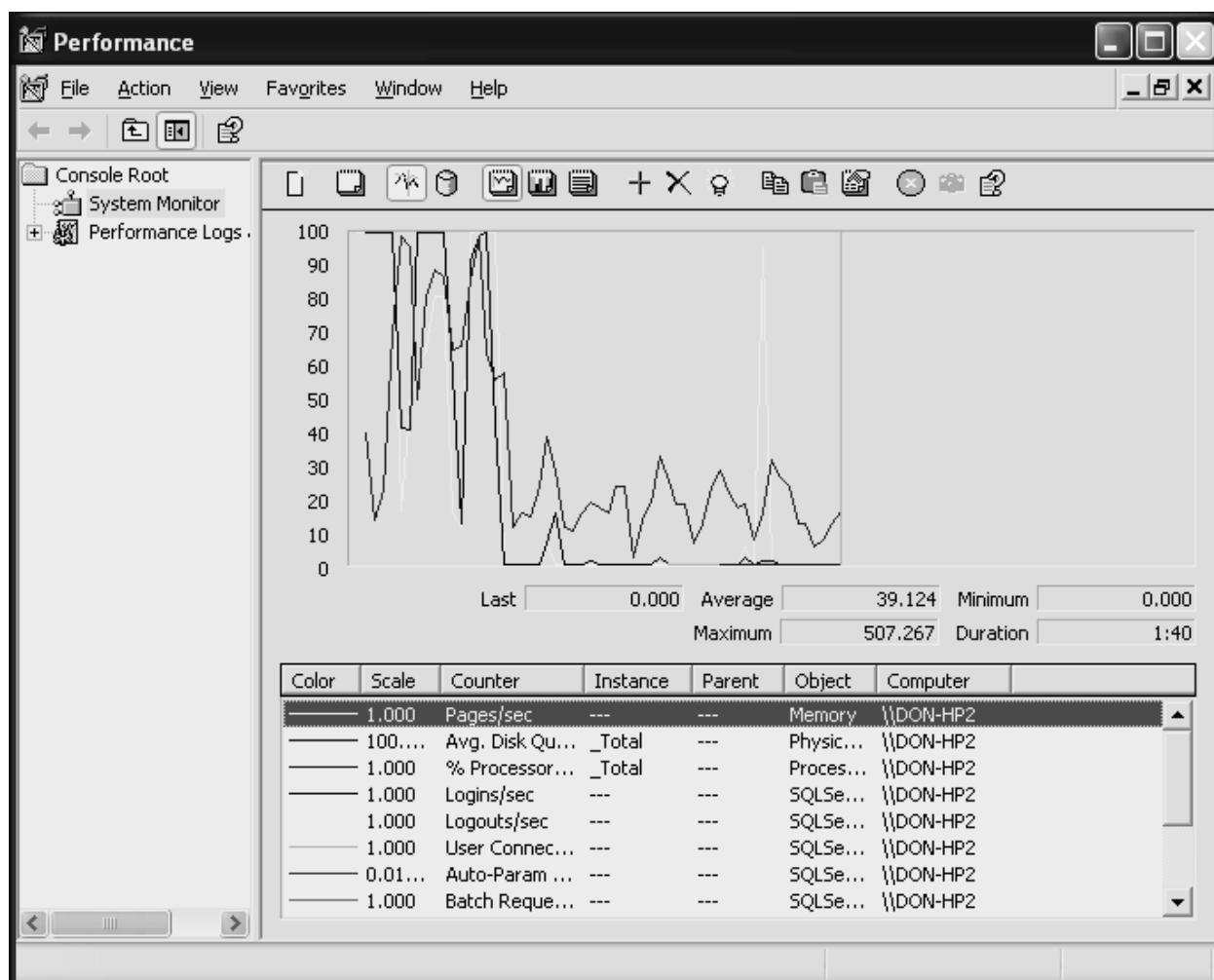
Figure 1.3: System Monitor lets you monitor several performance statistics in a single window.

System Monitor is great for what it is—a free performance-monitoring tool. Unfortunately, if you have more than one server or even if you simply want to regularly perform monitoring and document the results, System Monitor is inadequate. Fortunately, several third-party tools exist to make performance monitoring more manageable, especially in larger environments. The key is to select a tool that supports methodical performance monitoring. That means finding a tool that lets you easily document core performance factors—something System Monitor does *not* do. You will definitely want to consider tools that add a little bit of intelligence to the monitoring process. For example, rather than spend all day sitting and watching System Monitor, which few administrators have the time to do, look for a solution that can automatically measure server performance and automatically translate that data into health information, letting you know if your servers are in good health, marginal health, or poor health.

I'll cover System Monitor in more detail in Chapter 2. I'll also introduce you to some of the available third-party monitoring tools so that you can get an idea of what they can do for you.

VERITAS™

Another valuable performance tool is SQL Server Profiler, which is included with SQL Server. Profiler allows you to see every little action that SQL Server's engine performs, including query processing, user logins, and so forth. In addition to showing you the actions, Profiler allows you to see how long each action takes. For queries, you can examine the execution plan that SQL Server's query optimizer came up with, and you can see exactly how long queries are taking to execute on your servers.

What to Measure

SQL Server exposes a *lot* of performance statistics. System Monitor includes more than a dozen performance objects for replication, query execution, user connections, and so forth, and each of those objects includes anywhere from four to a couple of dozen individual measurement points. In addition to these objects, you need to be concerned about basic server performance factors, such as process and memory utilization, disk and network throughput, swapfile performance, and so forth. You also need to have a firm opinion of what you consider "good" and "bad" performance (see the sidebar "Know They Service Levels").

So what do you measure? There are certain performance factors that directly indicate server health, such as individual processor utilization, overall disk throughput, and network throughput. (Memory utilization can be important, but only in context.) SQL Server, Enterprise Edition, for example, can allocate all of a server's memory without actually using it, so watching the server's overall memory utilization isn't actually very educational.

Within SQL Server, you'll want to monitor overall user connections. Doing so will give you some basic idea of the server's approximate workload context. In other words, if the server is sweating it with only three users connected, those users are obviously doing something aberrant to the server. If, however, the server is running on the redline with 6000 users connected, you'll have a pretty good idea why—too many fingers in the pie.

Know Thy Service Levels

Before you start worrying about what, specifically, to measure in SQL Server, you need to know what you care about from a business perspective. Usually, that means defining *service levels* that specify in layman's terms how your servers will perform. Statements like "Average queries will execute in .02 seconds" or "This particular query will execute in 2 seconds" are good examples of what goes into a service-level document.

Without service levels, you're shooting in the dark. Sure, it's nice to know that processor performance is less than 50 percent at all times, but it doesn't make your users feel any better if their response times are still slow. Service-level statements like "Application X will take Y seconds to call up screen Z" take into account everything that can affect performance, from query execution and memory utilization on SQL Server to network availability and client response times. Your service levels help define "healthy" and "unhealthy" conditions for your servers. Once defined, you can then apply the appropriate performance measures to determine whether your servers are meeting your service levels.

I'm not going to reel off a list of everything you should monitor in SQL Server—the list would be huge and wouldn't necessarily apply to your specific environment. Instead, throughout this book, I'll describe what you should measure to determine factors such as query execution speed, cache memory performance, and so forth.

Documenting Your Measurements

From a methodology standpoint, documentation is almost more important than what you're actually measuring. Whenever you take performance measurements, you need to document what the final measurements were and under what circumstances the measurements were taken. For example, I recommend four phases of performance measurements:

1. Take your first set of measurements—your *baseline*—before your servers go into active production. These measurements will give you an "at rest" set of readings.
2. Take your next set of measurements in production, when the server seems to be operating in an acceptable fashion. These measurements are your second baseline and represent a typical acceptable performance level.
3. Continue to document additional measurements on a periodic basis (perhaps monthly). This practice is referred to as *trending* and will allow you to perform a *trend analysis*. Basically, trend analysis is just noticing the small month-to-month changes in performance, then extrapolating those changes to determine when the server's performance will become unacceptable. For example, suppose that processor utilization increases by 1 percent each month. By extrapolating, you'll be able to determine when processor utilization will reach unacceptable levels and take steps to *prevent* that from happening. Figure 1.4 shows a sample graph that includes baseline, trend, and extrapolated performance.
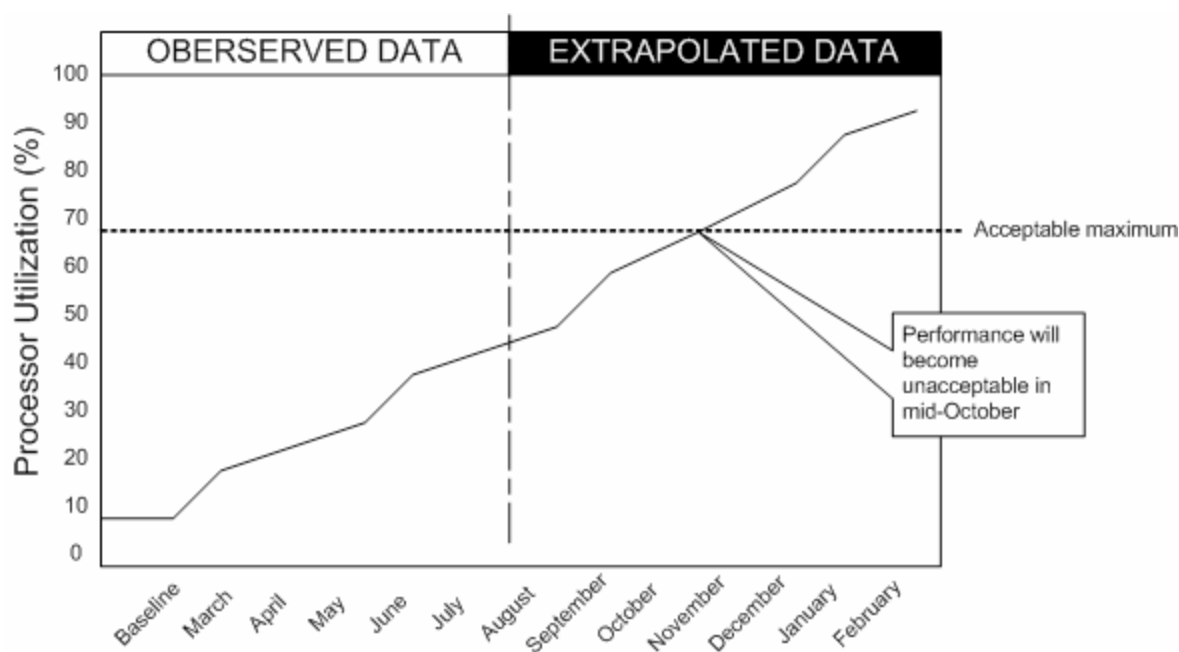
Figure 1.4: Representing performance information on a graph is often the easiest way to determine when performance will fall below acceptable minimums.

4. On an as-needed basis, take additional measurements. You'll usually do so only when server performance changes suddenly for no apparent reason or just after you've implemented important changes. Compare these measurements to your baselines to determine whether the changes you made resulted in an improvement or to spot exactly which performance factors are causing the server's overall performance changes.

VERITAS™

How should you document your performance measurements? The answer depends largely on the tool you're using. If you're relying on the freebie tools included with Windows, you'll need to haul out Excel and punch performance numbers into a spreadsheet by hand. Many third-party tools provide automated documentation and comparison capabilities, which makes using a performance-management methodology easier.

📖 I'll cover third-party performance tools in more detail in Chapter 2.

Types of Databases

Different types of databases are designed to perform various functions, and understanding the performance differences will help you take the best advantage of SQL Server's capabilities. The two main types of databases are online transaction processing (OLTP) and data warehouses.

OLTP Databases

OLTP databases are the typical databases that you probably work with the most. They're designed to be queried and updated on a fairly regular basis. Nearly every database-supported application uses a database that can be classified as OLTP: sales databases, customer databases, ordering databases, you name it.

OLTP databases are characterized by their use of *normalization.* Normalization seeks to reduce data redundancy by breaking up data into logical entities and creating one database table for each entity. For example, in an order-processing database, you might have one table for orders, another for order line items, another for customers, another for addresses (because customers can have multiple addresses), and so forth. Normalization can speed data updates because data (such as a customer's name) is only written once. From then on, that same data is referred to by an ID number or some other unique piece of information. So when one customer places 100 orders, that customer's customer information has to be saved to the database only one time.

Well-designed OLTP databases often have a deliberate amount of *denormalization.* Sometimes, the denormalization is to meet specific business needs. For example, in an order-processing database, you would store a sale price for each order line item. This information is technically duplicated information because items' prices are also stored in the items' own table along with their item names and descriptions. But a sales price must be fixed to a point in time. In other words, just because an item's price changes next month doesn't mean that all previous sales for that item must also have their prices adjusted. So, for that fixed-in-time business reason, the price information must be denormalized and stored in both tables. Figure 1.5 shows a diagram of the example database, including the relationships between the tables.
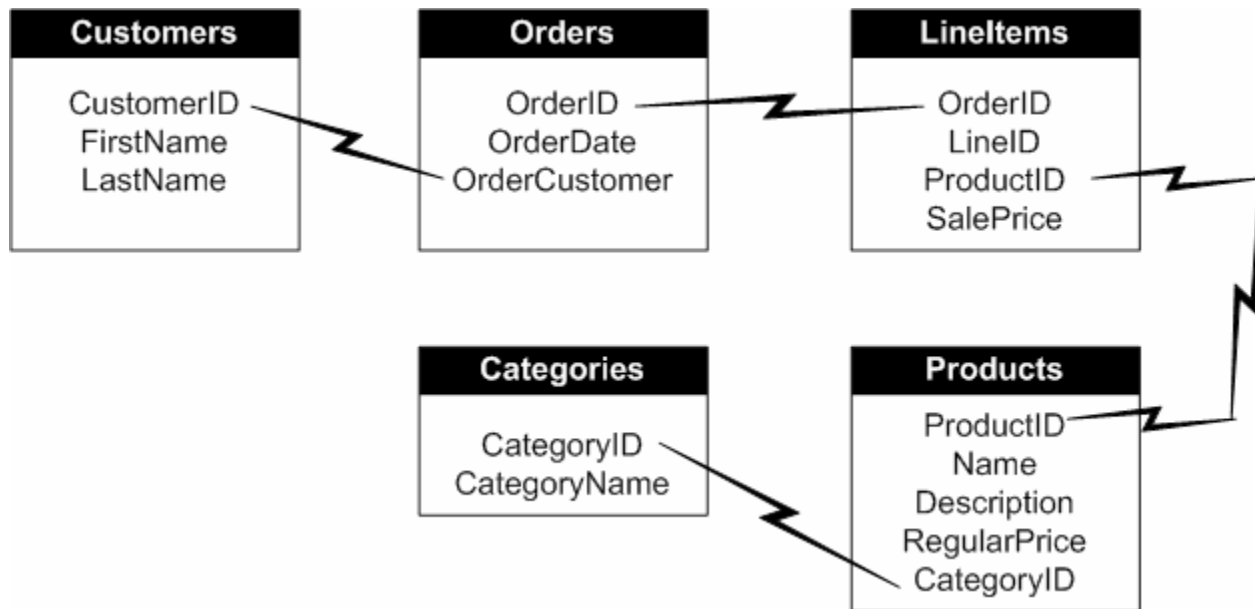
Figure 1.5: An example order-processing database. Note how the various tables refer to each others' ID numbers rather than duplicating information.

OLTP databases don't always offer the best query performance. For example, suppose you wanted to query your order-processing system for the description of every item ordered by a particular customer over the past year, only for items in a particular category. You'd need to write a query that joined several tables: customers, orders, order line items, product categories, and products. Those joins take SQL Server some time to process, and your search criteria would fall across those tables. The orders table would include the order date, allowing you to get all orders in the past year, and the categories table would provide the filter for items in a specific category. Although SQL Server is designed to handle that kind of query, doing so takes more time than simpler, single-table queries. What normalization gives you is faster inserting and updating of data. By normalizing data, you're not repeating it throughout the database. So when it comes time to add data or change it, you don't have to do as much work.

Data Warehouses

A data warehouse is essentially a completely denormalized database stored in a single table. (It's actually a lot more complex than that, but I'll get into the details later.) So for the example order-processing database, your database might include a single row for every item ever purchased by a customer. Each row would include the customer's information, order information, category names, order line item details, and product details. The result is a database that takes up a *lot* more space than its OLTP cousin, but can be queried from a single table without joins—in other words, *very* quickly.

Data warehouses aren't used to run line-of-business applications. Instead, regular OLTP databases are used as a data source to periodically *load*, or populate, a data warehouse. So although your company continues to cruise along on its OLTP order-processing system, an application could query the data from the OLTP database and denormalize it into a data warehouse. The data warehouse could then be used to very quickly generate large, complex reports. Figure 1.6 shows how the various tables of an OLTP database can be loaded into a single, denormalized data warehouse table.
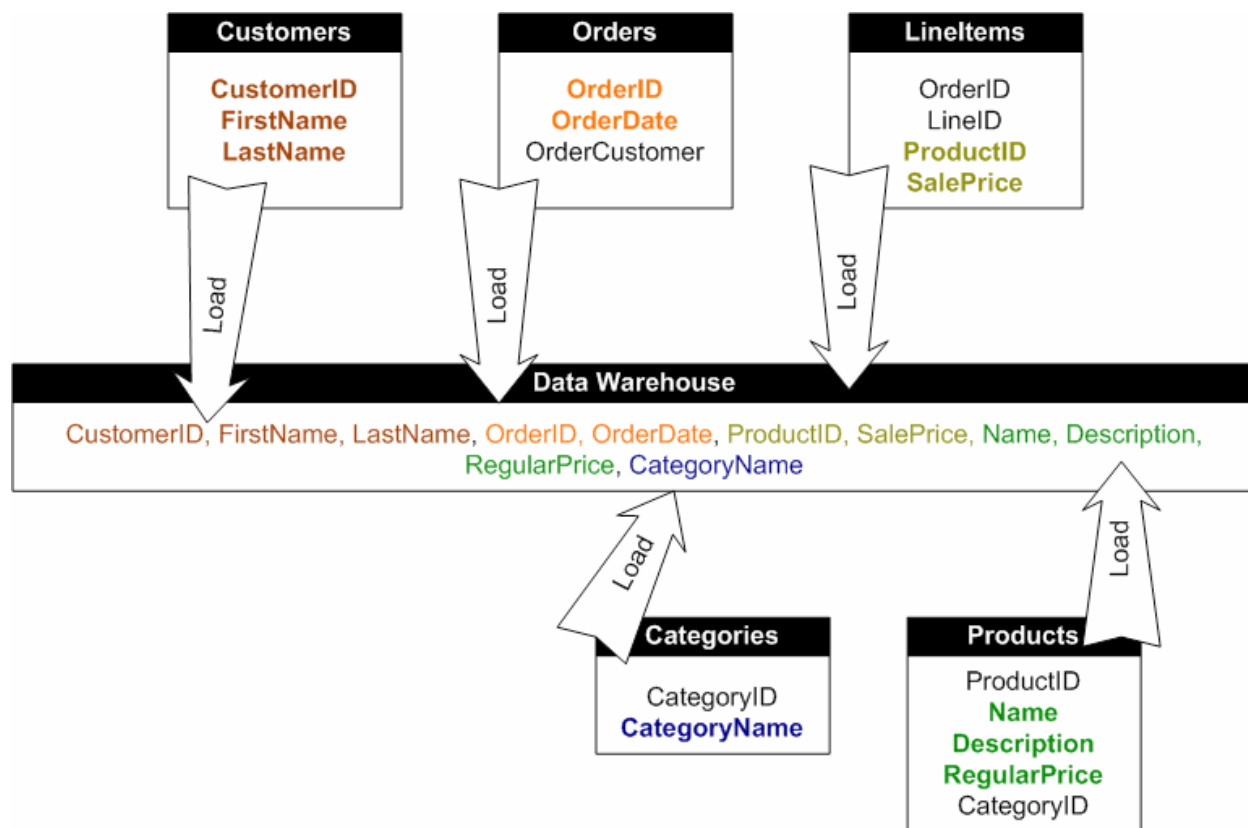
Figure 1.6: Data warehouses are often loaded in the evenings or on weekends, making them available for reporting purposes during business hours.

Scaling Concepts

Scalability is a measure of a solution's ability to easily expand to accommodate additional workload. In other words, scalability is simply how difficult it is to make something perform more work. In the case of SQL Server, for example, scalability tells you how difficult it is to get SQL Server to support additional users, host an additional database, or run a larger number of complex queries.

There are two methods for *scaling*, or growing, a solution. The first method is the one we tend to think of first—buy bigger hardware. This technique is called *scaling up,* because it simply means upgrading your existing hardware to support more capacity. SQL Server works well in scale-up situations, but scaling up has an upper limit. Most servers are built to accommodate only a certain number of processors, a certain amount of memory, and so forth. After you've scaled up as far as you can go, you're stuck. Even if you're willing to buy brand-new hardware from scratch, scaling up will only take you so far because you can only buy so big of a server at any given time.

The other method of scaling is *scaling out* (adding more servers to the solution). Each new server performs the same tasks as the original server. Web farms are the classic example of scale-out architecture, in which you simply keep adding Web servers (all hosting the same content) until you've got enough of them to handle the number of users attempting to reach your Web site. Contrary to popular belief, SQL Server *can* scale out, although doing so is definitely much more difficult than scaling out a Web farm. In SQL Server, scaling out means using federated databases, replication, linked servers, and other techniques that allow multiple SQL Server computers to look like a single server, even if they're not hosting identical copies of the database.

> 📖 Chapter 6 focuses on scaling SQL Server and includes an explanation of federated databases, distributed databases, replication, linked servers, and much more.

Scaling can sometimes be the easiest solution to performance problems. If your current SQL Server system can't handle your workload, either upgrade it (scale up) or add another one (scale out) to help handle the load. Scaling doesn't require any fancy query tuning, complex performance-troubleshooting procedures, or any of that stuff. Unfortunately, scaling—especially with SQL Server—is almost always expensive because it involves buying more server hardware. In the case of SQL Server, scaling out can also mean a lot of additional administrative overhead, as you're adding more SQL Server computers that you'll have to maintain and troubleshoot. So although scaling is often the *easiest* solution to performance problems, it isn't always the *best* solution. The best solution is often to make more efficient use of the hardware you've got, which means digging in and fine-tuning everything. Fortunately, that's what the rest of this book is all about!

Holistic Performance

Always remember to look at the big picture when it comes to performance. Many administrators fall into what I call the performance trap—looking for the one little detail that, once fixed, will noticeably improve performance every time. Other folks call it the magic bullet, which, when fired, will kill your performance problems. Unfortunately, there is no such thing. Modern servers are *systems,* which means they consist of several interlocking subsystems that all work together. Changing one subsystem might simply shift a performance problem to another subsystem. SQL Server is a system, too, with memory caches, a query optimizer, database statistics, and much more. SQL Server interacts closely with both the Windows OS and your server hardware, so changing any particular performance component is bound to have an effect somewhere else.

Thus, you can't focus on individual performance counters in System Monitor. Instead, you have to look at the *whole* performance picture, which is why it's called *holistic performance.* In the next few sections, I'll talk about the various components that can, often invisibly, contribute to performance problems in a SQL Server computer.

> 📖 I'll expand on many of these concepts in Chapter 3, in which I'll show you how to pinpoint performance bottlenecks and resolve them.

Server Hardware

Server hardware is the basis for everything that runs on the server. You're probably familiar with some of the basics of computer hardware performance, such as having lots of RAM and a fast processor. But servers have different processing needs than desktop computers; thus, server hardware must be optimized slightly differently. Here are some things to look for in a high-powered server:

- Server-optimized processors (such as Intel's Pentium Xeon series)—Unlike desktop-optimized processors, server-optimized processors focus less on multimedia acceleration and more on parallel instruction processing and the symmetric multiprocessing upon which Windows is based.

- Processors with a fast, large level-two (L2) cache have quicker access to memory and waste less time waiting for data to be transferred from memory. Even if you're only saving a thousandth of a second, thousandths can add up very quickly.

- Almost all servers use memory caching to speed memory access, but there are different types of cache architecture. A look-aside architecture puts memory requests out to main memory and the cache at the same time, allowing either to respond quickly. If the requested memory is present in the cache, it will be faster than main memory as a result of the type of memory from which caches are built. Look-through caches require the processor to place the request with the cache, which must then access main memory independently if the requested data isn't in the cache. Although somewhat slower than look-aside caches, look-through caches prevent unnecessary requests from reaching main memory, leaving the server's memory bus free to service requests from bus-mastering devices. Both architectures have advantages, but SQL Server computers, which usually rely a bit less on bus-mastering devices, tend to benefit from look-aside caches.

- Servers should include error-correcting memory, which allows the server's memory controllers to automatically correct for multibit errors without having to waste processor cycles in requerying memory. Simple parity memory—found in most desktop computers——will tell the controller that an error is present but still requires the controller to requery the memory to retrieve the correct data.

- High-speed disk controllers with onboard, battery–backed–up caches can vastly improve server performance. These controllers accept data from the OS as fast as the PCI bus architecture will allow, letting the OS (and therefore SQL Server) quickly move on to other tasks. The controller then writes the data to disk as fast as the disks will allow. The controller's onboard battery ensures that no data is lost if the power fails before the cache is emptied.

☞ When Windows shuts down normally, it instructs all disk controllers to flush their caches to disk and waits for them to finish. When you perform a *dirty* shutdown of Windows by pulling the plug or simply pressing the power button, Windows doesn't get the opportunity to flush the disk controllers. Keep in mind that SQL Server checkpoints transactions in the transaction log as soon as Windows informs SQL Server that the data was written to disk. Because the data might not, in fact, be on disk, but might simply be in the controller's cache, SQL Server's recovery features won't help you if the controller shutdown is dirty and fails to retain the cache information.

Controllers with a battery backup will usually retain their cache contents for a few hours or more, and flush the cache to disk when power is reapplied to the server. This feature is a great failsafe in the event of a power loss, but you shouldn't rely on the battery backup to save your data every time you perform a dirty shutdown of the server. Make it a practice—especially with SQL Server computers—to cleanly shut down the OS, allowing the disk controllers to finish writing data before power is removed.

📖 The interactions between a server's various busses, its memory and processors, the network adapters you install, and SQL Server are complex. Chapters 2, 3, and 6 will include more detailed information about eliminating hardware-related bottlenecks and configuring SQL Server to use its hardware more efficiently.

Software Settings

You can degrade SQL Server's performance through seemingly innocent actions. For example, each SQL Server database has a configuration option called *Auto Close.* Usually, SQL Server keeps database files open all the time (which is why you can't use a normal file backup utility to back up database files). With Auto Close turned on, however, SQL Server closes the database's disk files when the last user connected to the database disconnects. When the next user tries to access the database, SQL Server automatically reopens the file. This process imposes quite a lot of overhead on SQL Server. In addition, on a database that's accessed regularly by only a few users, Auto Close can cause significantly slower performance.

Other types of software settings can have a negative impact on performance over time. For example, one of the configuration options on an index is its *fill factor.* Like database rows, indexes are stored on 8KB pages. When you create a new index, you can specify a fill factor in terms of percentage: 50 leaves half of every index page (about 4KB) empty. Why would you do so? Take your local phone book as an example. It's an index created with a 100 percent fill factor—no empty space on any page. When someone new moves into the neighborhood, you can't enter his or her name directly on the proper page. Instead, you have to flip to a blank page in the back of the book and write in the new person's name. You also have to make a notation on the correct page that more information is located at the back of the book. What you now have is a *split page,* or *fragmented index,* on which the information is no longer in the desired order (in this case, alphabetical). Using fragmented indexes is much slower, which is why SQL Server allows you to specify a fill factor. If the phone book had a fill factor of 80, each page would be 20 percent blank, allowing plenty of room to write in new neighbors' information without splitting the page and turning to the back of the book. Of course, the index would be much larger as a result of all the free space.

📖 I'll include indexes and their settings in my query performance-tuning information in Chapters 4 and 5.

You might find that certain so-called "features" actually diminish performance. For example, I worked with one company that decided to turn off SQL Server's automatic database statistics updating. The company created an automated task to update statistics late at night once a week (which reflected the number of changes they made to the database each day). The company found that some strange, middle-of-the-day performance issues went away—issues that might have been caused by SQL Server taking a quick break to update database statistics in the middle of the day. Another client found that SQL Server was coming up with some very poor query execution plans on its 8-processor server. Disabling SQL Server's parallelism feature resulted in the faster execution plans the client had expected, suggesting that SQL Server's query optimizer was underestimating the cost of using parallelism when it considered different execution plans.

Design Issues

Database design can kill SQL Server's performance. For example, I one worked with a real estate company that stored, among other things, street address information. Addresses are obviously important to a real estate company, because addresses represent the company's products (homes and property for sale). The company created a database that used no less than nine tables to represent a single address, breaking items such as the street type (road, street, avenue, and so on) into one table, the street's direction (N, NW, and so on) into another table, and so forth. The client's goal was to ensure that users could only select from the options included in the tables, which improved the quality of the data in the database.

Thus, the idea is a good one. You certainly can improve data quality by providing users with drop-down lists for items such as road, street, avenue, and so forth. Allowing users to simply manually type in that information will inevitably result in errors such as the typo raod for road or aevenue for avenue. In addition, pulling the contents of those drop-down lists from a database table makes it easy to add new information when necessary. However, all that information should be denormalized into a single table for the final address. That way, SQL Server can query address information from one table, instead of having to join nine tables.

☞ How many tables is too many? In general, if you're joining seven tables on a regular basis to produce query results, you need to think about denormalizing your database if you want to improve query performance. Your goal should be to have SQL Server join as few tables as possible; only use normalization where it makes sense. Don't normalize your databases to the fifth normal form just for the sake of having done so; make sure there's a valid business reason for doing so.

It's perfectly acceptable to keep the denormalized information in different columns. For example, keeping road in a different column from the street number and name will make it easier to update the row in the future. The only time it makes sense to keep information normalized is when it might change and you don't want to have to make the same change in a million different places. For example, in an order-processing database, you wouldn't copy the customer's name into every order, because the customer's name might change (if the customer gets married, for example). Leaving the name normalized into a separate table allows all the customer's orders to remain linked to that customer, even if a name change occurs in the future. Normalization also prevents you from having to change every one of the customer's past orders when a name change does occur. However, when information isn't subject to change, you can improve SQL Server's performance by denormalizing the data and eliminating performance-impacting table joins. Keep your table of acceptable values (called a *lookup table*) because it improves data quality, but don't force SQL Server to join to the lookup table just to obtain that kind of information.

Client Considerations

Database applications can cause plenty of performance problems that have almost nothing to do with SQL Server. Here are some of the most common worst practices that client applications exhibit:

- Older data access libraries, such as Microsoft Data Access Objects (DAO) or Open Database Connectivity (ODBC) don't offer the fast performance of newer data access libraries such as ActiveX Data Objects (ADO) and Object Linking and Embedding Database (OLE DB). Make sure that your client applications are using the latest available access libraries, data providers, and other drivers.

- Client applications are sometimes designed to query more information than they need. For example, it's not uncommon for client applications to query hundreds of rows of data, then search through those rows to find a few rows that they actually need to work with. It may be quicker to have the client specify a more precise query and send only the data that the client really needs.

- Building a two-tier (client/server) application to serve thousands of users doesn't make the best use of SQL Server's resources. After all, most clients don't need a continuous connection to SQL Server, which requires SQL Server to set aside a small portion of memory for each open connection regardless of whether that connection is doing any work. A better architecture is multi-tier, in which clients contact a middle-tier server to request data. The middle-tier server can open multiple connections with SQL Server and use a free connection to obtain data on behalf of clients. This technique is called *connection pooling* and is a very effective way to allow a large number of clients to access a single SQL Server computer without a negative performance impact.

- Make sure you (or your database developers) are familiar with the quirks of your data access method. For example, when ADO receives a query like
  ```
  SELECT * FROM Customers
  ```
  ADO and SQL Server actually have to expand the "*" into the list of columns in the table. Even if you want all the table's columns, simply specifying them, as in
  ```
  SELECT FirstName,LastName FROM Customers
  ```
  will help ADO and SQL Server execute the query somewhat more quickly. Knowing about the little quirks of your data access method will help you avoid minor performance hits in your database applications.

- In general, SQL Server can execute stored procedures much more quickly than ad-hoc queries because SQL Server retains stored procedures' execution plans, meaning they don't have to be run through the query optimizer each time they execute. Client applications should interact with SQL Server through stored procedures to the greatest degree possible.

Obviously, these are just a few of the many performance-related considerations you can check out in your client applications.

> 📖 Chapter 7 is devoted entirely to improving SQL Server's performance by creating client applications that utilize SQL Server more efficiently.

VERITAS™

Summary

In this chapter, I've given you an introduction to SQL Server performance and health. You should have a solid understanding of how SQL Server's architecture lends itself to performance, and you should understand the difference between observing server performance and measuring a server's actual health. You should also have a good start on understanding the variety of issues that can contribute to poor server performance.

In the next seven chapters, I'll focus on individual performance issues in more detail. Chapter 2 will focus entirely on measuring and monitoring performance with a variety of tools. Chapter 3 will focus on core server performance, including the most commonly used performance measurement points, and tips about finding and resolving specific types of bottlenecks related to hardware, software, and database design. I'll also expand on my performance monitoring methodology and give you some step-by-step information for creating your own. In Chapters 4 and 5, I'll focus on query optimization, which is definitely a big area to cover. Chapter 6 will cover SQL Server's scalability, including scale-out and scale-up techniques that you can put to use in your environment. Chapter 7 will deal primarily with client-side performance issues. Finally, Chapter 8 will sort of wrap everything up with tips for getting the best performance "bang for your buck." I'll cover the most common performance problems, and show you how to make performance-related changes that have the most noticeable impact on your SQL Server operations.

Welcome aboard! I think you'll find that SQL Server performance can be a fun puzzle to solve, and getting "under the hood," so to speak, can reveal a lot of new insights about how SQL Server and your database applications work together.

VERITAS™

Chapter 2: Measuring Performance

Measuring SQL Server performance isn't always easy. For starters, SQL Server is such a complex product that it's often difficult to figure out where to start. If a particular query is running slowly, do you start with System Monitor? Query Analyzer? SQL Profiler? Of course, knowing *when* to measure performance is just as important as knowing *what* to measure, because without baselines and trends, you won't truly be able to get a good handle on your servers' performance issues. In this chapter, I'll introduce you to some of the tools of the SQL Server performance trade, and show you how to use them. I'll also include some third-party tools that help fill the functionality gaps of the tools that Microsoft provides. Then I'll explain when to measure performance and how to conduct trending and trend analysis—two important techniques in SQL Server performance monitoring and tuning.

System (Performance) Monitor

The Windows 2000 (Win2K) System Monitor—called Performance Monitor in Windows NT, and available in Win2K's Performance Console—is most administrators' first step in performance management. System Monitor consists of a number of *performance objects,* which each represent some major operating system (OS) or application subsystem. For example, Win2K performance objects include processor, memory, and so forth. Each performance object consists of one or more *performance counters,* which each represent a specific measurable component of the performance object. The Memory performance object, for example, includes counters for the total amount of memory in use, the amount of virtual memory in use, and so forth.

SQL Server Performance Objects

System Monitor's greatest strength is its extensibility. Almost every Microsoft .NET Enterprise Server—including SQL Server—and many third-party applications add performance objects to System Monitor, allowing it to monitor the performance for those products. SQL Server includes a rather large number of performance objects, reflecting its complexity, and each object includes several counters:

- SQLServer:Access Methods—This object measures performance from a variety of SQL Server database access methods. Counters include Full Scans/sec, which indicates the number of full table scans SQL Server performs in a second; Index Searches/sec, which indicates the number of times SQL Server uses an index in a second; and so forth.

- SQLServer:Backup Device—This object includes a single counter, which lets you monitor the throughput (in bytes per second) to a device that is being used to back up a SQL Server database or transaction log.

- SQLServer:Buffer Manager—This object allows you to monitor the effectiveness of SQL Server's buffer manager, including the number of buffer lookups per second, buffer cache hit ratio, free buffer pages, and so forth. The purpose of these buffers is to retain commonly used information in RAM, reducing the need for SQL Server to read that data from the slower disk subsystem.

- SQLServer:Buffer Partition—This object provides access to three counters, which allows you to monitor the free list requests and pages for SQL Server's buffer partition.

- SQLServer:Cache Manager—This object allows you to monitor SQL Server's Cache Manager performance, including cache hit ratio, number of pages in the cache, and so forth.

- SQLServer:Databases—This object provides access to several database-specific counters, including the number of active transactions, bulk copy throughput, data file size, and so forth. This object includes one instance of itself for each database in SQL Server. Each instance of the object monitors a specific database, letting you monitor, for example, a specific database's active transactions. You can also monitor a special _Total instance, which aggregates all other instances into a single set of server-wide statistics.

- SQLServer:General Statistics—This object includes three counters for miscellaneous statistics, including logons per second, logouts per second, and number of active user connections.

- SQLServer:Latches—This object includes three counters that provide access to SQL Server latch requests, including average latch wait time, latch waits, and total latch wait time. Latches are used to synchronize objects for a very short period of time. For example, when SQL Server reads a row of data from a table, it latches the row, ensuring that all columns in the row remain the same until the read operation is complete. Unlike locks, latches are not meant to exist for more than the life of a transaction or, in fact, for more than a split second.

- SQLServer:Locks—This object provides insight into the number of locks SQL Server is managing, including the average (and total) lock wait times, number of deadlocks, and so forth.

- SQLServer:Memory Manager—This object, perhaps one of the most important for SQL Server, allows you to monitor SQL Server's use of memory, including the amount of memory allocated to connections, locks, the query optimizer, and so forth.

- SQLServer:Replication Agents—This object includes a single counter, which shows how many SQL Server replication agents are currently running.

- SQLServer:Replication Dist—This object includes three counters, which show the number of Replication Distributor commands and transactions that are occurring.

- SQLServer:Replication Logreader—This object includes three counters, which show the number of Replication Log Reader commands and transactions that are occurring.

- SQLServer:Replication Merge—This object includes three counters, which show how many merge replication changes are being uploaded and downloaded and how many merge conflicts are occurring.

- SQLServer:Replication Snapshot—This object includes two counters, which show how many snapshot replication commands and transactions are occurring.

- SQLServer:SQL Statistics—This object is sort of the general-purpose bucket for performance counters, including SQL compilations and re-compilations, auto-parameters statistics, and so forth.

- SQLServer:User Settable—This object provides access to as many as 10 instances of a performance counter named Query. The Query performance counter allows developers to insert performance-tracking counters into SQL Server queries and batches.

- SQLXML—This performance object includes only one counter: Queue Size. The counter indicates the number of requests in SQL Server's XML processing queue. This object (and counter) is only available when you install one of Microsoft's Web Releases for SQL Server 2000, which add XML features to SQL Server's basic feature set.

> ✎ Performance objects are present for each SQL Server instance. Keep in mind that SQL Server 2000 allows you to run multiple instances on a single server. In other words, one server computer can act as multiple SQL Server computers. Each instance of SQL Server has its own complete set of performance objects. The default instance of SQL Server—the instance that doesn't have a specific name—uses the performance object names previously listed. Other instances add their instance name to the performance object name. For example, an instance named SECOND would have a performance object named SQLServer$SECOND:SQL Statistics.
>
> When you fire up System Monitor, make sure you're selecting the objects that go with the SQL Server instance you intend to monitor. Otherwise, you'll spend time watching a completely different instance's statistics, which won't be any help at all!

For now, don't worry too much about how to use each of these performance objects and their counters. I'll refer to specific objects and counters throughout the rest of this book as needed, and you can always refer to the SQL Server Books Online for detailed explanations of what every counter measures. At this point, you just need to have a good feeling for the variety of performance objects that are available and have a rough idea of what each object allows you to measure.

Using System Monitor

System Monitor is fairly easy to use. Simply launch the Performance console from your computer's Administrative Tools folder (located on the Start menu). You'll start with a blank graph, to which you can add counters. To add counters, click the + icon in the toolbar. As Figure 2.1 shows, you can select the performance object, then the counter, that you want to monitor. Click Add to add the counter to the graph.

> ✎ Don't see the SQL Server performance objects? If you're running System Monitor on your workstation, you'll need to install the SQL Server Client Tools to get the SQL Server performance objects on your workstation. Doing so will allow you to run System Monitor on your workstation to measure the performance counters on a remote SQL Server computer. You do *not* need the full SQL Server product running on your workstation to measure SQL Server performance on *other* computers.
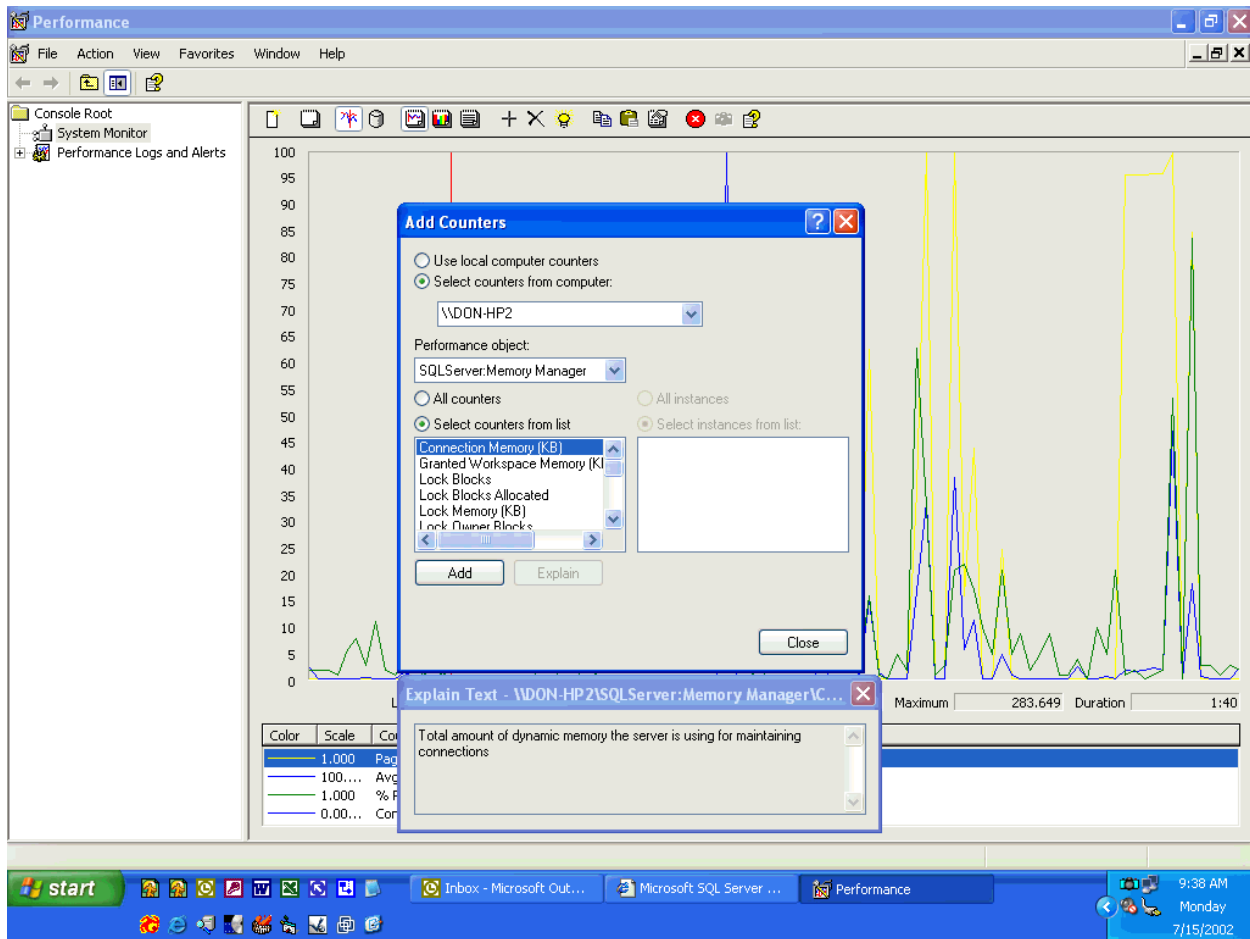
Figure 2.1: You can add as many counters as you need to a single graph.

☞ Can't remember what a counter measures? Highlight a counter and click Explain. System Monitor will present a small window containing a brief explanation of the counter's purpose.

When you add a counter, System Monitor immediately begins tracking its performance values in the chart (which you can see in the background of Figure 2.1). When you're finished adding counters, you can monitor them individually as Figure 2.2 shows. The main screen offers a number of features:

- Press Ctrl + H to turn highlighting on and off. As Figure 2.2 shows, highlighting colors the currently selected counter with a bold black line, making it easier to pick the line out from the others on the chart.

- The status bar along the bottom of the graph shows long-term statistics for the selected counter, including its last value, average value, minimum and maximum values, and the length of time you've been monitoring that counter.

- The counter list shows the *scale* for each counter. For example, with a scale of 1.000 (the default scale for the processor utilization counter), each vertical unit on the graph represents one unit in the counter. For processor utilization, for example, a 100 on the graph means 100 percent utilization.
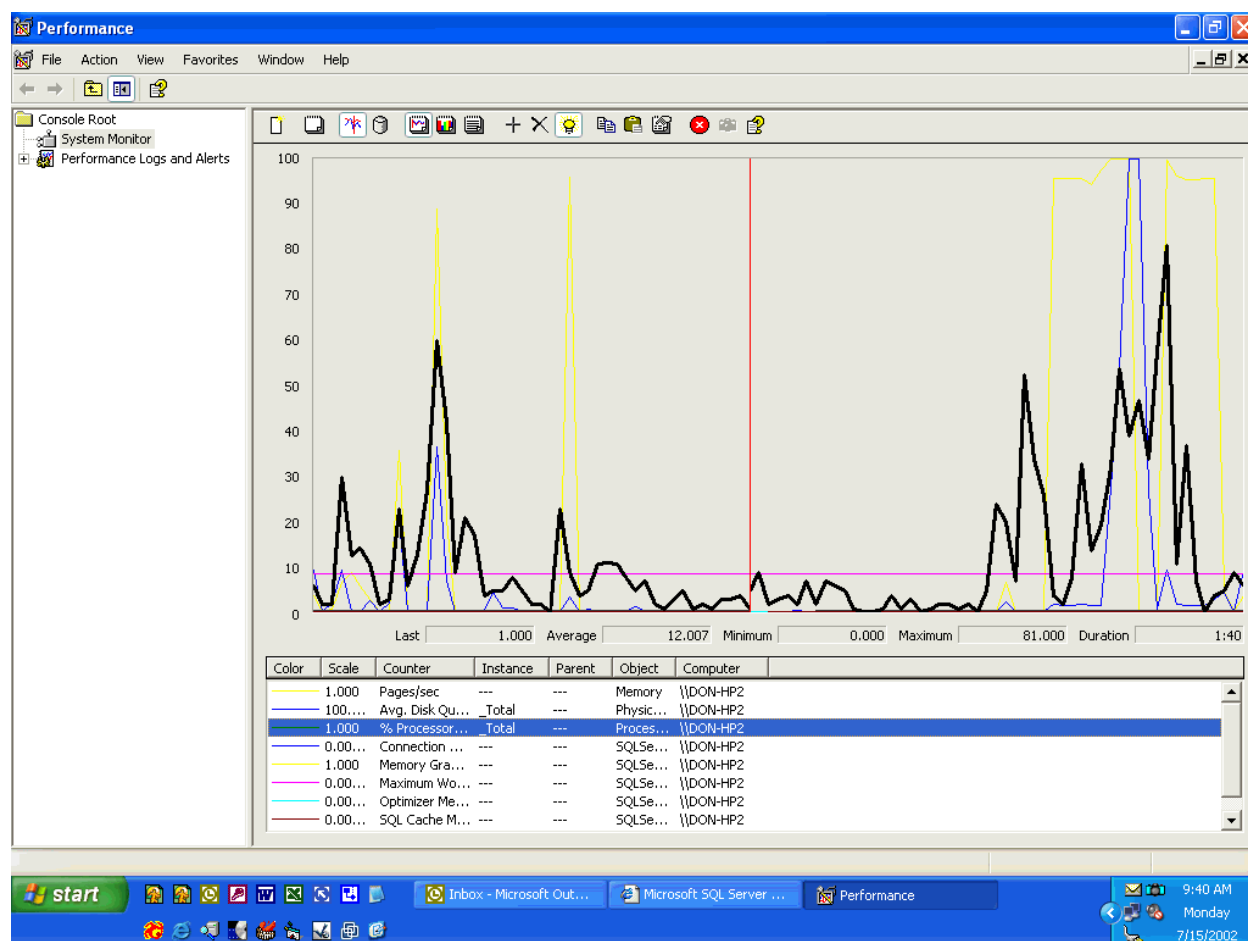
Figure 2.2: Use the highlighting features to easily pick specific performance counters from a cluttered graph.

You can right-click a counter in the list and select Properties from the pop-up list to modify the counter's display properties, including its graph line color, style, and width, and the scale that the counter uses. You can also modify general properties for the graph, including the display of vertical and horizontal gridlines, the vertical scale, graph appearance, and so forth.

☞ Save those counters! If you're like me, you can spend hours coming up with the perfect set of counters to measure a server's performance and customizing the graph display for maximum readability. Don't let that work go to waster when you close System Monitor! Save your counter settings to a file, instead. Then you can double-click that file to reopen System Monitor with your saved settings. I've created several files with specific counter settings: One for monitoring lock statistics, another for general statistics, one for each of the major databases on my servers, and so forth.

VERITAS™

Benefits of System Monitor

System Monitor is designed to view performance in real time or to capture performance in real time into a log file for later review. Despite the large number of SQL Server performance objects, however, System Monitor doesn't provide a very granular look at your servers' performance. For example, seeing the Table Scans/sec counter at a very high rate definitely indicates a problem performance condition, but System Monitor can't show you which statements are causing the table scans, which users are calling those statements, and so forth. Rather than a granular view of performance, System Monitor is designed to provide you with a server-wide view, aggregating the server's entire operations into a single performance chart or log.

That server-wide view makes System Monitor *almost* perfect for checking server health. Unfortunately, because System Monitor isn't designed to accumulate information over a long period of time and categorize performance data into good, warning, and critical health conditions, you'll have to do a lot of work on your own to translate raw performance data into health information.

System Monitor also has some performance limitations of its own. For example, it's not a good idea to run System Monitor on the server you're actually monitoring because the act of running System Monitor will actually impact the server's performance. Few administrators can dedicate a workstation to running System Monitor 24 hours a day, which means they usually only launch System Monitor when there's a problem that they're trying to track down. This behavior illustrates a basic problem with System Monitor for enterprise performance management: The tool is on-demand only. There's no way to launch System Monitor and look back in time to see what was happening on the server 5 minutes ago; System Monitor can show you only what's happening right now. You have to anticipate a performance problem and have System Monitor up and running for the tool to be useful. Third-party tools can significantly improve upon System Monitor, while relying on the same basic underlying technologies. NetIQ's AppManager for SQL Server is designed to monitor multiple servers simultaneously, automatically recording performance information. VERITAS Indepth is also designed to run continuously, watching for different performance problems and identifying their causes. I'll introduce you to both tools later in this chapter.

Query Analyzer

Query Analyzer isn't something most administrators look to for performance information. In fact, many administrators might prefer to stay away from Query Analyzer completely; instead performing as much work as possible in the friendlier graphical user interface (GUI) of SQL Enterprise Manager. Nonetheless, Query Analyzer can be a great tool for checking—and tuning—the performance of specific queries.

Using Query Analyzer

Query Analyzer is basically used to submit T-SQL queries to SQL Server for processing. At its most basic level of operation, Query Analyzer submits the query, waits for SQL Server to execute the query, and displays any results returned by SQL Server. That level of operation might be useful for the crudest of performance measurements ("Why is this query taking so long to complete?") but Query Analyzer offers additional, more advanced features that provide great insight into SQL Server's performance.

To start using Query Analyzer, simply type a query into the main portion of its window. Click the Execute toolbar button (it looks like a VCR's Play button) or press Alt + X (or Ctrl + E) to submit the query to SQL Server. Query Analyzer splits the right side of its window into two panes, with your query displayed in the upper pane and the results displayed in the lower pane. By default, Query Analyzer displays your queries' results, if any, in a grid. You can select additional display features from the Query menu.

☞ Need a handy reference? Earlier versions of Query Analyzer forced you to remember table names, column names, and other information in order to write queries. SQL Server 2000's version of Query Analyzer, however, provides a reference for those object names in the left side of its window, in a convenient tree view.

Even better, Query Analyzer lets you drag objects (such as column names and table names) from the left tree view into the main window. When you do, the objects' names appear within the query you're creating. Now you don't even have to retype long object names!

Perhaps the most useful feature in Query Analyzer is its ability to display a query's *execution plan.* The execution plan is SQL Server's strategy for executing the query, including any indexes, table scans, and other operations that SQL Server will perform to generate the query results you've requested. By displaying the execution plan for a query, Query Analyzer allows you to determine whether SQL Server is taking advantage of indexes, whether it needs to have additional indexes for the most effective execution of the query, and so forth. Figure 2.3 shows a query execution plan, which is displayed as a graphical tree of the operations SQL Server took to complete the query.

📖 I'll cover query execution plans in more detail in Chapters 4 and 5.
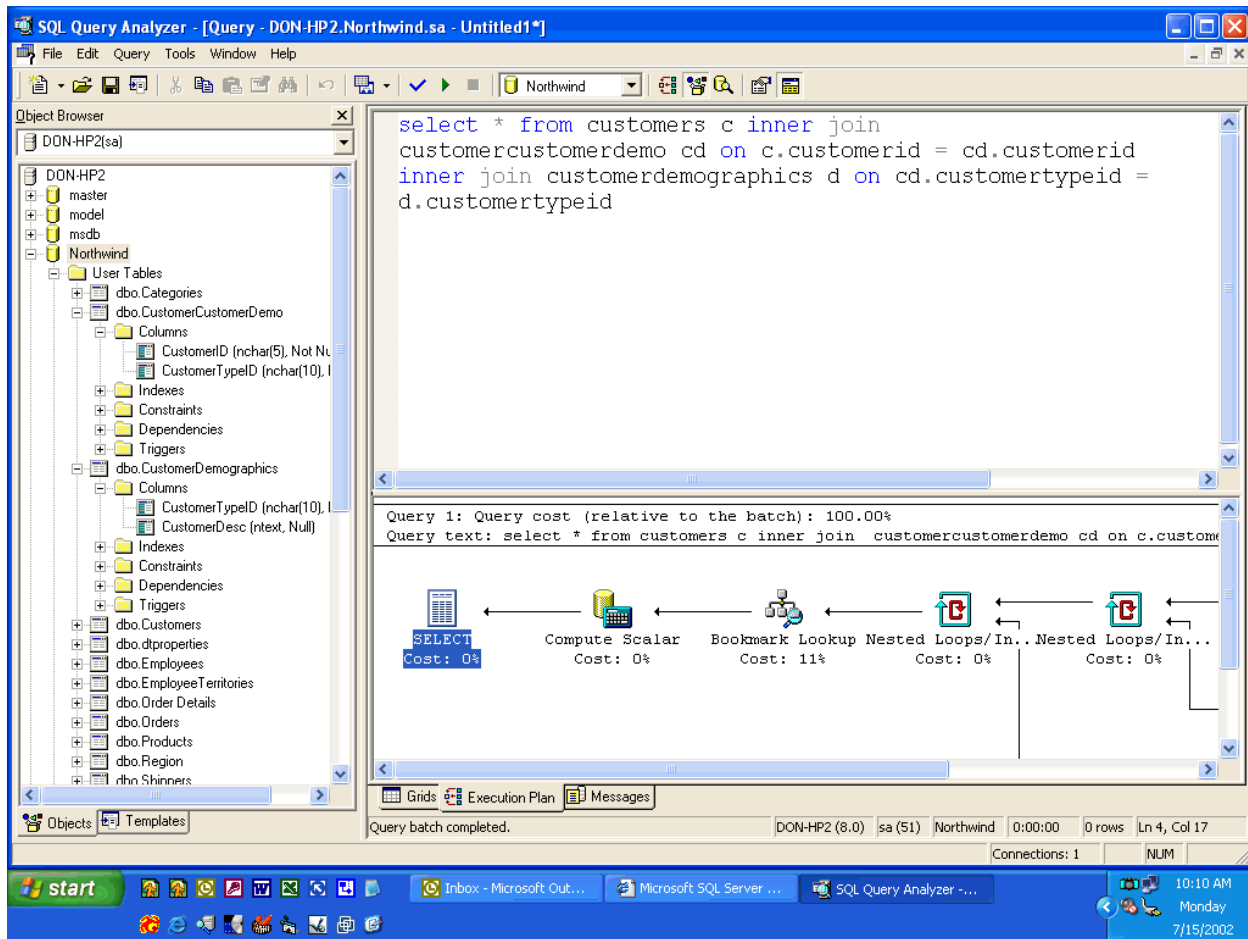
Figure 2.3: The execution plan displays in the lower-right results pane of the Query Analyzer window.

Each step in the execution plan includes an estimate of that step's cost in the overall query. For example, in Figure 2.3, 11 percent of the query's execution time went toward a bookmark lookup. Analyzing these execution plans can help you identify high-cost execution steps, and implement means—such as additional indexes, query hints, and so forth—to help SQL Server execute the query more efficiently.

✎ Query Analyzer also allows you to view an *estimated* execution plan. This feature displays the query plan SQL Server will likely use without actually submitting the query to SQL Server for execution. If you're concerned that a query might be long-running or cause problems, displaying its estimated execution plan can help you evaluate the query without risking your SQL Server.

However, estimated plans can often vary widely from actual plans. One of the most important steps in SQL Server's execution plan generation is an estimation of the server's current workload. A busy server will generate markedly different execution plans than a less-busy server, which will have time to search for more efficient plans. Estimated plans don't take any current server activity into account.

Another Query Analyzer feature lets you see the *trace* associated with your query. A trace is basically a list of each step SQL Server took to complete the query, including the duration (in milliseconds) for each step. Traces are normally associated with SQL Profiler (which I'll discuss later); however, Query Analyzer allows you to view the trace events associated with the query you submitted. Figure 2.4 shows the trace display.
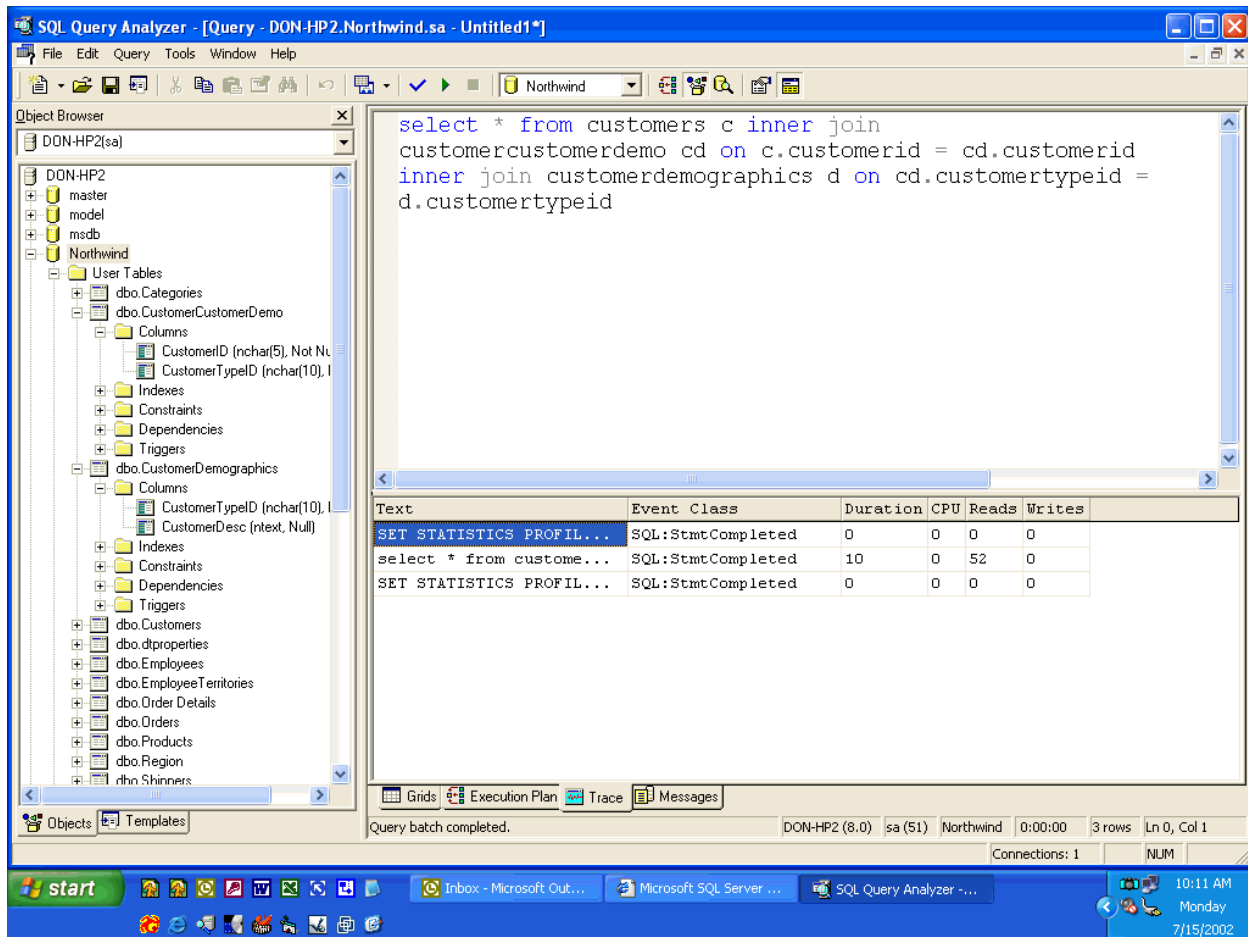
Figure 2.4: Traces are useful for analyzing even duration and CPU cost in more detail.

☞ You can have Query Analyzer display multiple types of results at once. Notice in Figure 2.4 that Query Analyzer is displaying the query's results, its execution plan, and its trace. You can switch between the different result types by using the tabs along the bottom portion of the results pane.

In this example, the query submitted didn't take very long to complete: The time for the StmtCompleted event for the query was just 10 milliseconds, and the query used 0 CPU, an indication that the actual CPU utilization was too low to measure. The query required 52 read operations to pull data from disk (or from cache). The other two operations shown in this trace were automatically submitted by Query Analyzer to force SQL Server to return statistics for the query.

Traces are especially useful when submitting multiple-statement batches to SQL Server because the trace events are broken down for each statement. You can easily review the statements that required the most CPU or read activity or the longest amount of time. If your developers are constructing complex stored procedures or other batches, the trace view in Query Analyzer can be a valuable performance-monitoring tool.

VERITAS™

Benefits of Query Analyzer

Although Query Analyzer is far from a complete performance-monitoring solution, it definitely has its uses. For example, if you've used another tool to identify a particular query or batch as a performance problem, you can use Query Analyzer to narrow the problem to a specific execution step. You can then try to modify the query and interactively resubmit it to SQL Server and take note of any performance changes.

Query Analyzer also provides access to the Index Tuning Wizard, which Figure 2.5 shows. The wizard can analyze a current query or batch, and recommend indexes that could be added or dropped from the database to improve those particular queries' execution. As shown, the wizard also displays a complete breakdown of which indexes the database contains and how they were utilized by the queries the wizard analyzed.

📖 I'll cover the Index Tuning Wizard in more detail in Chapter 4.
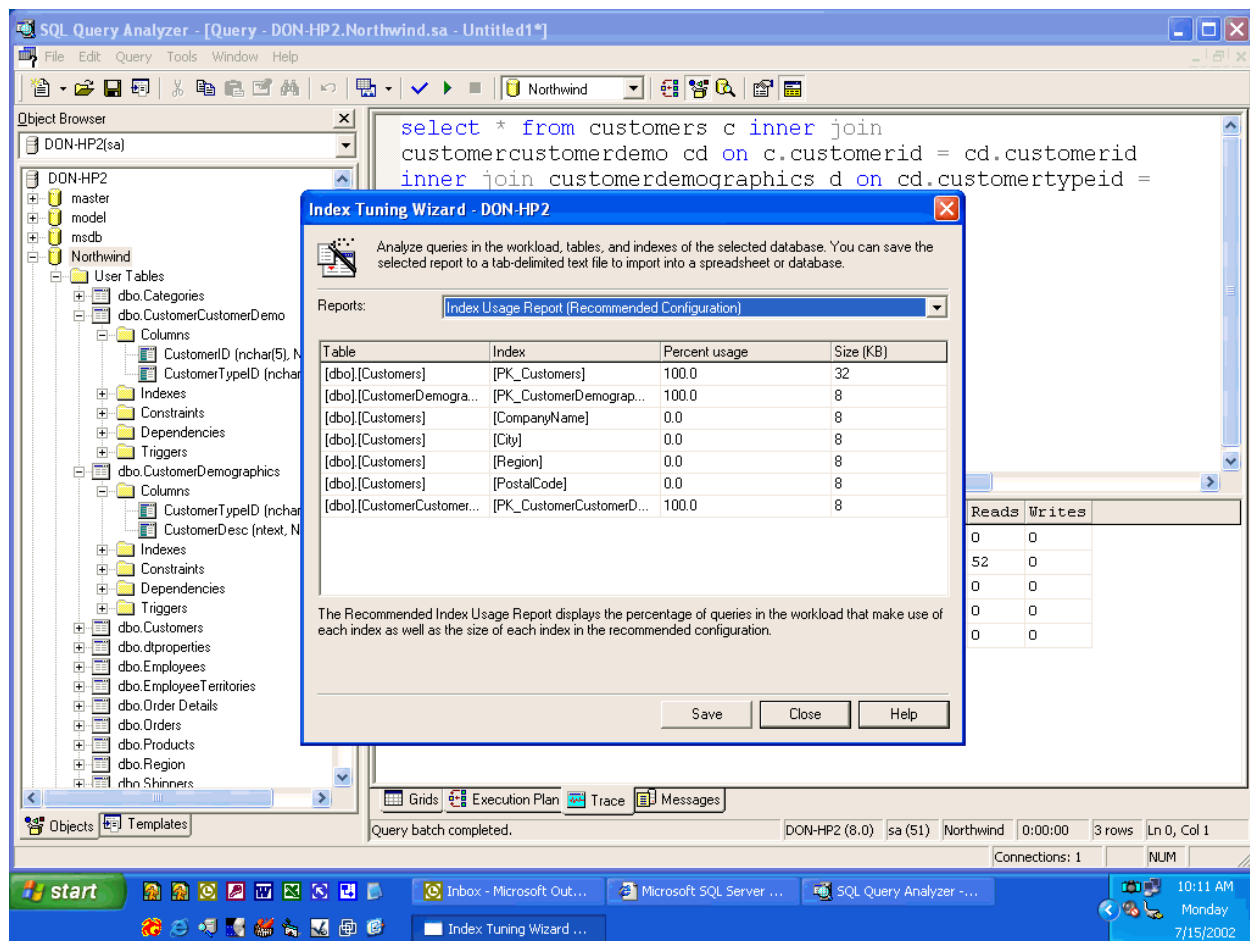
Figure 2.5: The Index Tuning Wizard's recommendations only take into account the particular query you analyzed, not all the queries that run against your databases.

Query Analyzer's strengths at individual query optimization are also its weaknesses. In a large environment, the biggest problem is often tracking down which query is causing performance problems, and Query Analyzer will be of no help there. However, as I'll discuss later in this chapter, Query Analyzer should play an important part in your performance baselines, trending, and trend analysis.

SQL Profiler

SQL Profiler is an often-misunderstood tool that provides an inside look at SQL Server's second-to-second operations. SQL Profiler allows you to view each and every statement that SQL Server executed and view internal details about those statements. You'll start with SQL Profiler *events,* which represent the many different events that can occur within SQL Server: statement execution, deadlock resolution, user logins, and so forth. These events will then be captured in a *trace,* which is essentially a log file of all the events that you asked SQL Profiler to capture.

SQL Profiler Events

When you launch SQL Profiler, you can either choose to begin a new trace or create a new trace template. *Trace templates* are sets of events, data columns, and other settings that define a particular type of trace. SQL Profiler comes with a number of trace templates to get you started, and you can use any trace template as the basis for a new trace.

☞ A template is a template is a template. Think of trace templates as Word templates: When you open a new Word document, it starts as a copy of its template, with all the settings that template includes, such as formatting and styles. Trace templates are similar—every new trace can be based upon a template, and will start with all the events and other settings included in the template. Unlike Word, however, you can also start a blank trace that isn't based upon a template. In that case, you'll have to choose all the trace's settings manually before you can start.

Figure 2.6 shows the event selection screen in a new trace template (which is pretty much the same as the event selection screen for a new, blank trace).
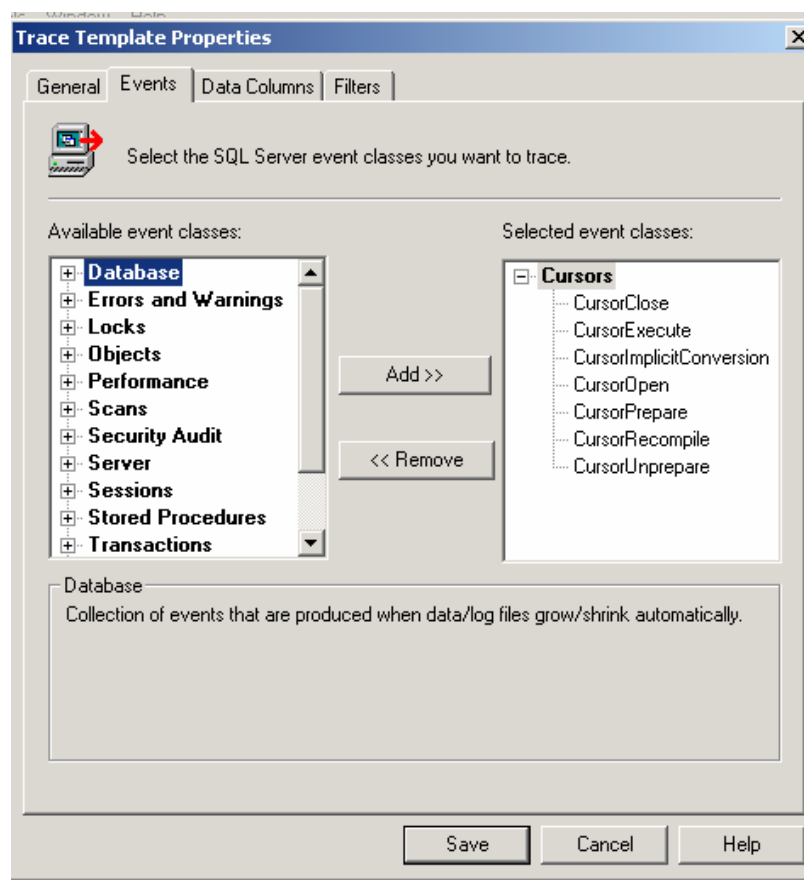
Figure 2.6: Selecting events for a new trace template.

As shown, events are divided into categories, such as database events, cursor events, and so forth. You can select any or all of the events within a category, and you can add as many events as necessary to your new trace. The event categories are:

- Cursors—This category includes events that relate to server-side cursors, or record sets. You can monitor the creation, execution, and preparation of cursors, along with other key events. Server-side cursors are often a performance bottleneck in high-volume database applications, and monitoring the events in this category will allow you to assess cursors' impact on query execution and overall server performance.

- Database—This category monitors database and log file auto-growth and auto-shrink operations. On a busy server, these operations can significantly impact database performance. Monitoring for these events can help you discover if an unnecessary amount of auto-growth operations, in particular, are occurring. If they are, you can manually resize the database to avoid repeated auto-growth operations.

- Errors and Warnings—This category encompasses all of SQL Server's warning and error events, including low-level informational messages, hash warnings, missing column statistics warnings, OLE DB errors, and so forth. I usually include this category in all of my traces—you never know when an unusual error might be causing a performance problem. Keep in mind that some errors occur only after a lengthy timeout period has passed; fixing the errors can therefore improve server response times.

- Locks—This category contains events that indicate when SQL Server acquires a lock, releases a lock, times out while waiting for a lock, cancels a lock, escalates a lock, or deadlocks. The deadlock events in particular can help indicate server problems. Keep in mind that SQL Server always resolves deadlocks automatically by killing one of the deadlocked processes, but only after a timeout period passes. A significant number of deadlocks will be perceived by users as a very slow server, so locating and resolving the deadlock conditions will improve server response time.

- Objects—This category lets you track SQL Server's opening, closing, creating, and removing of database objects. Whenever you use this category, you'll also want to add some trace filters. Keep in mind that practically everything SQL Server does generates multiple events in this category; filtering will make your trace more manageable and useful.

- Performance—This category allows you to monitor a variety of performance-based events, including the all-important Execution Plan event. This event contains the execution plan of every query SQL Server executes. By capturing this event in a trace, you can ensure that queries are consistently using the execution plan you expect them to.

☞ Document execution plans. Whenever your developers implement a new stored procedure or query, document the execution plan of the query as it is expected to run. Over time, you should periodically retrace the query or stored procedure and verify that its execution plan is still the one you expect. Execution plans will change over time as indexes change, as server performance changes, and so forth; catching changed execution plans allows you to review what characteristic of the server has changed and make sure that your database is configured to provide optimal execution speed.

For example, when indexes become fragmented, SQL Server will be less likely to use the index in an execution plan, because the index is slower and costs more to use. If you notice that SQL Server has recompiled your execution plans and that the new plans don't include the index, you'll know that you need to rebuild your indexes so that SQL Server can benefit from them again.

- Scans—This category allows you to monitor the start and end of table and index scans. Table scans in particular are a notorious source of slow database performance, and watching the beginning and end of a scan—and thus determining the amount of time the scan required to complete—will allow you to pin down poorly-performing queries.

- Security audit—This category includes all of SQL Server's security events, including role membership changes, user privilege changes, login and logout events, statement permission changes, and so forth.

- Server—This category includes one event: Server Memory Changed .This event allows you to monitor how frequently SQL Server allocates and de-allocates memory from Windows. Memory management is reasonably time-consuming for SQL Server, and if you discover a lot of this event in a trace, you should manually modify SQL Server's configuration to statically configure memory to meet SQL Server's needs.

- Sessions—This category includes on event: ExistingConnection. This event detects activity by all users since before the trace was started. This event substitutes for the Connect event for connections that are already in place when the trace begins.

- Stored procedures—This category includes events that allow you to monitor SQL Server's usage of stored procedures. A number of these events are performance-related, such as whether a stored procedure was found in cache or had to be retrieved from the database, when stored procedures are recompiled (generating a new execution plan), and so forth. Most SQL Server applications make heavy use of stored procedures, so monitoring cache hits and recompilations can help you determine when stored procedures are performing efficiently.

- Transactions—This category monitors both local SQL Server and distributed transactions, allowing you to see the transactions SQL Sever is performing. Transactions can often create locks on a large number of database objects, and long-running transactions can negatively impact server performance; using this event to monitor transactions will help you track down any problems.

- TSQL—This category monitors the preparation, execution, and completion of SQL statements. You can use these events to determine how long particular statements or batches take to execute.

☞ Profiler will happily collect as many events as you want it to, then it's up to you to wade through them to find the information you were looking for to begin with. To make your job easier, limit your event selection to those that have to do with the problem you're troubleshooting. For example, if you're troubleshooting a long-running stored procedure and want to capture its events, you probably don't need to capture login/logout events and other security audit information.

After you've selected events, you'll need to select the data columns that SQL Profiler will capture. Each type of event has different data columns associated with it, and you'll need to make sure that you capture all the relevant columns in order for the trace to contain useful data. For example, you could have SQL Profiler capture all events but only capture the SPID column (which is a process' ID number) and the trace will be functionally useless to you because it will contain only a list of SPIDs. Figure 2.7 shows the data column selection screen.
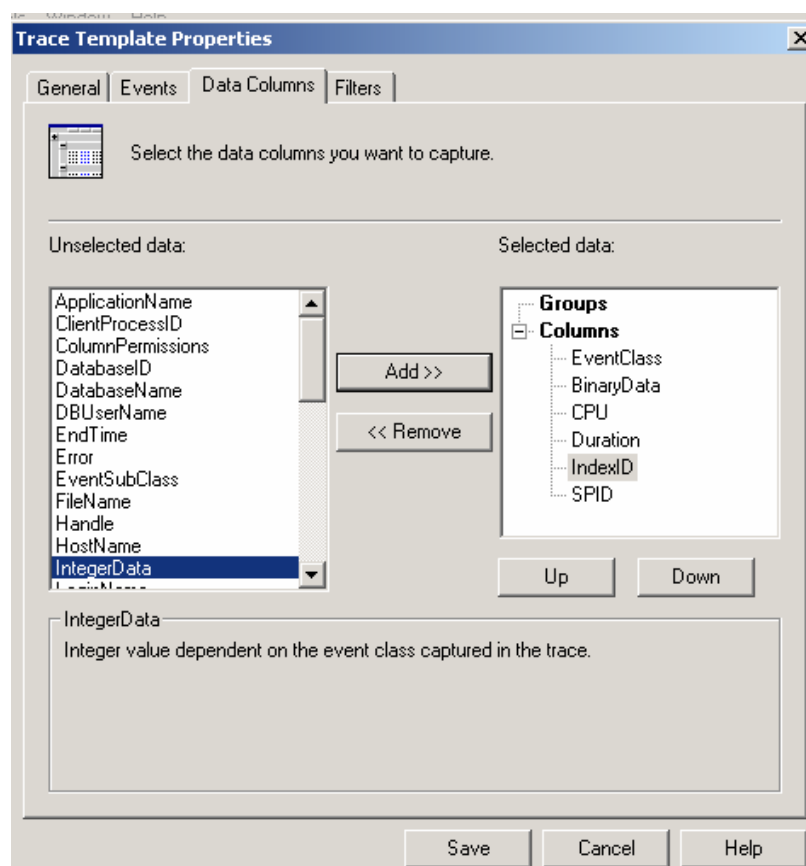
Figure 2.7: Selecting data columns for a new trace template.

Some of the most common data columns that you'll want to capture include:

- SPID—Selected by default, this column shows the process ID that generated the event in the trace.

- ApplicationName—This column indicates the application name provided to SQL Server by applications. For example, connecting to SQL Server with Query Analyzer will generate events with Query Analyzer's application name.

- CPU—This column indicates the amount of CPU utilization that was required to complete the action that generated an event.

- DatabaseName—This column tells you which database was in use when an event was generated. On a multi-database server, you'll often filter this column so that SQL Profiler captures only events associated with the database you want to profile.

- Duration. This column lists the duration of the action that generated the event.

- StartTime/EndTime—These columns indicate the time an event occurred, down to the millisecond.

- LoginName—This column shows the login name of the user that generated the event. Similarly, NTDomainName and NTUserName show the domain credentials of the user that generated the event.

- ObjectName—This column shows the name of the object that generated the event. Objects can include indexes, stored procedures, triggers, and so forth.

- Reads/Writes—These columns show the number of read or write operations, respectively, that SQL Server had to perform to complete the action that generated the event.

Figure 2.8 shows an often ignored and very useful part of traces—filters. As shown, you can filter the events captured by SQL Profiler based on whatever criteria you want. This feature allows you to have SQL Profiler automatically ignore events that don't pertain to the problem you're trying to troubleshoot. For example, if you plan to troubleshoot a long-running stored procedure by submitting it from a particular workstation on your network, you can filter for that workstation's process ID, or filter on your user account. Filtering is especially effective when you're running traces against busy SQL Server computers; without filtering, you'll receive more events than you can easily wade through to find your problem.

💣 Use caution when running SQL Profiler against a production server. SQL Server 2000 uses a new client-server SQL Profiler architecture that imposes significantly less server overhead when you run traces. However, it's never a good idea to run extensive, long-running traces against an already-busy production server because SQL Profiler creates a performance impact.

Instead, run SQL Profiler against a development or test server. You can use SQL Server's tools to move stored procedures, queries, databases, and other objects over to the test server, where you can trace them without fear of impacting production performance.
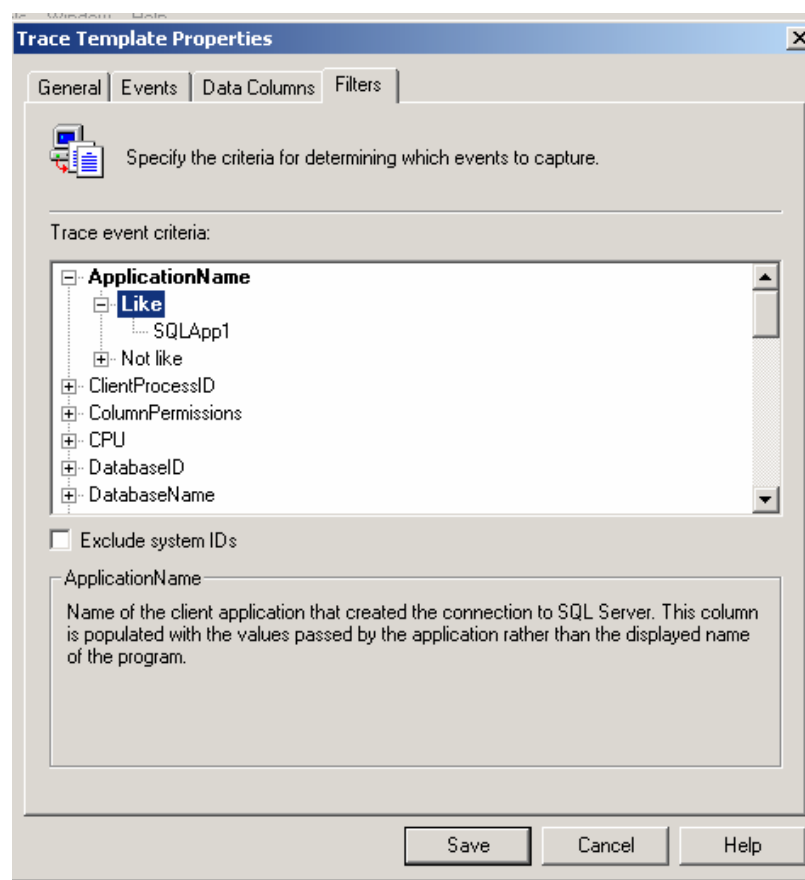
Figure 2.8: Filtering is your best weapon against overly large trace files.

VERITAS™

When you're done configuring your trace, you can let 'er rip. SQL Profiler displays trace events as they occur, but you'll have to be a speedy reader to keep up. Usually, you'll let SQL Profiler capture events while you perform whatever tasks are necessary to troubleshoot your problem, such as executing the query you want to trace, for example. When you're finished, you can stop the trace and analyze the results at your leisure. Figure 2.9 shows a completed trace.
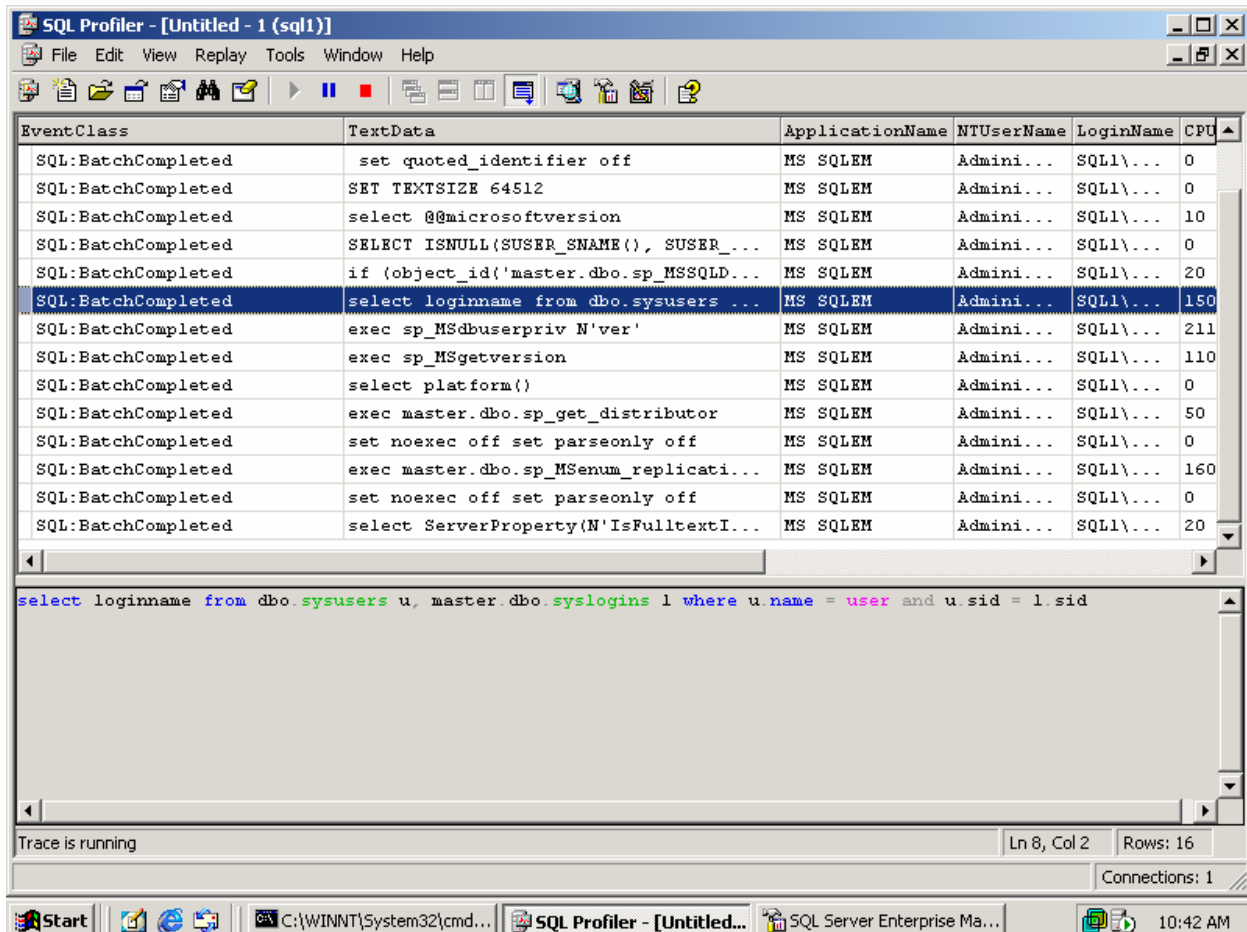
Figure 2.9: Analyzing a completed trace. Notice that the events appear in the upper pane, while the details for the selected event appear in the bottom.

✎ Trace to database or to file? Profiler lets you save captured trace data either to a file or to a database. Saving to a database is a great idea if you plan to run the trace through the Index Tuning Wizard, because the wizard can read trace data right from the database. *Don't* trace to the same server that you're capturing from, though, or you'll impact the performance of the server you're tracing.

If you don't plan to do anything complex with the trace results or you need to easily send them to Microsoft Product Support or someone else via FTP or email, capture the trace data to a file, which is easier to move around than a database.

Using SQL Profiler

When you use SQL Profiler, you'll start by connecting to the SQL Server that you want to trace. You'll need to provide the appropriate login credentials to connect, then SQL Profiler will allow you to begin a new trace. You can start a new, blank trace, start a new trace template, or start a new trace based upon a trace template. SQL Profiler comes with several different trace templates that you can use to get started.

Whenever you start a new trace—whether it's blank or based upon a trace template—you can modify the data captured by a trace. You can select the events to capture, the data columns to capture, and define capture filters. You'll also decide at this point whether your capture will save its data into a file or to a SQL Server database. SQL Profiler saves the data as it's captured, so you'll need to make sure that the file (or database) has plenty of room to save the data you'll be capturing. If you're capturing to a database, do *not* rely on database auto-growth to accommodate your capture; auto-growth can sometimes take long enough to execute that SQL Profiler will drop captured information to keep up. Instead, pre-expand the database to have sufficient free space to hold the amount of data you expect to capture.

When you're done capturing the trace, click the Stop button on the toolbar. You can also use the Pause button to temporarily stop the trace, then resume capturing events later. After the trace is complete, you can scroll through the captured events and analyze them. If you saved the trace to a database, you can also launch the Index Tuning Wizard right from SQL Profiler, and have it tune all the queries captured in your trace.

> 🖉 Running Index Tuning Wizard on traces. The Index Tuning Wizard can offer index tuning suggestions based on *all* the queries in your trace. This technique is better than simply tuning for a single query by running the wizard from within Query Analyzer because you'll be accommodating a greater range of database activity with your tuning results. However, the wizard can only work if your trace includes captured query statements and their execution plans. Those are two events that you can select when you begin a trace.

Benefits of SQL Profiler

SQL Profiler is perhaps the best way to measure statement-level performance in a production environment. SQL Profiler is often the only way (at least, the only way Microsoft provides) to track down queries that are causing deadlocks or other difficult-to-find performance problems. Unfortunately, you'll need a pretty thorough understanding of how SQL Server works to interpret SQL Profiler traces correctly. Those traces are complex enough that entire books have been written on the subject of understanding them (and of course, I'll cover some aspects of traces throughout this book); not every administrator has time to become a SQL Profiler expert.

Third-Party Performance Analysis Tools

By now, you're probably wondering how Microsoft expects you to tune SQL Server's performance at all. System Monitor doesn't provide detailed-enough information, Query Analyzer only lets you deal with individual queries or batches, and SQL Profiler is akin to rocket science sometimes. Fortunately, there's a fairly robust selection of third-party solutions designed to address SQL Server performance.

VERITAS Indepth for SQL Server

VERITAS's offering is one of the newest SQL Server performance tools. Like NetIQ's AppManager, Indepth continuously monitors your servers' performance conditions. One unique feature of Indepth is its understanding of Enterprise Resource Planning (ERP) systems such as SAP R/3—Indepth not only monitors server health in the abstract but also for the specific ways that systems such as R/3 utilize SQL Server. This capability allows Indepth to more precisely manage performance in R/3 and other ERP environments.

Indepth measures performance down to the statement level, essentially acting as a combination System Monitor, Query Analyzer, and SQL Profiler—with automation throw in, of course. Indepth scans for unacceptable performance conditions, such as long-running queries, and automatically tries to identify the precise problem and correct it or recommend solutions. This proactive, granular level of performance monitoring and management is unique to Indepth. Read more about Indepth for SQL Server at http://www.veritas.com.

Indepth's strength is its ability to detect performance problems both in a generic SQL Server environment and in ERP-specific environments, determine the cause of the problems; and offer corrective actions. These capabilities offer end-to-end performance management at a very detailed level, helping to automate some of the most difficult tasks required to tune SQL Server performance.

NetIQ AppManager for SQL Server

NetIQ's AppManager for SQL Server is an automated server health solution. In a nutshell, AppManager watches your servers' performance at all times, and classifies various conditions as healthy or not based on the performance data your servers provide. Basically, AppManager collects performance counter data and compares it with the product's preconfigured thresholds. Those thresholds determine whether various server conditions are considered healthy by AppManager.

> 🖉 If AppManager's functionality sounds similar to Microsoft Operation Manager's (MOM's) functionality, you're not far off; MOM is actually a licensed version of NetIQ's enterprise management suite, which AppManager is a part of. Like AppManager, MOM's job is to watch server performance and classify it as healthy or not based on predetermined performance thresholds.

> 📖 For more information about MOM and AppManager, check out Jeremy Moskowitz's *The Definitive Guide to Enterprise Manageability*, a link to which can be found at http://www.realtimepublishers.com.

AppManager allows you to continuously monitor multiple servers from a single location. It has built-in thresholds that tell you whether your servers are healthy, without making you worry about what performance conditions actually *mean* healthy, and without having to continually record raw performance data on your own. Learn more about AppManager at http://www.netiq.com.

Although AppManager is designed to continuously monitor SQL Server and call your attention to performance problems, its primary focus is on server-wide performance problems such as memory utilization and processor utilization. It also catches events from the SQL Server log and the Windows event logs and can be configured to handle them for you, if necessary, through actions that you define. AppManager can analyze down to the SQL Server statement level, correlating memory utilization or locking problems to specific statements, among other capabilities.

A major benefit of AppManager, of course, is that it's not a SQL Server-specific tool. AppManager is an entire enterprise-management framework; and AppManager for SQL Server is essentially a plug-in module that adds SQL Server-specific functionality. If your company has already standardized on AppManager for enterprise manageability, then AppManager for SQL Server is a great add-in.

Intrinsic Design Coefficient

The best description for Intrinsic Design's Coefficient is "SQL Profiler on steroids." Coefficient captures a somewhat wider variety of events, and perhaps most importantly, uses a wizard to help set up new traces. In other words, you won't have to worry about selecting the right events and data columns to capture; Coefficient can set itself up to capture the information you need for a variety of common performance-tuning tasks. Coefficient also includes analysis tools that can look at the traces you've captured and provide you with tips for correcting problems and improving performance. Learn more about Coefficient at http://www.idisoft.com/.

Coefficient isn't a continuous monitoring solution. It relies on you to run the tool whenever you need to capture specific information, much like SQL Profiler. You'll also need to have a good idea of what the problem is to begin with so that you tell Coefficient which events it should capture. So, although Coefficient can be useful for determining why, for example, a specific query is running so slowly, it can't necessarily help you identify a particular query as the cause of a server-wide performance problem.

Lumigent Log Explorer

Lumigent Log Explorer isn't marketed as a performance-optimization tool, but it certainly has applications in performance optimization. Log Explorer's whole purpose is to scan through the reams of information SQL Server dumps into its log files, extracting specific information to help you solve problems, audit server activity, or monitor performance conditions. You can use Log Explorer to hunt down log entries that hint at performance issues, then take the appropriate steps to correct those problems. Check out Log Explorer at http://www.lumigent.com.

Like many other tools, Log Explorer is really only useful if you know what you're looking for; its purpose is to help you find something more quickly than you could by manually scanning through SQL Server's log files.

When to Measure Performance

Perhaps the biggest question in performance optimization is, When do I measure performance? Many administrators don't bother firing up System Monitor until something is wrong (for example, users complaining that their database application is running too slowly). However, performance isn't a topic that you deal with only when things are wrong: Performance is best measured all the time, and most especially when things are running just fine. That's actually why tools such as VERITAS's Indepth and NetIQ's AppManager are so valuable, because they can continuously monitor and measure performance rather than requiring you to take time out of your busy schedule to do so.

> 📖 Continuously measuring performance is part of a performance-tuning methodology, and I'll introduce you to such a methodology in Chapter 3.

Before you even decide *when* to measure performance, you'll need to decide *what* to measure, depending on your environment. Most certainly, you'll be concerned with factors such as processor and memory utilization, the length of time it takes SQL Server to complete certain key queries, and so forth. Chapter 3 will focus mostly on identifying these key performance indicators. After you've decided on a standardized set of performance points to measure, you can create a performance monitoring schedule. Here's what I recommend for most enterprise environments:

- Take a complete set of measurements on a new SQL Server computer. Focus on hardware-related measurements, such as processor utilization, when the server isn't being used to execute queries or other tasks. This set of measurements will become your *at rest* baseline for the server.

- Take another complete set of measurements when the server is operating under an average production load, complete with query execution and whatever other tasks the server performs for your users. This set of measurements will become your *production* baseline. Take specific note of response times for key queries.

- Depending upon the growth your organization is experiencing—either in personnel or in data—take additional complete sets of measurements every week, month, or quarter. If you're not sure how your growth will impact performance, start by measuring every week. If the measurements aren't showing much change, you can fall back to measuring every other week or once a month. These measurements will become *trend points,* which I'll discuss in the next section.

Note that you'll need to perform these measurements individually for *each server* on your network. Even if your servers utilize identical hardware, different servers will wind up being used somewhat differently by your users, so you'll need to measure them independently.

You'll also need to measure performance again every time anything changes on your server. And I do mean *anything*: Whenever a developer makes a significant change to a stored procedure, adds a new stored procedure, changes an index, rebuilds an index, and so forth. Any action that can impact the performance of your server needs to be accompanied by a new set of measurements. Sounds like a major time-consuming pain in the neck, doesn't it? It is, and again, it's where tools that provide continuous monitoring can really be a headache-saver.

VERITAS™

> ☞ How you record your performance measurements depends on how much time and effort you want to invest into performance management. At the least, you need to jot down key performance measurements on a piece of graph paper. You may prefer to enter the information into an Excel spreadsheet. If you're really on the ball, you'll enter the information into a SQL Server database, from which you can report on the information and analyze it more easily. No matter what you decide to do, you must document and archive your performance measurements for them to be of any use.

Trending and Trend Analysis

After you've got a ream of performance measurements, you can start actually using them in a process called *trending*. Trending is simply the act of comparing a series of performance measurements to one another and performing *trend analysis.* Trend analysis is basically just making predictions based on that sequence of performance measurements.

For example, suppose you install a brand-new SQL Server computer on your network. Your at-rest processor utilization is a nice low 4 percent. Your production processor utilization is only 50 percent, and you've decided that a processor utilization of 85 percent should be cause for alarm, giving you plenty of room for growth. If you're lazy, you can just wait until the processor utilization tips over the 85 percent mark, then do something about it. A proactive administrator, however, would continue to monitor the server's performance and use trend analysis to *predict* when the server will pass 85 percent, and do something about it *before* it happens. Suppose you monitor the server's performance once a week, and log the following information:

- Week 1: Added 100 new summer interns for data processing. Processor utilization is now averaging 70 percent.

- Week 2: Added 50 new interns. Processor utilization is now at 78 percent. Data entry department says no new interns will be added and the existing ones will be let go at the end of the summer (still 8 weeks away).

- Week 3: Developers report that an additional data-entry application will come online in 4 weeks. Test server shows about 30 percent processor utilization.

- Week 4: New sales campaign kicked off and data entry volume doubled—processor utilization is up to 82 percent.

By week 3, you know you're in trouble and by week 4 you know it's going to come soon. The new data-entry application may add another 20 to 30 percent utilization to your already close-to-the-line SQL Server computer, and those hard-working interns won't be leaving until several weeks later, putting your server well over the line of acceptable performance. Plus, the new sales campaign has already doubled data-entry volume. Although this example is an extreme case, it shows how constant performance monitoring, and a close eye on current events, can help you identify server problems *before* they happen. In this example, you can start talking to the developers about holding off on the new data-entry application until after the interns leave, or you can look at renting a server to take some of the slack until that time. The point is that your server is still running within acceptable limits, giving you time to do something before performance worsens.

VERITAS™

Processor utilization, however, is rarely the key indicator that you'll be watching for. It's one of them, but it's not something that immediately impacts users. If you think of performance problems as diseases, processor utilization might be the disease—an overburdened processor— and the symptoms that your users will first experience are slow-running applications. That's why, in addition to measuring basic performance statistics such as processor and memory utilization, you need to measure the symptoms that users will notice. For the most part, that means monitoring query response times. Eventually, your trending will help establish a correlation between hardware-utilization factors and user symptoms, and you'll be able to make better performance predictions. For example, take a look at the chart in Figure 2.10.
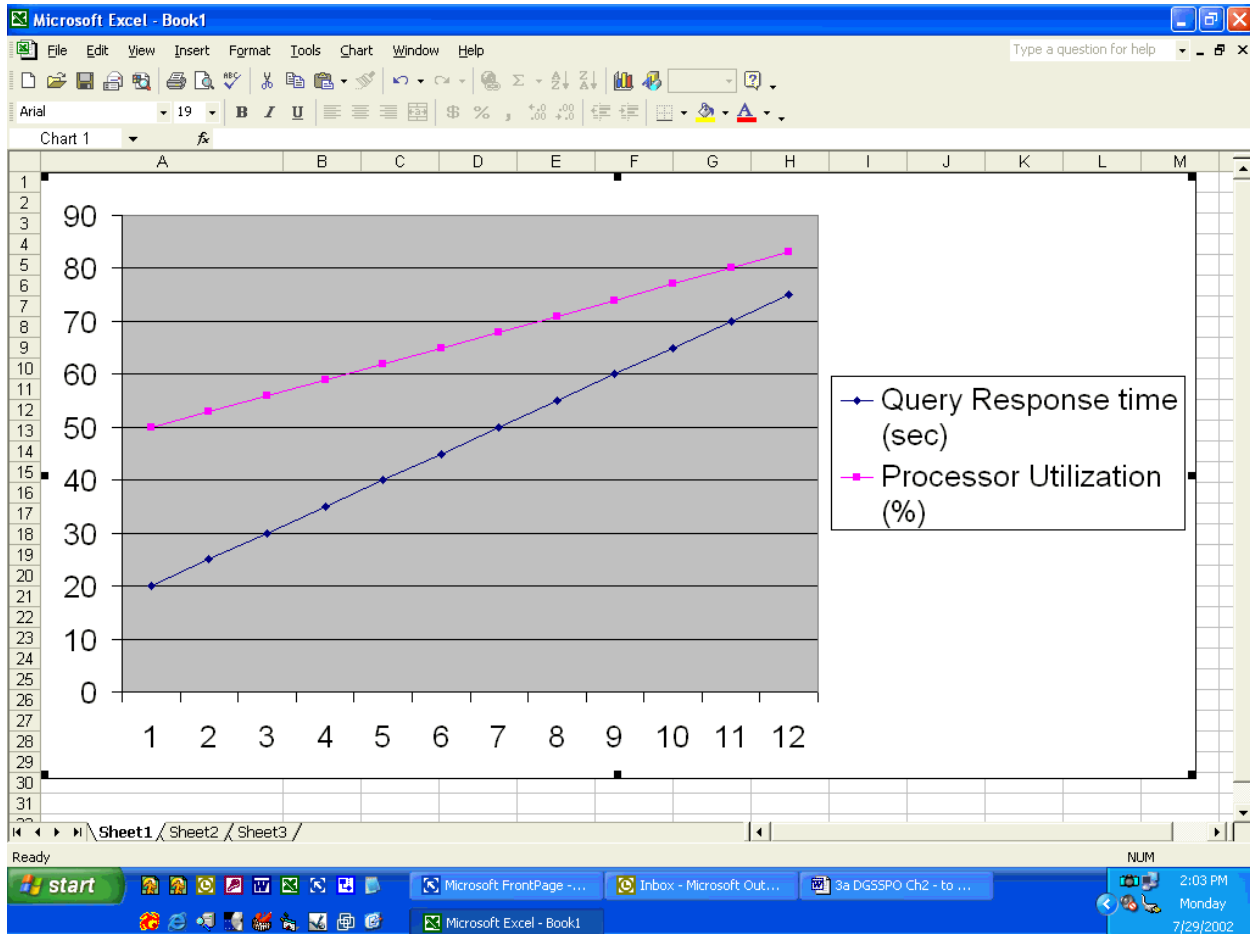
Figure 2.10: Using trends to correlate hardware performance with user symptoms.

In this example, critical query response time increases by about 5 seconds for every 3 percent of processor utilization. Although real-world performance isn't usually that straightforward, it's a good example of how trending can help you intercept problems before they occur. Using the correlation between processor utilization and query response time, you can extend the chart into the future, determining that query response time will become an unacceptable 60 seconds when processor utilization hits 75 percent. That makes 75 percent your "line in the sand," and you can even predict the data that you'll hit that line based on your trend-point measurements. Now you'll have plenty of time to come up with corrective action before the problem actually occurs.

It's important that your trends take current events into account, too. Continuing the precious example, suppose you ask your developers to fine-tune some key queries so that the queries run faster. The developers will most likely do so in a development or test environment, which won't precisely reflect the performance conditions of your production environment. Thus, you'll need to perform additional trend analysis to determine the benefit of their changes. Figure 2.11 shows the revised trend chart, complete with a notation of when the developers' fine-tuned query was implemented in production.
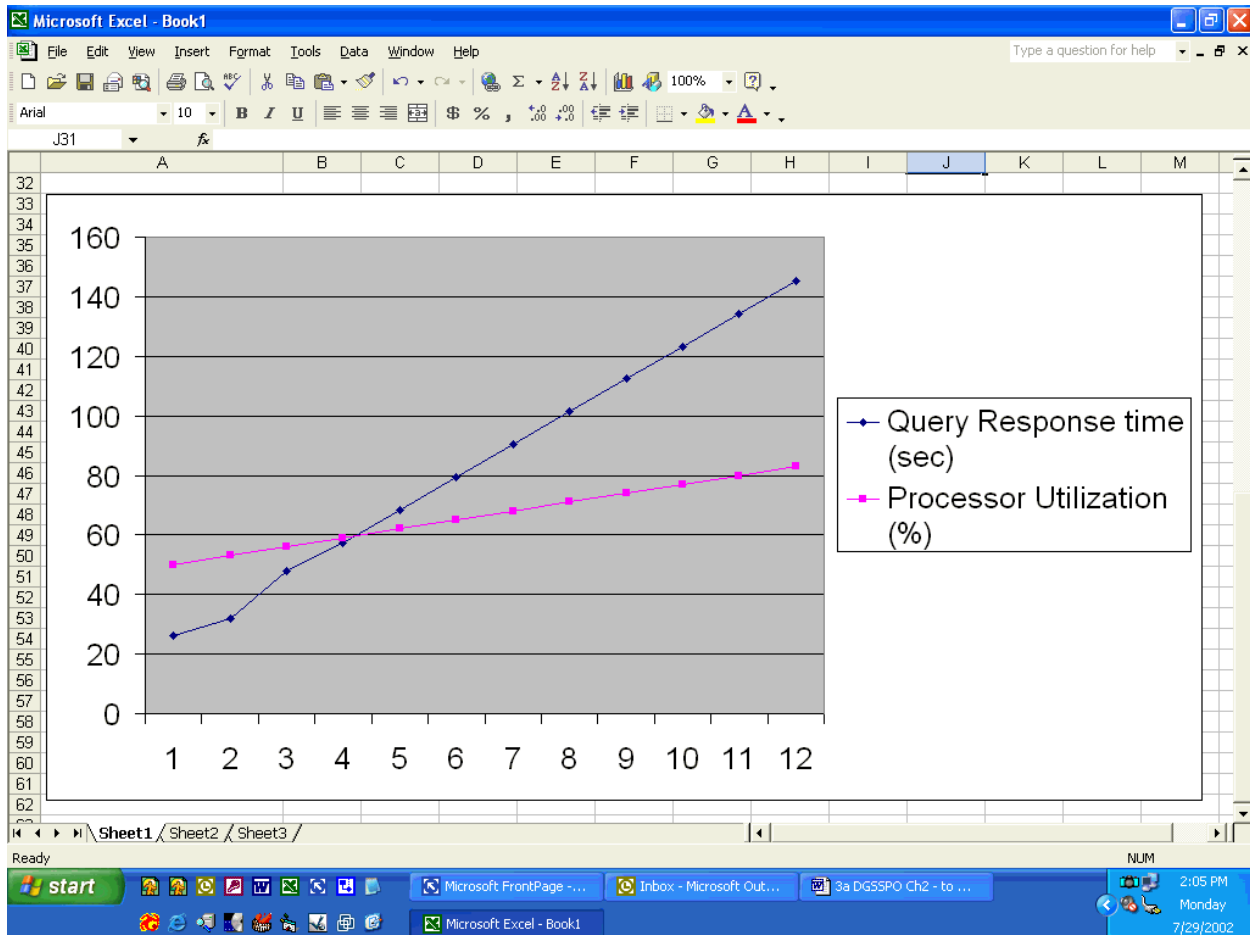
Figure 2.11: Continuing to trend performance after query changes have been implemented.

In this case, the query seems to be performing worse than before, which means your server will run into problems sooner rather than later. (For more information about why such might be the case, see the sidebar "Why Performance Doesn't Always Improve.") Because you've measured performance right after the change was implemented, and because you only allow your developers to implement one change at a time (you *do* restrict them to one change at a time, right?), you'll know exactly what needs to be undone to return the server to its former operating condition.

Why Performance Doesn't Always Improve

It's not unusual for a developer to fine-tune a query for better performance and wind up with worse overall server performance. That's because fine-tuning a query can be a very narrow-minded affair. For example, a developer might decide to drop a few indexes to make a particular UPDATE query run more quickly. However, doing so neglects the fact that a dozen other queries might have relied on those indexes for *their* performance. That's why it's important to test query changes in an environment as close to production as possible.

Also, the performance improvements found in a test or development environment might not carry over to a production environment. Query performance relies heavily on the efficiency of SQL Server's internal query optimizer, which decides how queries will be executed. On a less-busy test computer, the optimizer might spend a few extra seconds thinking up a really efficient query execution plan; on a much-busier production server, the optimizer might settle for a less-efficient plan just to get the query executed as quickly as possible. That less-efficient plan might not make use of additional indexes or other performance refinements.

Developers often have load-simulation routines that allow them to automatically submit a production-level workload to their test servers. If your developers have this capability, ask them to test their refined queries under a production-level workload. However, there's no substitute for testing the query in the actual production environment and carefully noting how its performance has changed. No matter what kind of simulated load you place on a test server, the complexity of SQL Server leaves plenty of room for different performance characteristics in a production environment.

Trend analysis can be a good way to plot the growth of your company, too, even if nobody can tell you exactly how many new employees have been hired or how much new data is going into the company's SQL Server databases. Figure 2.12 shows a basic trend plot, which shows overall server utilization going up over a period of time. By establishing what you consider to be minimally acceptable performance levels, you can extend the lines of the trend chart and determine when growth will surpass the server's ability to keep up. Whether that growth is due to new employees or an increase in data volume doesn't matter—you'll be able to start working on the performance problem before it even exists, and create a plan to add servers, increase existing server capacity, or some other solution.
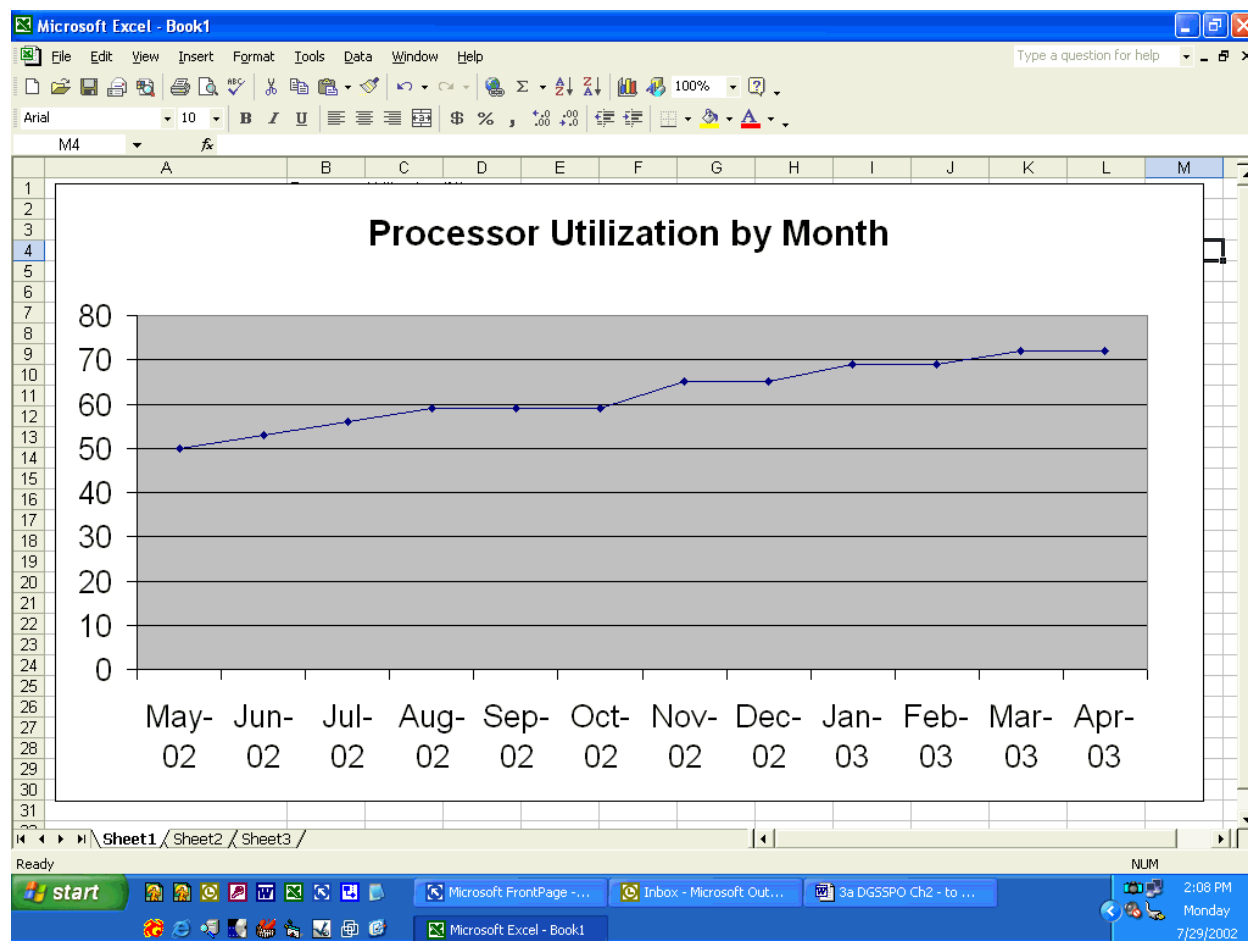
Figure 2.12: Plotting trend points on a graph is an easy to way perform basic trend analysis.

Again, the whole point with trend analysis is to identify performance trends, whether they're a result of growth, configuration changes, or new applications. The trends can help you determine when your server's performance will become unacceptable, and take action *before* users start complaining.

Summary

In this chapter, I've introduced you to the basic tools of the SQL Server performance trade: System Monitor, SQL Profiler, and Query Optimizer. I've also introduced you to some third-party performance-management tools, which you can check out and evaluate for their suitability in your environment. At this point, you should have a basic understanding of how to use System Monitor, Query Analyzer, and SQL Profiler, and you'll be relying on those tools to gather performance information and fine-tune SQL Server throughout the rest of this book. In the next chapter, you'll learn about SQL Server's key measurement points, and learn how to identify and resolve the most common performance bottlenecks.

You should also understand when to measure performance, how to build a trend chart from your measurements, and how to perform basic trend analysis. In the next chapter, I'll build on the idea of trending to create a complete performance-tuning methodology that you can use to manage SQL Server performance in your environment.

Chapter 3: Performance Tuning Methodology

As I've mentioned previously, performance tuning shouldn't be a one-time task, nor should it be something that you do only when performance is already poor. Performance tuning is something you need to do on a regular basis, using a consistent methodology to ensure consistent, accurate results. There are many methodologies that you can use; what I'm going to show you is a common, fairly generic one that should work well in most environments.

First, you'll need to select your performance monitoring tool. Windows' built-in System Monitor is an obvious choice because it's free, but it doesn't lend itself well to recording performance trends over long periods of time. If you decide to use System Monitor, you'll have to do a lot of manual work to record performance values and graph trends.

> 📖 I discussed System Monitor (also called Performance Monitor) in Chapter 2, in which I also introduced you to trending and trend analysis.

There are a number of freeware, shareware, and commercial tools designed to automatically record performance values and graph trends for you. One popular tool is Predictor from Argent Software (http://www.argent.com), which Figure 3.1 shows.
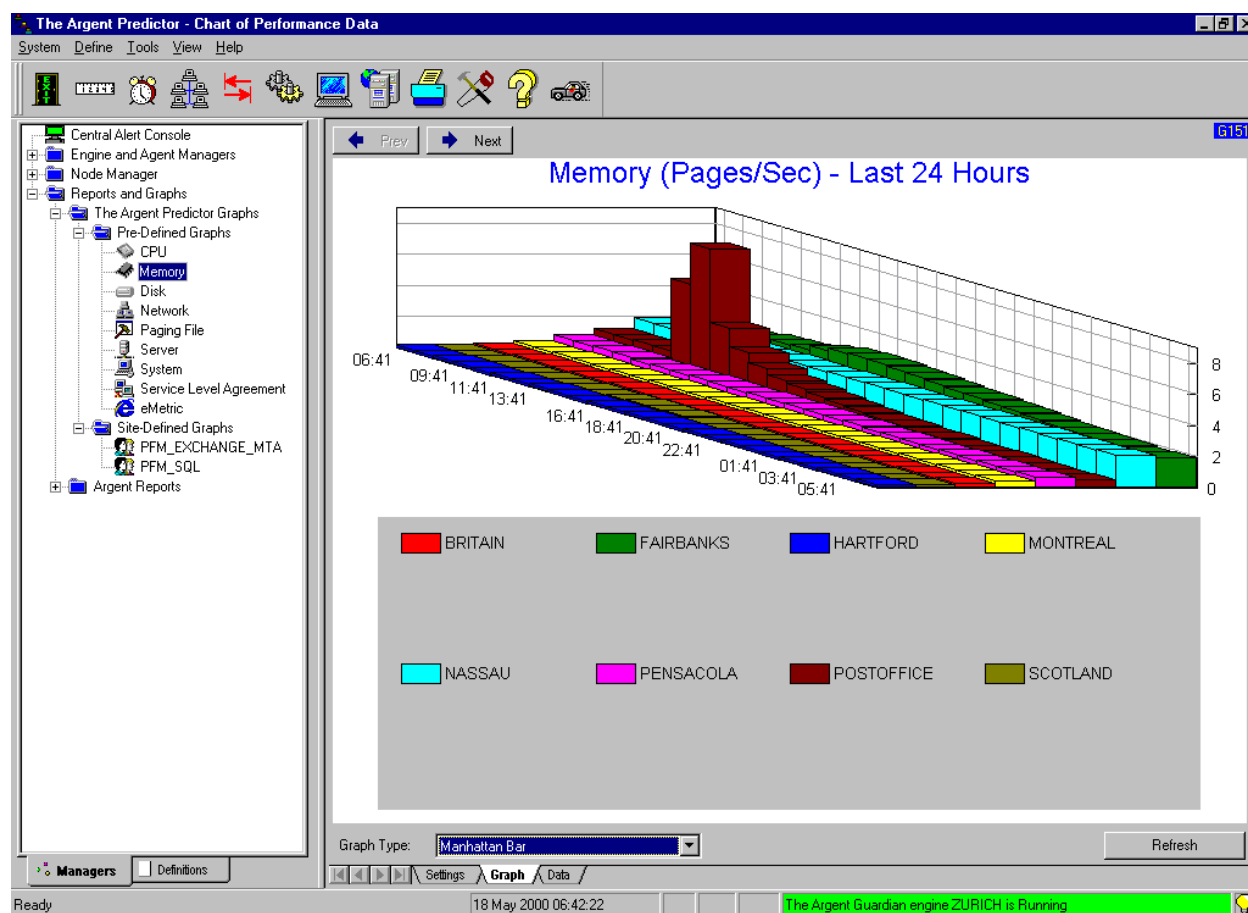
Figure 3.1: Predictor can record predefined performance values and plot trend graphs automatically.

VERITAS™

Tools such as Predictor can be useful in any performance tuning methodology because they help you easily spot trends, such as increasing memory consumption or decreasing query access time. Predictor and similar tools won't do anything to help you fix poor performance, of course; they simply make the job of recording performance values and creating trend graphs somewhat easier than doing it manually by using System Monitor.

> 🖉 To make sure you and I have a level playing field between us, I'm going to assume that you're using the free System Monitor tool for trending purposes. If you're using a different tool, all of the concepts I show you will still apply; you'll just use your tool's specific techniques to gather and track performance information.

The Purpose of Trending: A Quick Review

As I discussed in the previous chapter, the basic purpose of trending is to spot performance problems *before* they become problems. By regularly monitoring key performance factors, you should be able to spot declining performance trends and detect oncoming bottlenecks before they affect your users. At its simplest, trending involves the regular collection of performance data, which you then place into a graphical chart of some kind. The chart helps you visually identify trends, such as increasing memory utilization or decreasing query performance, and focus in on those trends to find out what's going on and what corrective action needs to be taken.

Trending is only useful when it's performed regularly, on each of your SQL Server computers, and when you take the time to make and carefully examine the necessary graphs. Some performance trends will be subtle, and you won't spot them until you have three or four cycles of performance data on your chart. In fact, you should be slightly suspicious of any "trend" that appears after one or two performance monitoring cycles. Such trends may simply reflect temporary aberrations. Whenever you spot sudden trends, consider repeating your performance-gathering cycle after a short while to make sure the performance values you originally recorded weren't just transient conditions. For example, if you gather performance data once a week and start at the beginning of the month, you might notice a sudden marked decrease in server performance at the end of the month. That could be a transient condition caused by your company's end-of-month bookkeeping activities. You could easily re-gather that performance information a few days later or wait until the following week to see whether the performance change was an actual trend or simply an aberration caused by unusually high user activity.

System Monitor's Log View

In the previous chapter, I focused primarily on System Monitor's chart view, which shows a real-time chart of server performance. In this chapter, I'm going to concentrate mainly on System Monitor's log view. Like chart view, log view lets you specify the performance objects that you want to monitor. Log view also lets you specify how long you want to collect data and will automatically collect the data and save it to a performance log, which you can analyze later at your leisure.

To create a new performance log, open the Performance Console. Expand the Performance Logs and Alerts items, right-click the Counter Logs item, select New Log Settings from the pop-up menu, and give the new log settings a name. You'll see a dialog box (see Figure 3.2) that lets you specify the performance objects and counters that you want included in the log. You can also define how often System Monitor will collect data from these counters. The default setting is every 15 seconds; if you're planning to collect data over a long period of time, change the interval to a more reasonable period, such as once every 2 or 3 minutes.

> ✎ The interval setting specifies an averaging period. Many administrators think that System Monitor simply sits and waits for the interval to pass, then saves the current value of the performance collectors. That's not quite how it works. Instead, System Monitor averages the performance values for the duration of the interval, then saves those averages to the log. For example, if you specify a 1-minute interval, System Monitor will average each performance counter for a minute, then save that average to the log. This behavior is actually better for trending because you're not collecting disparate data points. System Monitor's averaging helps smooth out peaks and valleys in performance to give you a true picture of your server's average performance.
>
> A long interval, however, can obscure frequent performance spikes by averaging them over an unnecessarily long period of time. I don't recommend an interval of more than 10 minutes if you're collecting for a 24-hour period, and I recommend shorter intervals if you're collecting for shorter periods of time.
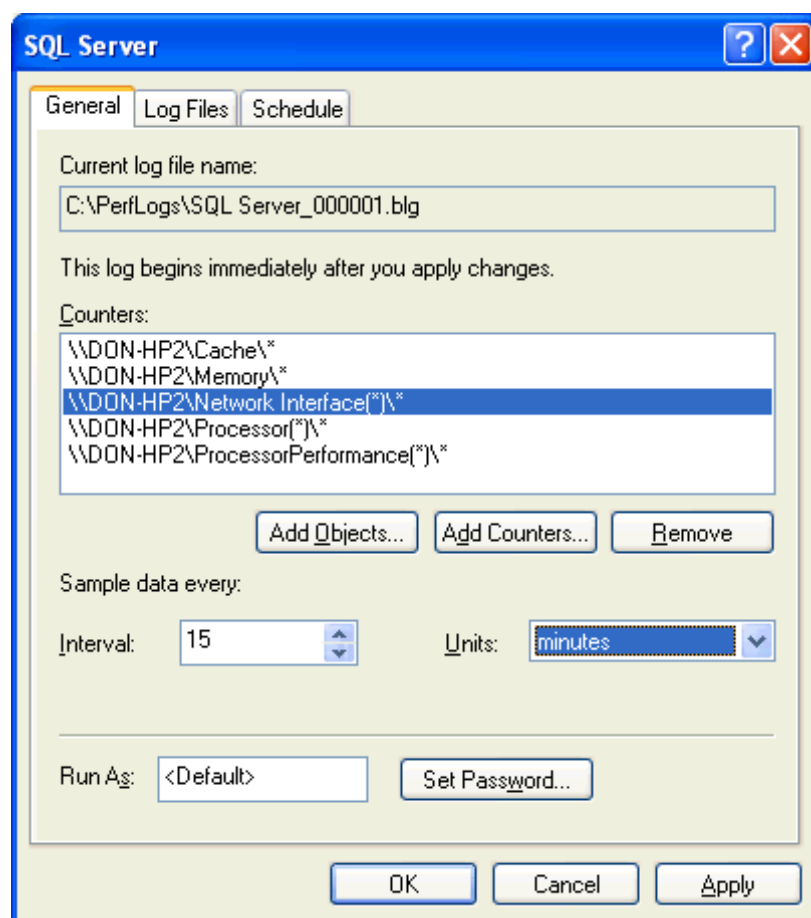
Figure 3.2: Adding performance objects and counters and specifying a collection interval.

Figure 3.3 shows the Schedule tab for the log properties. On this tab, you can set when the performance collection will start, when it will stop, and what System Monitor will do after it stops collecting data. As the figure shows, you can start collecting manually or automatically at a specific time and stop collecting manually after a specified interval or at a specified date and time. I like to have System Monitor notify me when collection is finished by using the Net Send command to send a pop-up message to my workstation; you might want to have System Monitor run another executable or simply do nothing at all when the collection is complete.
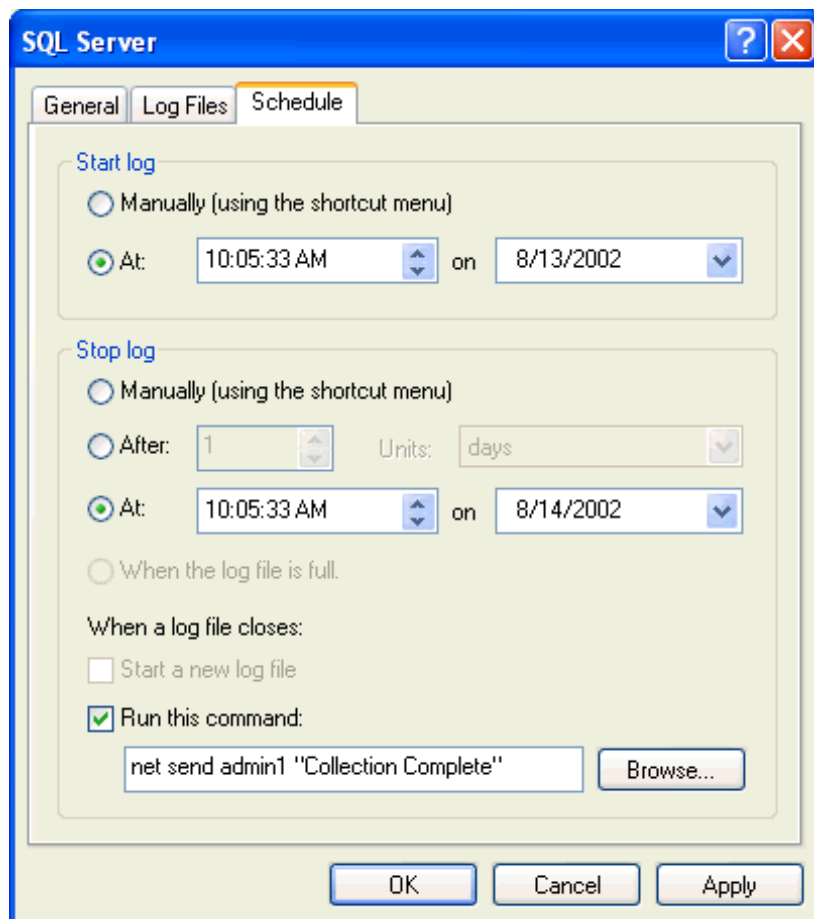
Figure 3.3: Scheduling the beginning and end of the collection period.

Don't run System Monitor on the server you're monitoring! As I've mentioned previously, System Monitor can impose a performance load of its own. System Monitor lets you collect data from remote servers. So for the most accurate results, you should run System Monitor on another computer. The computer running System Monitor will need the SQL Server administrative tools installed so that System Monitor will recognize the SQL Server-specific performance objects.

One of the benefits of many third-party performance monitoring tools is that they run as a background service, which means they don't impose quite as much overhead as System Monitor does (System Monitor runs in the foreground and makes fairly heavy use of the OS's graphic-drawing routines). Third-party performance monitoring tools that run as a service almost always run on the server on which they're monitoring.

After you've collected data into a log, you can view it with the regular System Monitor. To do so, click the Log Source icon (the little gray cylinder), and change the log source to the log file you want to review. You can also specify a time range if you only want to review a portion of the log; however, you'll most often want to review the entire log.

After you've switched System Monitor from reviewing real-time data to reviewing the data in your log file, you can use either chart or report view to analyze the data you've collected. I prefer report view for trending purposes because it displays the raw performance data, including averages for the time period that you're reviewing. Those averages can be punched into an Excel spreadsheet for recordkeeping, as I'll show you later. Figure 3.4 shows report view.
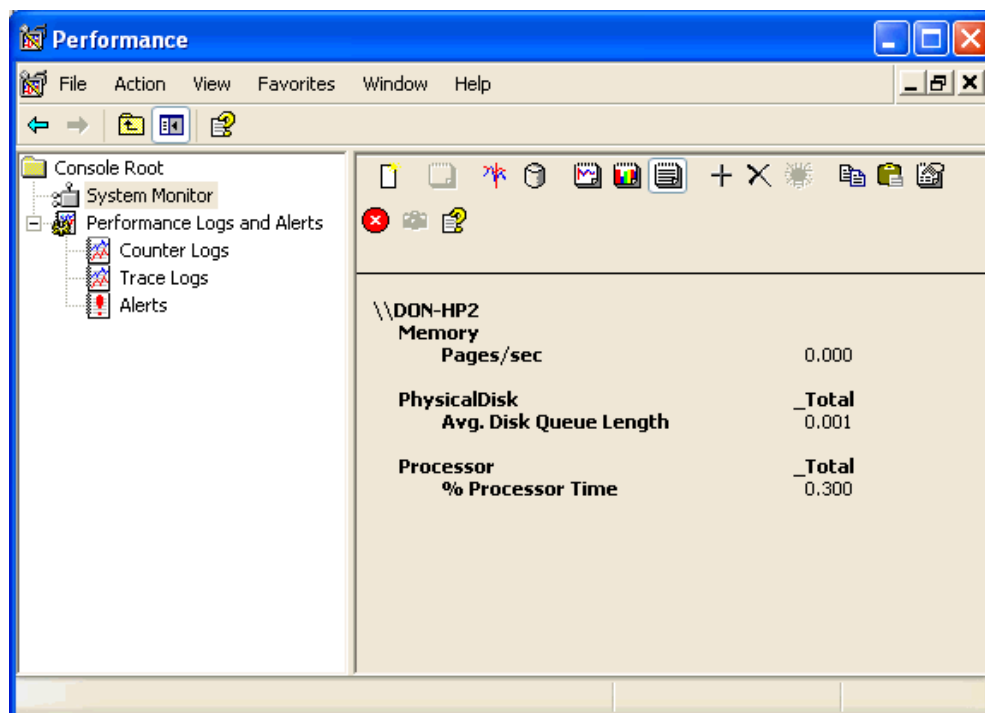
Figure 3.4: System Monitor's report view.

☞ You can also use the regular chart view to view the short-term trends included in the log file you're reviewing. I like to let System Monitor log data for an entire day, then use the chart view to see how my servers' performance varies throughout the day. Such short-term trends can help you identify idle periods that might be ideal for running maintenance jobs as well as discover recurring peak periods that exceed the server's capacity on a regular basis.

Selecting Performance Objects

Obviously, your performance logs will only be useful if they contain useful data, which means you need to be careful about which performance objects you include in the log. Including everything System Monitor offers will probably be overkill, and you'll have to wade through the data to get to the information you need. Later in this chapter, I'll cover most of the key performance objects and counters that you'll need to watch to catch most SQL Server bottlenecks. For now, you just need to know the most commonly monitored objects:

- Memory

- Network Segment

> 🖉 Can't find the Network Segment object? If this object isn't available when you try to add it, you need to install Network Monitor on your computer. Network Monitor is an optional component included with Win2K and later, and you can add it from the Add/Remove Windows Components Control Panel applet.

- PhysicalDisk

- Processor

- Server

- System

- SQL Server: Access Methods

- SQL Server: Buffer Manager

- SQL Server: General Statistics

- SQL Server: Locks

- SQL Server: SQL Statistics

Your environment might require you to collect other counters and objects. Hopefully, some of the tips I've provided in this chapter will direct you to the most useful SQL Server and Windows performance objects for consideration in your trending program.

Storing the Log Data

As I've already mentioned, System Monitor's report view can provide a useful overview of the log data you collect. Unfortunately, report view isn't very granular, so it's not useful as much more than a high-level overview. Fortunately, you can use SQL Server to store performance log data, which makes it much easier to generate detailed performance statistics for trending purposes. However, Microsoft doesn't make it terribly easy to get performance data into SQL Server. It is possible, though, and it's worth the effort.

> 🖉 Third-party tools are easier to use! As I've already mentioned, third-party performance monitoring tools can relieve much of the burden of manual monitoring and trending. Most high-end performance solutions, such as VERITAS's Indepth for SQL Server and NetIQ's AppManager for SQL Server, automatically log data to a database, automatically analyze trends, and so forth.

The following steps walk you through what you'll need to do to get your performance data into a SQL Server database:

1. Open System Monitor and get your log into the chart view. Make sure all the counters you want to export to SQL Server are displayed at the bottom part of the graph.

2. Export the counter data into a comma-separated values (CSV) file. To do so, right-click the chart area and select Save Data As from the pop-up menu. Be sure to change the file type to Text File (comma delimited); the default binary file type won't be of any use.

VERITAS™

3. Tweak the data a bit, using Notepad or Excel. Normally, you'll just want to clean up the column headings a bit. As Figure 3.5 shows, the column headings can be a bit long and won't translate well into SQL Server column names. Renaming them in Excel, then resaving the file will make the next step a bit easier. You should also add a date column with the current data so that your SQL Server database will include date information.
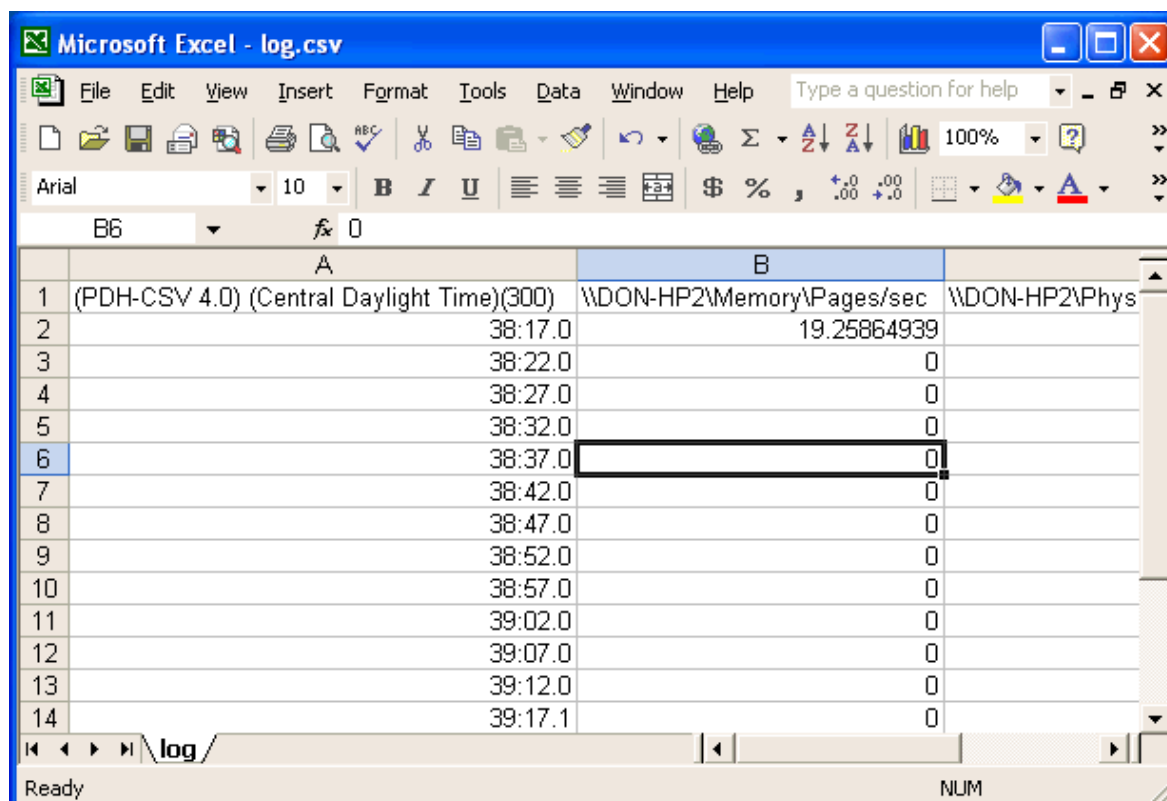
Figure 3.5: Renaming the column headers will make it easier to import the data into SQL Server.

4. Use Data Transformation Services (DTS) to import the data into a SQL Server table. Simply use SQL Server's Import/Export Wizard to import the log file (either in CSV or native Excel format) into a table. The wizard can even create a new table if this is your first import. At the end of the wizard, be sure to select the option to save the DTS package for future use. From then on, you'll be able to simply execute the package to import new performance log data into the table.

Analyzing the Data

After you've got all the data in SQL Server, you can start performing trend analysis. Using SQL Server to do so lets you efficiently store months and months of performance data, which you can then export in bulk for trending purposes.

I use the DTS Import/Export Wizard to export all my collected performance data into an Excel spreadsheet. The wizard makes it easy to simply copy months of performance data into a single Excel spreadsheet, and Excel makes it easy to instantly generate charts from that data. If you want to analyze trends for a specific period of time, you can specify a query in the Import/Export Wizard to filter the data that is exported into the Excel spreadsheet.

☞ Save your DTS packages! After you've used the wizard to create a DTS package that exports your performance data to an Excel spreadsheet, save the package. You'll be able to easily run it again in the future to re-export your data, including the most recent additions. You can even use SQL Server Agent to schedule regular exports: I set up Agent to automatically export my performance data at the end of each month, during the evening. When I come to work the next day, my spreadsheet is all ready for me to analyze for my monthly trend update.

After the data is in Excel, select the entire range, as Figure 3.6 shows (note that I'm working with a small set of data for illustrative purposes; you'll likely be working with a couple of dozen columns of performance data).
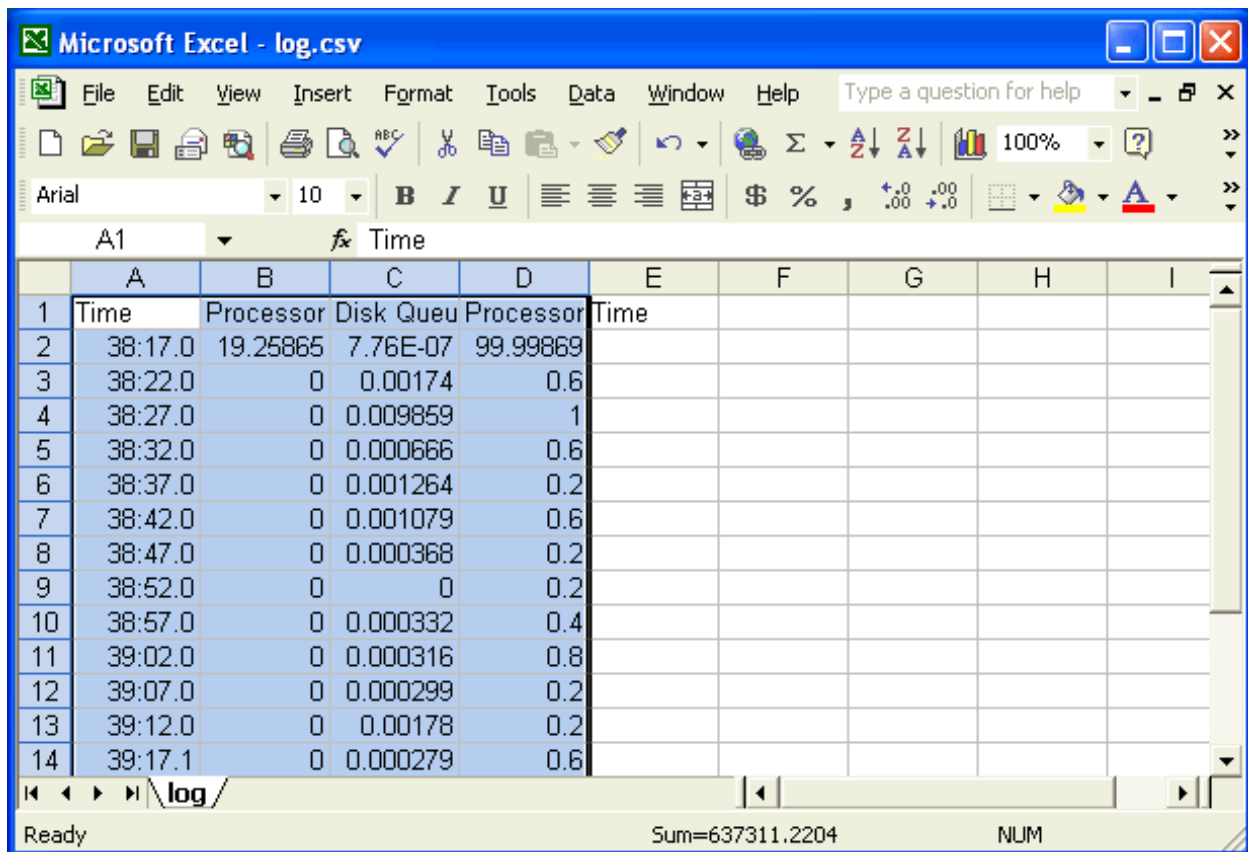
Figure 3.6: Be sure to select all the data in the spreadsheet to generate an accurate chart.

After you've selected the data, launch the Chart Wizard, either from the toolbar or by selecting Chart from the Insert menu. The first thing that the wizard will need to know is the type of chart you want. I find that line charts are the easiest to decipher for trend analysis, and Excel offers a variety of line chart types.

☞ Preview your chart before you make it. When you're selecting a chart type, Excel lets you quickly preview your data in the selected type. Simply click the Press and Hold to View Sample button in the chart type window to see a miniature version of the chart you're creating. The preview lets you ensure that the chart type you're selecting is appropriate for your data.

The wizard's next screen lets you select your data range, which should already be filled in based on the data that you selected in the spreadsheet. The third screen lets you dress up your chart by adding labels, modifying colors, and so on. Finally, the fourth screen lets you choose where your new chart will go. You can either have Excel create your new chart on a new sheet within the file or as a floating object on an existing sheet, as Figure 3.7 shows.
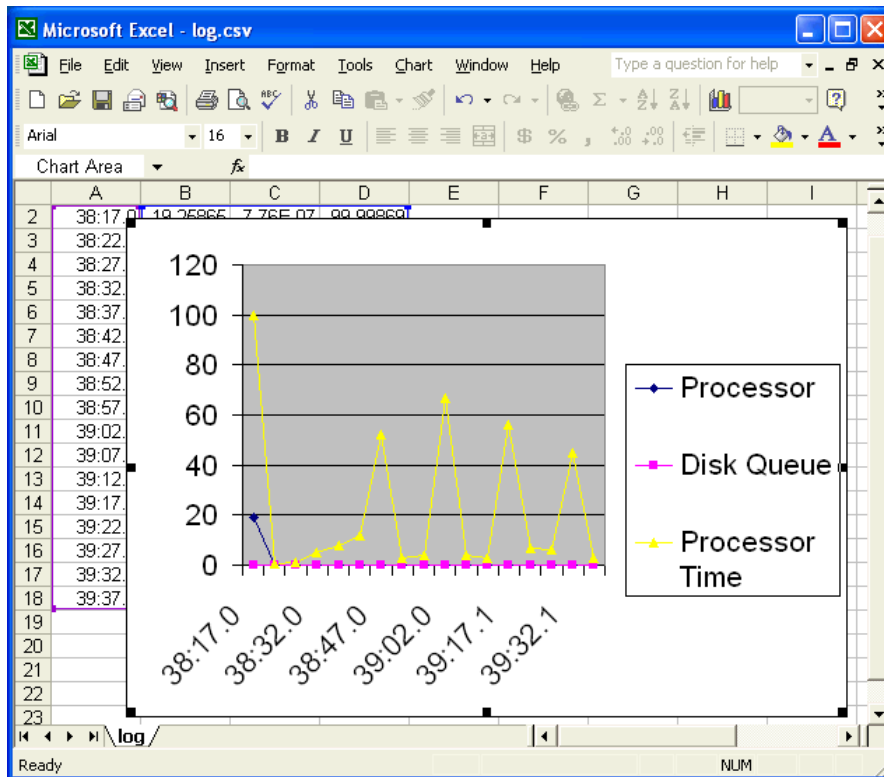
Figure 3.7: This chart looks a lot like System Monitor's chart view, but displays data for weeks, months, or years rather than just the past few minutes.

With a result that looks much like System Monitor's chart view, you might be wondering why you went through all this trouble. Keep in mind that your Excel chart may include months and months of performance data, making it very easy to spot trends such as a decreasing amount of available memory or increasing average processor utilization. More important, however, is Excel's ability to help you predict future performance. Don't forget that Excel is more than just a fancy calculator! It has some heavy-duty statistical analysis tools that are invaluable in trend analysis. I am emphatically not a math whiz, so I tend to use fairly straightforward techniques such as Excel's Trend function. If you're a bit more comfortable with statistics or can borrow a math whiz to set up a spreadsheet for you, Excel offers a world of statistical possibilities. To perform basic trending in Excel

1. Make sure you have a chart up in Excel based on your current data.

2. In the chart, select one of the lines. Remember that these lines each represent a particular performance counter. Right-click the line, and select Add Trendline from the pop-up menu.

3. I tend to stick with the Linear trend type because it makes the most sense to me. The Linear trendline simply extends your current performance trends into the future. Other trendlines offer more complex options for predicting the future:

- Polynomial trends tend to account better for wildly variant performance over time by calculating the least fit through your existing data points. Use this trend type if your past performance data doesn't seem to indicate a clear trend.

- Exponential trends work best when your performance is growing at, well, an exponential rate. In other words, if your growth percentage is a little higher from week to week or month to month, a logarithmic trend will be more effective.

4. The Options tab lets you decide how far into the future you want to predict by changing the Forward Periods entry. Keep in mind that the further ahead you predict, the less accurate the prediction will probably be.

5. Click OK to add the trendline. As Figure 3.8 illustrates, I can expect my overall processor utilization to actually decline in the future, according to the trendline. This outcome is a good example of why you need to collect data from your production servers and not your test servers. You also need to make sure that the data you collect is representative of your servers' usual performance, which is why I advocate capturing performance logs primarily during the workday when server load tends to be heaviest.
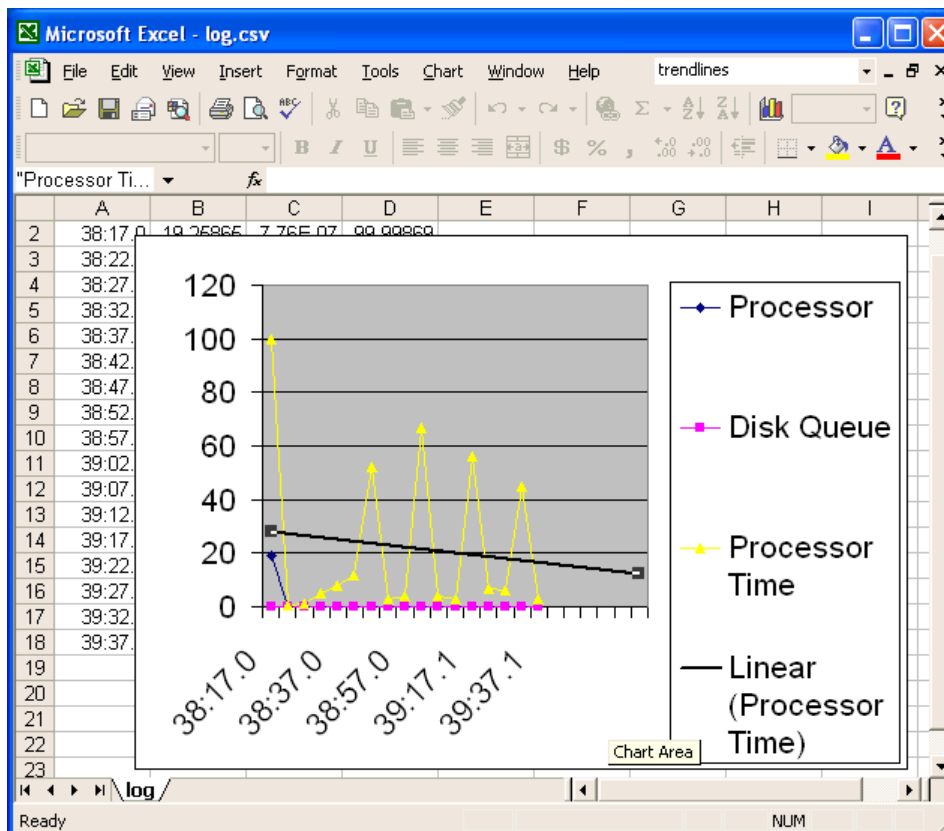
Figure 3.8: You can add trendlines for each of the performance counters in your chart.

The practical upshot of this entire trending exercise is to have Excel clearly point out when performance is likely to become unacceptable. If you've drawn an imaginary line on, say, processor performance, you might not want processor utilization to exceed 75 percent. Excel's trendline can show you exactly when that's likely to occur based on *regular* data collection and analysis.

Lather, Rinse, Repeat!

The most important factor in any trending analysis isn't how you do it; it's how *often* you do it. I collect full data twice a week. On Mondays, which tend to be the heaviest workload days, I collect a full set of data for the entire workday from about 7AM to 7PM. On Thursdays, which are a little slower, I collect data for 24 hours, using System Monitor's ability to automatically stop the log on the following Friday. I use the Monday log data for my main trend analysis, because that represents my worst-case performance. I use the Thursday data for planning purposes, because it helps me spot idle periods that are ideal for running maintenance jobs. The Thursday data also lets me keep an eye on evening performance, which is a factor because I run a lot of long-running report generation jobs, data warehouse loads, and so forth during the evenings. In the remainder of this chapter, I'll focus on specific types of bottlenecks that can plague SQL Server computers, show you how to detect them, and offer some advice for eliminating them.

Identifying Bottlenecks

It's rare that a server is at maximum capacity in every respect, and so performance optimization nearly always comes down to bottlenecks. In performance lingo, a *bottleneck* is simply the one thing that's slowing you down, and if you could just fix it, everything would run faster—until, of course, you hit the next bottleneck. I'll discuss three basic types of bottlenecks in the next sections:

- Hardware bottlenecks, which include processor, memory, and disk bottlenecks.

- What I call software bottlenecks, which relate primarily to the OS or other configuration issues.

- Database bottlenecks, which are cause by particular programming practices, database designs, and so forth.

For each type of bottleneck, I'll point out specific ways that you can detect them and provide tips for working around them. Expect to spend a lot of time in System Monitor, because that's where Windows shows you the data that points to bottlenecks.

> ☞ Finding bottlenecks can be a royal pain because every subsystem within a server—memory, processor, and so forth—affects the others. For example, you might think you have an over-utilized processor, but adding another one simply highlights the fact that your server doesn't have enough memory.
>
> An emerging class of third-party applications, including VERITAS Indepth for SQL Server, Microsoft Operations Manager (MOM), and NetIQ AppManager for SQL Server, are designed to detect bottleneck conditions automatically and alert you to them. They each work a bit differently, so if you're interested in using them, obtain evaluation versions to test in your environment.

Hardware Bottlenecks

Hardware, including the processor, memory, hard disk, and network, are the most monitored and most blamed components when it comes to performance monitoring. They're also among the easiest server components to upgrade, in most cases, and they definitely make it easier to quantify the improvements from an upgrade. When you add memory to a server, for example, you don't usually even need System Monitor to detect performance improvements.

Processor Performance

Processor performance tends to be the ultimate bottleneck. After all, very few servers accept significant processor upgrades, if any, and most servers max out at a total of four processors. Adding multiple processors isn't always an easy win, either; not all applications can take full advantage of multiple processors. SQL Server scales well across multiple processors, which means that, all other factors being equal, SQL Server will tend to use all processors in the server equally, rather than concentrating a certain amount of workload on one processor.

The obvious favorite for monitoring processor performance is the Processor:% Processor Time performance counter. System Monitor provides an instance for each installed processor and a _Total instance that averages all processors' performance. Unless you're just interested in seeing how each processor is being used relative to the others in a multiprocessor system, focus on the _Total instance, because it provides the best average view of your system's processor utilization.

💣 Keep an eye on processor zero. Processor zero—the first processor in your system—is likely to be the busiest in a multiprocessor system. The reason is that certain Windows OS processes have an *affinity,* or preference, for processor zero. Although your system's total processor utilization may be fine, if processor zero maxes out for long periods of time, Windows isn't operating as efficiently as it could, which will lead to SQL Server performance concerns as well.

Another counter to watch for total processor utilization is System:% Total Processor Time, which is essentially the same as the _Total instance of Processor:% Processor Time. If these counters occasionally spike to 100 percent, don't worry. In fact, I'd wonder what was wrong if they *didn't* spike because their processing power is there to be used, and SQL Server will take advantage of it. However, if you find continued utilization—10 to 15 minutes or more—in excess of 75 to 80 percent, you've got a processor utilization problem. After utilization reaches 80 percent or so, processor efficiency starts to decrease pretty drastically as the processor no longer has time to devote to the task of managing itself effectively *and* running applications.

I also like to keep an eye on System:Processor Queue Length. This counter indicates the number of threads waiting for execution on the processor and should never exceed one or two per processor for a prolonged period of time (10 minutes or more). The queue length is system wide, so a system with two processors can have a queue length of as many as four without giving you cause for alarm. If the queue length is longer, an excess number of threads are waiting for execution for too long a period of time, and you have a processor problem.

Generally, you'll see the queue length max out only when processor utilization is also very high. That's because the processor is simply working too hard, and you either need more processors or faster ones. However, if you've configured SQL Server to use a large number of worker threads, it's possible to see a long System:Processor Queue Length counter with fairly low processor utilization. The fix, of course, is to reconfigure SQL Server to use fewer worker threads so that the processor can keep up with them all.

I realize that it's all well and good for me to say get more processors or get faster processors, but the reality, of course, is that you can't always rush out and buy a couple of new Pentiums for your servers. The best technique is to buy the best processors you can when shopping for a new server:

- Get the fastest server-specific processors you can. Intel's Xeon family, for example, is specifically designed for server applications.

- Get the latest processor family available. For example, if you're shopping for 64-bit Itanium servers, hold out for Itanium 2s rather than the original Itaniums. Each new processor family generally introduces significant improvements in parallel processing and multiprocessing, even if the raw clock speeds of the processors don't go up much.

- Buy the largest L2 (or, for Itaniums, L3) cache you can get. Processor price is often more a function of cache size than clock speed, so expect to pay more for a larger cache. The performance benefits, however, are well worth it.

- All other considerations being equal, go for the fastest clock speed. Be aware, though, that cache size and processor family is usually more important. For example, the first Itanium 2 processors run at about 1GHz, less than half the speed of the fastest Pentium Xeon processors. Clock speed isn't always telling, however, as the Itanium's 64-bit architecture and parallel processing improvements make it a blindingly fast processor; faster even that the higher-speed Xeons.

You can also enable Windows fibers, which SQL Server 7.0 and SQL Server 2000 can use to improve their multiprocessor performance. For more information about fibers and their benefits, consult the SQL Server Books Online.

Memory Performance

One of the biggest mistakes you can make, from a performance standpoint, is running SQL Server with a bevy of other server products. SQL Server works best when it's running by itself, and my tips will assume that you're running it that way. If you aren't, you'll need to take special precautions to ensure that my tips won't mess up the other products running along with SQL Server.

SQL Server absolutely loves memory. In most cases, maxing out server memory is the quickest, easiest fix to many performance problems. Adding memory can actually mask many other problems, such as a slow disk subsystem, and after you've maxed out on memory, those other problems may become apparent. Even so, I'm a huge advocate of giving SQL Server as much RAM as you possibly can, then monitoring that memory usage to make sure it's being used efficiently.

The various memory utilization counters, such as Memory:Available Bytes, are practically useless for performance tuning. Sure, it's interesting to see the graph fall off, but the fact is that SQL Server will grab a lot of memory for itself even if it doesn't need to immediately use that memory. What you really want to pay attention to is the Memory:Pages/Sec counter. Under perfect conditions, you'll see activity on that counter, but it should never be sustained activity. In other words, you should expect to see little peaks and valleys, but never a straight horizontal line. The reason is that Windows allocates a full 2GB (3GB under Advanced Server editions) to all running applications. If the server doesn't actually have that much memory, Windows makes up

for it by using its page file, also called a swap file, to create *virtual memory*. When applications need to access memory locations that are stored in the swap file, Windows *swaps* the needed data into physical memory, and saves an unneeded portion of physical memory into the swap file to make room. If SQL Server is running by itself, all of its allocated memory will be mapped to physical memory, and very little swapping will occur. Swapping is obviously very bad for performance, because the application must suspend execution while Windows reads data from disk.

Another cause for swapping is manually configuring SQL Server's memory space. By default, SQL Server is set to dynamically configure memory as needed, and will do so in a way that avoids page swaps as much as possible. If you statically configure SQL Server memory, you might inadvertently be causing page swaps. If you notice the Memory:Pages/Sec counter showing sustained page swaps, reconfigure SQL Server for dynamic memory management and see if that helps matters.

SQL Server generally tries to leave about 4MB to 10MB of physical RAM free, which means it won't allocate more memory for itself if it means leaving less free. If you watch the Memory:Available Bytes counter while SQL Server is running, you should always see at least 5MB available; if you don't, you need to add RAM to your server. If you've already got a full 4GB installed, and SQL Server *still* isn't getting enough RAM, you've got extra applications running and you should think about shutting them down or moving them to a different machine. "Extra applications" include OS activities, such as DHCP server services, DNS server services, domain controller functionality, and so forth. SQL Server's ideal platform is a standalone or member server that is running no other services.

Disk Performance

Ideally, SQL Server should receive immediate responses to its requests for disk reads and writes. In reality, disk subsystems are often the biggest bottleneck in any SQL Server environment because of the inherently slow nature of physical disks. The easiest way to determine whether SQL Server is getting a good disk response time is the PhysicalDisk:Avg. Disk Queue Length counter, which shows the number of requests that the disk controller has waiting to be processed. This counter should never exceed two *per disk* for an extended period of time, although it may stack up to four or five for very brief peaks on a busy server. For example, if you have an array of eight disks, a queue length of 16 is not unacceptable, and you may see peaks of 20 or more. This math only applies to disks that Windows sees as physically separate, such as software RAID arrays. Hardware-based arrays usually appear as a single physical disk to Windows, so you should see queue lengths of two or less.

You can also check the PhysicalDisk:% Disk Time counter, which shows how busy the disk is. On a busy server, this counter may be as high as 60 percent for sustained periods of time, but should never exceed that figure for more than 10 minutes or so. If it does, check the more specific PhysicalDisk:% Disk Read Time and PhysicalDisk:% Disk Write Time to find out whether read or write operations are causing the majority of the disk utilization.

You're almost guaranteed to experience a disk bottleneck during your career as a DBA. What can you do to resolve them? One solution is to get faster disks. High-quality drives with rotation speeds of 10,000rpm and more are available; the faster the drive spins, the faster it can stream data to your server. Also make sure that you're using a fast disk channel to move data from the disks to the disk controller. Newer SCSI controllers support speeds of 40MBps or more. You may also investigate Storage Area Networks (SANs), especially ones that connect to your servers through high-speed fiber-optic cabling.

Also use a bus-mastering PCI disk controller with lots of onboard cache memory; these controllers can establish a direct channel to the server's main system memory, offloading some work from the server's processors. In addition, the onboard cache helps get data to the server much more quickly. Onboard write caching can be a big help to, as it allows the controller to report an instant successful write when SQL Server saves data to the disk, then physically write the data to disk when it has time. Quality disk controllers should include a battery backup for this cache memory so that a sudden power outage will not result in lost data; the card should dump all pending data to disk when power is restored to the server.

After the components of your disk subsystem are running as fast as possible, you can add more disks to improve throughput. (Another solution is to add RAM, as the sidebar "Need Faster Disks? Add RAM" explains.) The more disks you include in a RAID 5 or RAID 10 array, for example, the faster your throughput will be. High-end storage devices from companies such as EMC can include dozens of drives in an array, creating very high-speed throughput.

Need Faster Disks? Add RAM

One innovative approach to improving disk throughput is to use RAM instead. After all, RAM is hundreds of times faster than even the fastest physical disk, although RAM is obviously pretty expensive stuff compared with disks. SQL Server definitely buys into the notion of using RAM whenever possible; in fact, it's the whole point of SQL Server's various complex caching systems: Keep data in memory whenever possible. Of course, we're limited by the memory addressing architecture of our servers. Most servers are capable of addressing only 4GB of RAM, placing a firm upper limit on how much of the stuff SQL Server can get.

The innovative bit about using RAM is to install it as a hard disk. Imperial Technologies (http://www.imperialtech.com) has a product called MegaRAM, which is essentially a hard disk composed of memory chips instead of platters and read/write heads. MegaRAM units are installed like any other hard drive, and the server recognizes them as drives, assigning them a drive letter and so on. By storing SQL Server databases on these devices, you're effectively caching the entire database, speeding up access quite a bit.

Of course, throughput is still limited by the bus that connects the drives to the computer. Even the fastest SCSI channels (including the 40MBps SCSI with which MegaRAM is compatible) don't provide the fast throughput of a server's main memory subsystem; however, devices such as MegaRAM at least eliminate the physical delays associated with hard drives: Seeking the head to a specific sector, dealing with fragmented files, and so forth.

Other innovative uses of solid-state memory (RAM) include cache accelerators, which transparently cache data from disk so that it's available when the server requests it. Many high-end RAID adapters include substantial cache memory for this purpose, often reading ahead on the disk to pre-cache memory that the server is expected to request soon. All of these solutions can go a long way toward improving disk-based bottlenecks in busy SQL Server computers.

You have several options for working around a disk subsystem problem. These don't directly address the problem, but instead build up other subsystems to compensate. For example, I've already explained how more system RAM can reduce the effects of a slow disk subsystem by allowing SQL Server to cache data in fast RAM instead of loading it from disk every time.

You can also fine tune your application so that it doesn't need to pound the disks so heavily. Writing queries that return less data will help reduce read activity, for example. If you tend to work with sequential data (for example, querying customers in alphabetical order), use a clustered index to physically sort the data in the order that you tend to query it. Remember that SQL Server reads data from the disk in pages of 8KB; if those pages can stream into memory containing the data SQL Server needs, SQL Server won't have to perform read operations as frenetically.

Write operations are more difficult to fine tune. SQL Server writes data the same way it reads it—in pages of 8KB. So you might think that changing a single column of data is less strenuous than changing an entire row, but such is not the case. SQL Server must read the entire page containing the row, change the one column in memory, then write the entire changed page back to disk, resulting in the same number of disk operations as if you'd changed the entire row. Almost all you can do is rely on the fact that SQL Server doesn't *immediately* save changed pages to disk; if you need to update an entire sequence of data—say a group of customers—try to do so in the sequence that they're stored on disk, which will usually be sorted by their clustered index or primary key. Doing so will help SQL Server read data pages once, make all of the necessary changes to all rows contained in that page, then write the page once.

> 📖 I'll delve into more detail about query tuning in Chapters 4 and 5.

Network Performance

I know a lot of administrators who tend to ignore network performance when they're fine tuning SQL Server; a bad decision because the whole point of SQL Server is to deliver data across a network, so the network becomes a pretty important player in SQL Server performance. Unfortunately, fine tuning network performance for SQL Server can be pretty complex.

Start by watching the Network Interface:Bytes Total/Sec counter. This counter shows the number of bytes being sent and received by the server; if the server is only running SQL Server, most of the traffic you see will belong to SQL Server. On an Ethernet network, the traffic should always be less than about 75 to 80 percent of the network's maximum capacity; Ethernet networks utilize shared network bandwidth and become very inefficient at prolonged utilization of 80 percent or more. For deterministic networks, such as Token Ring or Fiber Distributed Data Interface (FDDI), utilization can safely approach 85 to 90 percent before you need to worry.

What do you do when network performance falls off? There are several possible approaches:

- Upgrade to a faster network—This solution isn't always easy, of course, as it means upgrading hubs, routers, network adapters, and sometimes cabling, but it provides immediate relief from network congestion.

- Use a switched network—Switches help reserve network bandwidth on an Ethernet network by breaking up shared bandwidth (also called *collision domains*). Switches let more devices fully utilize their network connection rather than fighting each other for bandwidth.

VERITAS™

- Switch to a deterministic networking architecture—The biggest inefficiency in Ethernet is collisions, which result when two devices try to communicate at the same time on the same network segment. Ethernet networks are similar to a room full of people: The more people in the room, the more likely it is that two people will try to talk at once and drown one another out. Deterministic networks such as FDDI *tell* each device when it may use the network, thus eliminating collisions and improving the network's efficiency. FDDI is commonly used as a server backbone network, although the required adapters, switches, hubs, and fiber-optic cabling can be horribly expensive.

- Tune your application to reduce network data—Query as little data as possible, for example, so that data sent by SQL Server can be sent in the shortest time possible.

- Reduce the use of encrypted networking protocols as much as possible while maintaining your security requirements—Encryption can double or triple the size of data sent across the network, not to mention the additional processing overhead required to encrypt and decrypt data.

- Try to use only a single networking protocol, such as TCP/IP—Adding protocols, such as IPX/SPX, will impose additional network overhead, effectively wasting bandwidth with duplicated router communications, service advertisements, Windows browse master traffic, and so forth. Also, configure SQL Server's default network library to correspond with your network protocol so that the correct library will be selected without SQL Server having to try others first.

Software Bottlenecks

SQL Server can create a bottleneck on its own performance, depending on how you've configured it. SQL Server's performance counters can also help you detect hardware and other bottlenecks more easily, through those bottlenecks' effect on SQL Server.

Getting back to disk performance for a moment—the most common cause for wasted disk throughput on a SQL Server is page splitting. Page splits occur when an index or data page becomes full, and SQL Server must create a new page to contain data. You can think of page splits as a kind of fragmentation: Page 1 contains entries A through B, Page 2 contains C through D, and Page 3 contains some additional entries for B that wouldn't fit on Page 1. Page splits slow read performance and impose a significant write hit when the split actually occurs. You can watch the SQL Server Access Methods:Page Splits/sec counter to see how many page splits are occurring. The value should be as low as possible, and you can help achieve a low value by configuring your indexes with the appropriate fill factor, which leaves empty room on index pages to accommodate data without having to split pages.

SQL Server has to allocate a small portion of memory and processing power to maintain each connection to the server. You can use the SQL Server General Statistics:User Connections counter to see how many connections SQL Server is currently maintaining. Keep in mind that a single user (human being) can generate multiple connections. For example, if you open five windows in Query Analyzer, that's at least five connections to SQL Server. If your SQL Server computers are dealing with a tremendous number of connections, you can improve their performance by implementing a multi-tier application architecture. For example, in a three-tier architecture, users' applications connect to middle-tier servers, which often run COM+ components. The COM+ components, in turn, connect to SQL Server. The components can pool connections, servicing a large number of users with fairly fewer connections, and thus reducing the overhead imposed on SQL Server itself.

Another valuable counter is SQL Server Buffer Manager:Buffer Cache Hit Ratio. This counter shows the ratio of data requests that are filled from SQL Server's in-memory cache rather than from disk. You want this number to be as high as possible because disk access is always slower than cache access. You can help increase this number by giving SQL Server more memory to work with. A reasonable number to shoot for is a 95 percent hit ratio, with only 5 percent of your data access coming directly from disk (or transactional applications; data warehouses will inherently have a low hit ratio).

Database Bottlenecks

As I'll discuss in Chapters 4 and 5, table scans are the bane of every DBA's existence. Although you'll probably never eliminate them, they are the most inefficient way for SQL Server to locate data. Generally, table scans should be limited to small tables, as any large tables should include well-maintained indexes that make table scans unnecessary. Watch the SQL Server Access Methods:Full Scans/sec counter to see how often SQL Server is performing table scans. If the counter shows a sustained high value, use SQL Profiler to determine which queries are causing the scans, then fine tune those queries to eliminate or reduce the occurrence of table scans.

> 📖 I covered SQL Profiler in Chapter 2. In Chapters 4 and 5, I'll show you how to analyze query execution plans to spot things such as table scans and how to fine tune your indexes to reduce or eliminate the scans as much as possible.

I also like to watch the SQL Server Locks:Average Wait Time (ms) counter. This counter shows how long, in milliseconds, processes are waiting to be granted an object lock in SQL Server. You can use different instances of the counter to monitor extent locks, key locks, page locks, database locks, table locks, and more. The ideal value for these is zero, meaning all locks are instantly granted; if you're seeing sustained high values, use SQL Profiler to help determine which process is holding locks for a long period of time, thus preventing other processes from getting their locks. Several other factors can create database performance bottlenecks:

- Over-normalization—Generally, third normal form is as far as you need to go, and you need to apply a healthy dose of what I call the *common-sense normal form*, which simply states that any normalization that significantly reduces performance isn't beneficial, no matter how elegant it makes your database.

- Over-querying—Your queries should only return the exact amount of data that you need. Don't query a whole bunch of data, then use client-side ADO features to filter the results; query just the data you actually plan to work with.

VERITAS™

- Temporary tables—They're useful, sure, but they're dead slow. Temporary tables are created in SQL Server's tempdb database, and generally create slow performance. As an alternative, consider using the table datatype, which resides in memory, or even creating a permanent table.

- Cursors—Cursors are worse, performance-wise, than temporary tables. SQL Server is designed to work with data in batches; cursors are designed to work with data one row at a time and simply don't perform well. Again, the table datatype can often be substituted for a cursor and is more efficient.

- If you can't help but use a temporary table, make it as small as possible and *never* use SELECT INTO to populate it—SELECT INTO places locks on several system objects and is extremely inefficient. Instead, use normal T-SQL CREATE TABLE statements to create the table, and use INSERT INTO to populate it.

- Write client applications so that they query and update data by using stored procedures— Stored procedures are pre-optimized and can execute a bit more quickly than ad-hoc queries. Even simple queries can be handled faster by SQL Server if executed within a stored procedure instead of submitted ad-hoc.

- Make sure that SQL Server has updated statistics for your databases—Statistics play an important role in how SQL Server executes queries, and outdated statistics may lead SQL Server to make poor execution decisions that result in performance hits.

- Keep your indexes well maintained—Databases with frequent updates will need their indexes rebuilt more frequently to prevent split pages. Use a low fill factor on especially busy tables to allow room for growth between index rebuilds and avoid page splits altogether.

- Keep transactions short—Transactions establish locks, and locks are a leading cause of poor performance. Long-running transactions should be broken into pieces, if possible, to reduce their performance hit on your server.

📖 I'll cover tips for finding and eliminating many of these database-specific performance problems in Chapters 4 and 5.

Server Bottlenecks

Sometimes you do everything you can to eliminate bottlenecks, and performance still isn't what you'd like it to be. Unfortunately, this situation is just a fact of life: A single server can only do so much. However, multiple servers can do much more.

Most of the tips I've provided so far in this chapter focus on *scaling up*: adding memory, getting faster disks, adding processors, and so forth. SQL Server is also perfectly capable of *scaling out*, which means using multiple SQL Server systems to handle a given workload.

📖 I'll discuss specific techniques for scaling out in Chapter 6.

VERITAS™

When is it time to call it quits on scaling up and start shopping for more servers?

- If you've maxed out on memory and are still getting low cache hit ratios in your transactional applications.

- If your quad-processor server is running all processors at an average 70 percent utilization, yet disk utilization still isn't maxed out.

- If disk utilization is maxed out, and you've made as many improvements as possible in terms of faster disks, bigger arrays, faster controllers, and so forth *and* you've reduced SQL Server's disk needs by reducing table scans, fixing page splits, and so forth. Obviously, disk utilization is infinitely tunable, but you'll reach a point at which anything else you do just doesn't buy much extra horsepower; that's the point at which you need to consider scaling out.

- If your network is already fine tuned and is still maxed out. Networks are among the least flexible subsystems to fine tune; they have a fixed capacity, and although you can reduce utilization by writing more efficient queries, you can only do so much. Sometimes adding another server *on a less-utilized network segment* (or dedicated switch port) is the only answer.

Try not to hurry to the "I need an additional server" conclusion; scaling out with SQL Server is definitely possible, but nowhere near fun and easy. SQL Server isn't like a Web farm for which you just plop a new server into place and get an instant performance benefit; SQL Server scaling out is complex and time-consuming. Do as much as you can to squeeze every erg of performance from your existing server before you decide that scaling out is the way to go.

And, of course, make sure that you're consistent in your trend analysis. If you are, you'll see the need to scale out coming from a long way off, giving you plenty of time to plan that major undertaking before it becomes necessary to the survival of your production environment.

Summary

In this chapter, you learned about the most common SQL Server bottlenecks, and how to detect them. You also learned some tips for fixing these bottlenecks when they occur, and tips for spotting bottlenecks *before* they occur. In addition, I introduced you to a very basic performance trending methodology, which uses inexpensive and freely available tools along with SQL Server to record past performance data and predict future performance using Excel's powerful statistical analysis tools.

Chapter 4: Understanding the Query Optimizer

SQL Server's query optimizer is designed to analyze each and every query that runs through SQL Server and decide how that query will be executed. The optimizer's output is called a *query execution plan,* sometimes referred to simply as an *execution plan* or a *query plan*. The optimizer usually determines several possible plans for any one query; its job is to select the plan that will execute the fastest, thereby producing results in the shortest possible amount of time.

Database Statistics

SQL Server's statistics, which SQL Server collects for databases, tables, and indexes, are key to the optimizer's operation. These statistics let the optimizer estimate how long various query operations will take to execute, then select those operations that will execute most quickly. For example, simply having an index on a particular column isn't a guarantee that the optimizer will choose to use the index. Consider the following query:

```
SELECT CustomerName FROM Customers WHERE CustomerName LIKE
'JONES%'
```

In a large database, an index on the CustomerName column would help SQL Server execute this query more quickly. The index stores an alphabetical list of customer names, and SQL Server can simply move right to the "Jones" customers and return them in the query results. However, in a small table—say, 10 rows or so—the process of opening and using the index would take more effort than simply scanning through the existing rows. The optimizer's job is to decide which would be faster, and it uses statistics—such as the size of the table—to make that decision. SQL Server maintains the following statistics:

- The time that the statistics were last updated. As databases grow and change, statistics can become outdated. If the statistics aren't up to date, the optimizer can't rely on them to make accurate execution decisions.

- The number of rows in each table and index. SQL Server uses this statistic to estimate the length of time particular operations will require, allowing the optimizer to weigh various possible execution plans against one another.

- The number of data pages occupied by a table or index. This statistic helps SQL Server determine how many disk-read operations will be required; disk reads can significantly affect an operation's efficiency.

- The number of rows used to produce the table's histogram and density information. A histogram is a set of as many as 200 values for a given column, essentially making the histogram a statistical sampling of that column's values. Density is the number of unique values within a given range of rows. Both of these pieces of information help SQL Server decide how useful an index will be. The more non-unique values within a column, the less useful an index tends to be. For example, a column that lists a person's gender won't be very dense because there are only two possible values. Indexes on that column would be pretty useless, as they would only let SQL Server quickly skip to either "male" or "female" in an index seek. From there, SQL Server would be left to plod through row by row to match other search criteria.

- The average length of the values in a table's primary index, or key. Longer values are more difficult to scan through; thus, an index, if one is available, is more likely to be faster.

Obviously, out-of-date statistics will cause the optimizer to make decisions that aren't effective for the database's current condition. You can configure SQL Server to automatically update statistics at regular intervals, based primarily on the database's rate of change. To do so, simply open Enterprise Manager, right-click the database, and select Properties from the menu. As Figure 4.1 shows, you'll see an *Auto create statistics* check box as well as an *Auto update statistics* check box—select both to have SQL Server handle statistics automatically.
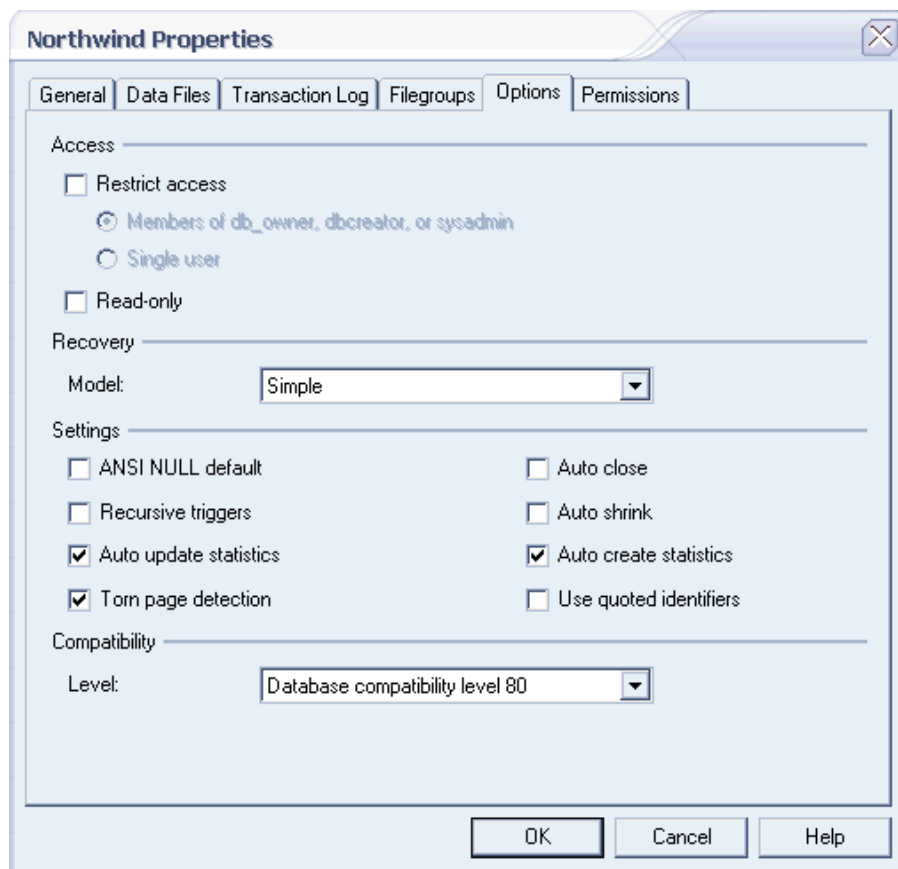
Figure 4.1: Configuring SQL Server to automatically update statistics for all objects in a database.

You can also have SQL Server update statistics on demand by using the UPDATE STATISTICS T-SQL command. You can run the command from Query Analyzer, specifying the name of the object—such as a table or index—to update. For example, the following command will update statistics for the Customers table:

```
UPDATE STATISTICS Customers
```

You can also execute the sp_updatestats system stored procedure to update all statistics for all objects in the current database.

✎ SQL Server 2000 maintains separate statistics for the rate of change in both tables and indexes, so SQL Server can update statistics only for those objects that need it the most. This behavior differs from earlier versions, in which statistics were maintained only for tables or columns, causing SQL Server to sometimes update statistics that weren't in need of an update. The more granular statistics tracking of SQL Server 2000 helps ensure that statistics updates only occur when necessary, reducing server overhead.

Perhaps the most important optimization technique that you, as a SQL Server administrator, can contribute is to ensure that your database objects have up-to-date statistics. These statistics represent the optimizer's only means of selecting an efficient query plan; outdated statistics will result in poor execution decisions and slow-running queries. Configure your databases to automatically update statistics for the best performance. If you're concerned about the overhead involved, you can manually update statistics, but make sure you do so frequently enough to ensure accurate data for the optimizer to work with.

When the Optimizer Runs

When the query optimizer receives a new query, it performs several steps to generate an execution plan. Microsoft designed this process so that the query optimizer produces an efficient plan within the fewest possible number of steps; the goal is to produce a *reasonably* efficient plan as opposed to the most efficient *possible* plan, which might take many more steps—and much more time—for the optimizer to determine. No matter which execution plan SQL Server selects, you can almost always theorize a more efficient plan. However, the process of coming up with a plan is fairly time consuming, so SQL Server tries to determine a *good* plan and start the query rather than wasting time trying to come up with the *perfect* plan for a particular query.

☞ One of the big benefits of using stored procedures is that SQL Server can store a stored procedure's execution plan in addition to the stored procedure. This capability means that the optimizer doesn't have to re-optimize stored procedures when they run; instead, the optimizer can use the stored plan to execute the procedure more efficiently.

Databases change over time, however, so Microsoft designed SQL Server to automatically recompile procedures' execution plans every so often to ensure that the stored plan is the most efficient plan possible. Nonetheless, stored procedures offer faster optimization—and therefore, faster execution—than an equivalent ad-hoc query.

Whenever the optimizer completes an execution plan, the optimizer caches the plan. SQL Server's execution plan cache can store a good number of execution plans (depending on the amount of server memory installed). These cached plans allow SQL Server to treat ad-hoc queries similarly to stored procedures: If the optimizer finds a cached query plan, the optimizer can use the plan rather than generate a new one. Unfortunately, the optimizer can only match queries to stored plans if the query *exactly* matches the one that originally created the plan. In other words, suppose you submit the following query to SQL Server:

```
SELECT CustomerName FROM Customers WHERE CustomerName = 'SMITH'
```

VERITAS™

The optimizer will compile an execution plan for this query, cache the plan, and execute it to generate the query's results. If you then re-submit the *exact same query* to SQL Server, query compilation won't be necessary; the optimizer can use the cached plan because it matches the plan of the original query. However, if you then submit the following query:

```
SELECT CustomerName FROM Customers WHERE CustomerName = 'JOHNSON'
```

SQL Server will compile a new execution plan because the query doesn't exactly match one that is already in the cache.

> ☞ The optimizer's exact-match behavior is an example of why stored procedures are so much more efficient: By creating a *parameterized* stored procedure (a stored procedure that can accept input values that govern the procedure's behavior), you can allow SQL Server to pre-compile the procedure's execution plan. The procedure will execute more quickly because its execution plan is stored with it, and you can still pass in the customer name that you're looking for, making the query somewhat dynamic.

Assuming an execution plan isn't available in the cache, SQL Server begins the process of compiling an execution plan for your query. This process involves several distinct steps, which SQL Server always performs in the same order. Each step in the compilation process generates one or more execution plans, and each plan is assigned a specific *cost.* The cost is based on the statistics analyzed by the optimizer. For example, an execution plan requiring an index lookup of a large table might have a lower cost than an execution plan that scans through each row of the table. In the end, SQL Server goes with the cheapest (lowest cost) execution plan, and executes it to generate your query's results. The optimizer uses the following steps to look for an execution plan.

First, the optimizer looks to see whether any *trivial* plans are available. The optimizer looks at the cost of a few standard, commonly used execution plans to see whether any of them will solve the query. The idea is that one of these plans might not be the fastest, but if the optimizer can decide on one, it'll be faster just to take the plan and run with it, rather than spending more time looking for a better plan. One example is INSERT queries that add rows to the database by using the VALUES clause. These queries have only one possible execution, so there's no point in searching for an efficient plan.

If no trivial plan is available for a low-enough cost, SQL Server tries to *simplify* the query by performing *syntactic transformations* on the query. In essence, the optimizer tries to rearrange the clauses in the query to see whether the query can be simplified—similar to how you simplify an equation in algebra before you actually start to solve the equation. This simplification step doesn't result in any possible query plans; instead, it's a preliminary step to help the optimizer analyze the query more efficiently. At this point, SQL Server also loads statistics so that they are available to use in the cost-based optimizing process that follows.

The cost-based optimizer is the main portion of the optimizer's operation, and consists of three distinct phases. The first is a *transaction-processing phase,* in which the optimizer picks out plans for simple queries that are typically found in transaction-processing databases. These queries are more complex than those tried by the trivial-plan optimizer, and require SQL Server to calculate a more precise cost estimate for each plan. As with the trivial optimizer, however, SQL Server will automatically go with a plan if it finds one under a certain cost threshold. That threshold represents the optimizer's gauge of "good enough," and prevents the optimizer from wasting time trying to find a marginally better plan.

If SQL Server can't find a plan that is cheap enough during the transaction-processing phase, it moves on to the *QuickPlan* phase. This phase expands the search for an execution plan to include choices that are often useful for more complex queries. Most plans found in this phase involve a single index and a nested loop join (which I'll discuss later). As with the other phases, after a plan is found with a low-enough cost, SQL Server runs with the plan rather than continuing to look for a better plan.

The last phase of optimization is called *full optimization,* and involves the more brute-force method of simply comparing the cost of every possible execution plan and selecting the lowest cost plan. This phase of optimization includes a built-in time limit, and the optimizer will automatically run with the cheapest plan found so far when the clock expires. The optimizer also starts to consider the cost of generating a plan. When the cost of generating a plan starts to outweigh the cost savings of a plan, the optimizer quits and runs with the cheapest plan that it has found so far.

As you can see from this process, the optimizer works best with simple queries and takes more time to compile plans for more complex queries. This outcome is fairly commonsensical, but it means that you'll derive the most benefit from complex queries that you implement as stored procedures. Very simple queries might not execute much quicker as a stored procedure because they can be optimized in the trivial-optimization phase. However, more complex queries will take much longer to compile and will always benefit from stored procedures' ability to use pre-compiled plans.

Understanding Query Execution Plans

The execution plans generated by the query optimizer aren't the last word on how quickly your queries can execute. In fact, learning to read and understand these plans can be a major factor in database redesign and optimization. Basically, you just need to understand why the optimizer is selecting particular plans and what you can do to your database to make the optimizer select more efficient plans instead. Each execution plan consists of two or more distinct operations, which can include:

- Table scans—Table scans are the slowest way for SQL Server to look up data. A table scan forces SQL Server to search through each row in a table one at a time, looking for rows that match your query criteria. Imagine that your local phone book wasn't organized alphabetically: With the names listed in no particular order, a table scan is what you would have to do to find a person's phone number. Table scans aren't bad if only a small number of rows are involved; as I mentioned earlier, it's faster for SQL Server to scan through a dozen rows or so than to open up and utilize an index. But large table scans are the number one cause for poor query performance.

☞ Updated statistics are key! SQL Server decides between large and small operations based solely on statistics. Imagine that you create a new table and don't have SQL Server automatically updating statistics. Initially, you create statistics on the table, which will indicate a small table, and SQL Server will generally select a table scan.

Suppose that your table grows to 20,000 rows, and you've added an index or two to speed up queries. However, if you've never updated the table's statistics, SQL Server will still think the table is small, and will ignore your indexes when executing queries. These poor decisions, based on outdated statistics, are also a leading cause of poor query performance.

- Index seek—This type of operation uses a non-clustered index to look up table results. Non-clustered indexes are like the index in the back of a book—they provide a sorted view of the data, with a pointer to the actual data. Index seeks are generally accompanied by a table lookup operation (called a bookmark lookup), in which SQL Server uses the index's pointer to find the actual row of data within the table.

✎ Not all index seeks automatically require a bookmark lookup. *Covered indexes* don't require the additional bookmark lookup when used. A covered index is one that actually contains the data that the query needs. For example, suppose you have a table named Customers, with an indexed column named CustomerName. If you execute the query

```
SELECT CustomerName FROM Customers ORDER BY CustomerName
```

all the data retrieved by the query—the CustomerName column—is included in the non-clustered index, making the index a covered index. SQL Server will be able to satisfy the query based entirely on the index without using the actual table at all. Creating covered indexes for frequently used queries is a common optimization technique because a large number of bookmark lookups—SQL Server's only choice when a covered index isn't available—can add quite a bit of time to a query's execution.

- Clustered index scan—Clustered indexes are like a phone book—the data is sorted into a particular order. Clustered index scans (as opposed to seeks, which I'll discuss next) are no different from a table scan, and generally indicate a performance problem in a large table.

- Clustered index seek—Similar to an index seek, this operation runs through a table's clustered index to locate query results. Because the table's data is physically sorted according to the clustered index, this operation doesn't require a corresponding lookup operation, making it the fastest possible way to locate data within a table.

- Joins—SQL Server will include one or more join operations in a query plan when necessary. I'll discuss how these operations work later in this chapter.

✎ Plans SQL Server 2000 execution plans can also include a level of *parallelism* on multiprocessor servers. Parallelism is the ability to execute a single query on multiple processors, ideally reducing the amount of time the query takes to complete by running different bits of it at once. The decision to use parallelism is made after the optimizer considers the server's current workload. For example, on an especially busy server, selecting parallelism might require the query to wait for other processes to complete, resulting in a net increase in execution time rather than a decrease. For this reason, the optimizer makes the decision to use parallelism each time a query is executed, as the decision hinges on the precise level of workload that the server is experiencing at that moment.

There are a number of other query operations that can show up in an execution plan; I'll provide a complete list in Chapter 5. The operations in the previous list will help you get started in reading basic execution plans.

VERITAS™

Reading Basic Execution Plans

SQL Query Analyzer provides the ability to view graphical query execution plans, including estimated execution plans, as Figure 4.2 shows.
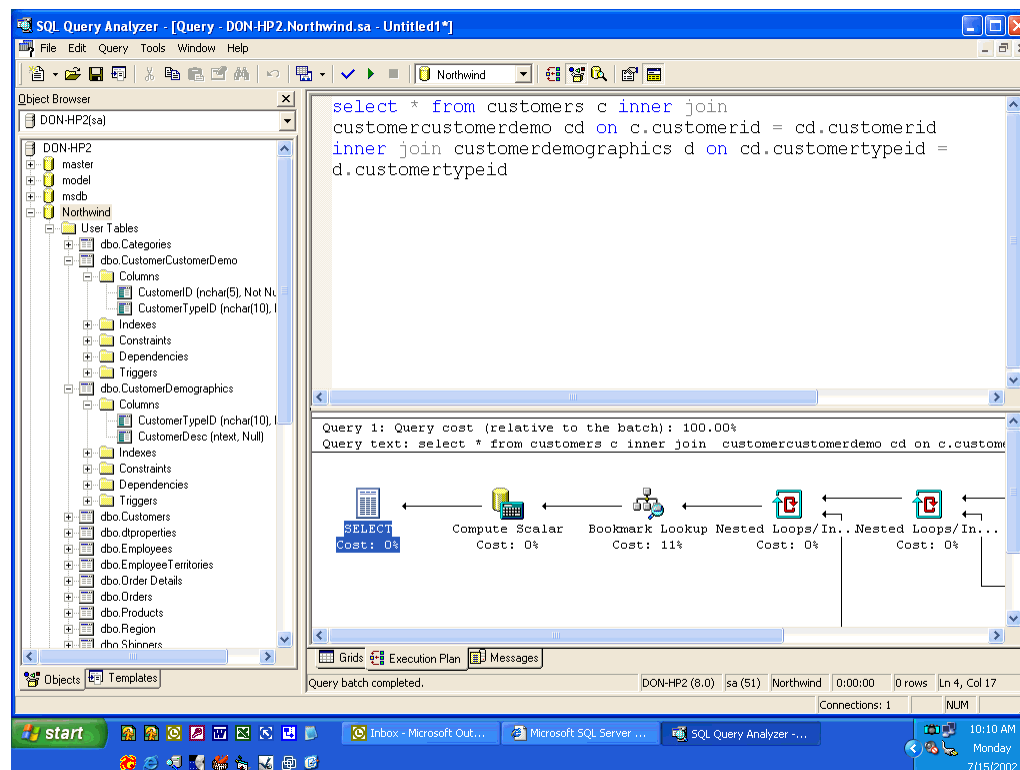
Figure 4.2: A graphical query execution plan displayed in SQL Query Analyzer.

📖 Really complex query execution plans are often easier to read in a non-graphical view. I'll show you how to view query plans this way in Chapter 5.

Reading these plans takes a bit of getting used to; they're read from right to left and from top to bottom. Each operation in the plan includes a cost percentage, which reflects that operation's relative cost to the other operations in the plan. Each operation is shown as an icon (there are about 50 possible icons) that represents the operation's purpose within the query. For queries with multiple statements—such as a batch or a query that contains a sub query—the graphical execution plan will include an execution plan for each query.

To display a query's execution plan, select Show Execution Plan from the Query menu in Query Analyzer. Execution plans are only displayed after the query executes. If you want to display an estimated plan without actually executing the query, select Display Estimated Execution Plan instead. This option immediately displays an execution plan, although it might not be exactly the same as the one SQL Server will use when the query is optimized and executed. The big difference, in fact, is that estimated plans don't take into account current server workload, which is often a factor in the costs assigned to execution plans by the query optimizer. For example, on a server with fairly low disk activity, the optimizer will select execution plans with higher disk costs; servers with high disk activity will result in execution plans that use the disk more efficiently. Estimated execution plans never take current activity into account.

VERITAS™

The following list provides some basic tips for reading graphical execution plans:

- As I previously mentioned, read from right to left and top to bottom.

- Operations requiring multiple steps are drawn in the same column. Lines with arrows connect operation nodes to their parent node.

- The thickness of the lines connecting the icons indicates the amount of data being transferred from step to step. A very thick line indicates a lot of data, which is inefficient; ideally, the first operations in a query (at the right of the diagram) should narrow the data set as much as possible so that subsequent operations can run against a smaller data set.

- You can move your cursor to hover over an icon to display statistics for that operation. Statistics include:

    - The physical operator used, such as a hash join or a nested loop. When a physical operator is displayed in red, it indicates that the optimizer is missing column statistics or other information, often resulting in a less efficient query.

> ☞ When an operation is displayed in red, you can right-click it and select Create Missing Statistics from the pop-up menu to have SQL Server automatically generate the statistics it was looking for. Of course, you should also review your database maintenance plan or the database's properties to ensure that statistics will continue to be updated as needed. Always re-run a query after creating (or updating) statistics, as the updated statistics will likely result in a different execution plan.

    - The logical operator used, such as a join operator.

    - The number of rows output by the operation.

    - The estimated size of the row output by the operation.

    - The estimated cost for the operation's input/output (I/O). Ideally, you want this value to be as low as possible, because I/O is the biggest contributor to poor query performance. You can reduce I/O cost by increasing server memory so that more data is in RAM than on slower disk storage.

    - The number of times the operation was executed. Operations executed several times indicate recursive operations, which are usually an inefficient way to perform a query.

    - The cost consumed by the optimizer to generate the execution plan. Plans that have a high optimization cost mean that your query is overly complex; try to reduce the query's complexity so that the optimizer can select an efficient query execution plan more quickly.

    - Any predicates and parameters used by the query. This information is provided for reference so that you can relate the operation to specific portions of your original query.

After you've displayed the plan, what do you do with it? Generally, you'll want to look for the operations with the highest cost in the plan, then think of ways to reduce their effect on the query's execution time. For example, table scan operations can often be improved by adding an index that would help SQL Server avoid the need for a table scan. You can use SQL Server's Index Tuning Wizard, which I'll discuss later in this chapter, to produce suggestions for indexes that would improve query performance.

Basic Query Performance Tuning Techniques

After you've got your query execution plan in hand, you can start trying to tune your queries. In the next few sections, I'll offer some basic advice for improving query performance. Chapter 5 will also focus on query tuning, using more advanced and esoteric techniques.

Easy Fixes

There are some tricks that will always help query performance:

- More RAM—SQL Server uses RAM in a big way, and adding memory improves the size of the data page cache, the cache used to store execution plans, and so forth.

- Multiple processors—SQL Server scales across multiple processors and can often make use of parallel execution to speed certain queries.

- Faster disks—The faster SQL Server can get data off of disk and into memory, the faster your queries will execute. Use large RAID arrays to provide fast read capabilities, and fast disk connections (such as the new Ultra SCSI or Fiber-Channel arrays) to stream data to the server as quickly as possible.

Of course, these are the same types of fixes that help almost any application running on Windows. SQL Server also benefits when it's the only product running on Windows so that it doesn't have to compete with other applications for system resources.

Keep in mind that there are certain types of queries that tend to run poorly no matter what you do. For example, any query returning a really large result set will run more slowly than a query returning a smaller result set, simply because SQL Server has to do more disk and memory operations to collect the results. Another culprit in slow queries are WHERE clauses with non-unique results. For example, consider the following query:

```
SELECT CustomerName FROM Customers WHERE Customer LIKE 'S%'
```

This query is just asking for trouble by returning all customers whose names begin with "S." Most customer databases will have a few thousand matching rows, making any available indexes less useful than they could be. Specifying more precise WHERE clauses will narrow the result set, make indexes more useful, and result in a faster-running query. Although adding faster disks, more RAM, and a faster processor (or more processors) will help a query such as this, these changes don't address the fact that it's a poorly written query because it doesn't let SQL Server leverage any of its techniques—such as indexes—that are designed to reduce query execution time.

Read Those Execution Plans

High-cost query operations are the biggest hit to query performance, and execution plans can help you easily identify them. Use the graphical execution plan to look for high-cost operations, and try to find ways to simplify them. Here are some tips for simplification:

- Query execution plan operations listed in red are a major warning sign: Get your statistics in place so that the optimizer has the information it needs to make accurate optimization decisions.

- Costly scan operations should take place only on smaller tables. Scans across large tables are inefficient and will generally benefit from the addition of an index.

- Complex queries might be better off if broken into smaller queries within a stored procedure. The results from one query can be saved into a table variable and used within the next query. Keep in mind that the optimizer can often compile multiple smaller queries more efficiently than one giant query.

- Given the previous point, try to avoid the use of temporary tables to store query results within a stored procedure. Temporary tables will usually result in slower execution, and they make it more difficult for the optimizer to come up with an efficient execution plan.

- The more joins you have in a query, the longer it will take to optimize and execute. Consider denormalizing tables if your queries start to contain seven or more join conditions; more than about seven joins in a query will usually ensure that the optimizer has to resort to brute-force optimization, which is very inefficient.

- Operations with a high I/O cost might be trying to work with too much data. Try rewriting the query to narrow the result set or add indexes that can help the optimizer quickly eliminate a large amount of data, if possible. Reading data from disk is very inefficient, so the more precise your result set, the less disk I/O you'll have, and the faster the query will run.

- A large number of bookmark lookups can be eliminated by creating additional covered indexes so that SQL Server has all the data needed to fulfill the query right within the index. Bookmark lookups aren't inherently bad, but if your query execution plans include many of them, covered indexes can help improve performance.

Rewrite Queries

Some queries just need rethinking to run more efficiently. For example, queries that utilize cursors can be extremely inefficient. At the very least, rewrite such queries to use efficient cursor types, such as a fast forward-only cursor, or rewrite the query to eliminate cursors and perform a single query operation instead. Avoiding cursors entirely will result in an instant performance boost to your queries.

In the past, developers often wrote queries that aliased a table to produce index intersections. For example, suppose you have an orders database that includes a table for order line items. That table might include a column for shipping date and a column for the line item ID. If both columns are indexed, you can produce a faster query by using an *index intersection*, effectively combining the two indexes to quickly eliminate non-matching rows from the result set. In the past, this technique required a somewhat complex query language that aliased the table multiple times within the query; SQL Server 2000 automatically detects and takes advantages of index intersections. Such a query should now be written entirely without table aliases:

```
SELECT CustomerID FROM LineItems WHERE LineID BETWEEN 10000 AND
10200 AND ShippingDate BETWEEN '9/1/2002' AND '10/1/2002'
```

Assuming both columns—LineID and ShippingDate—include an index, SQL Server will utilize both indexes to quickly eliminate all rows not meeting the double search criteria.

Limiting Query Run Time

On especially busy SQL Server computers, you can conserve resources by limiting how long queries can run. SQL Server includes a query governor configuration setting, which lets you prevent long-running queries from executing. By default, this setting is off, allowing all queries to execute; you can execute the SET QUERY_GOVERNOR_COST_LIMIT statement from Query Analyzer to configure the server to prevent long-running queries. Doing so can prevent long-running queries from consuming resources that could otherwise be used to execute other queries on the server.

> 🖉 Some DBAs use SQL Server Agent to turn on the query governor during the days and to turn off the query governor at night when long-running reporting queries are run to generate evening reports. Using SQL Server Agent in this fashion can automatically customize SQL Server's performance to meet the demands of daytime processing, while allowing longer queries to run later, when demands upon the sever aren't so great.

To set this option, first turn it on:

```
SET QUERY_GOVERNOR_COST_LIMIT value
```

Where *value* is the maximum cost the governor will allow. Any query with a higher estimated execution cost will not be allowed to run. This command only lasts for the current connection; use the sp_configure system stored procedure (which includes a query governor cost limit option) to configure the query governor for the entire server. A cost value of zero will return to the default condition of allowing queries to run regardless of their cost.

Reducing Joins

One often-overlooked query optimization technique is to simply reduce the number of joins included in a query. Joins can be expensive operations and require SQL Server to load statistics for multiple tables and indexes to compile an execution plan. Queries with more than two joins are usually passed on to the optimizer's last phase of optimization, which is the least efficient phase.

Reducing joins is simply a matter of changing your database design. A well-normalized database nearly always involves joins when querying data; the goal of normalization is to reduce data redundancy and duplication by ensuring that data is represented as distinct entities. This goal can reduce the time is takes to add or update data in a database, but generally increases the amount of time it takes to query data from the database.

For example, consider the database diagram in Figure 4.3. This diagram represents a set of tables used to store street address information. The design is well-normalized, with almost no repeated data. Indeed, even small pieces of data such as the Street, Drive, Road, and other street name suffixes are stored in separate tables. A central table contains the unique portion of the street address—the street name and number—as well as the ID numbers for the lookup tables that specify the street's direction, name suffix, and so forth. You could almost consider this design to be *over-normalized* because retrieving a single street address requires several tables to be joined together.
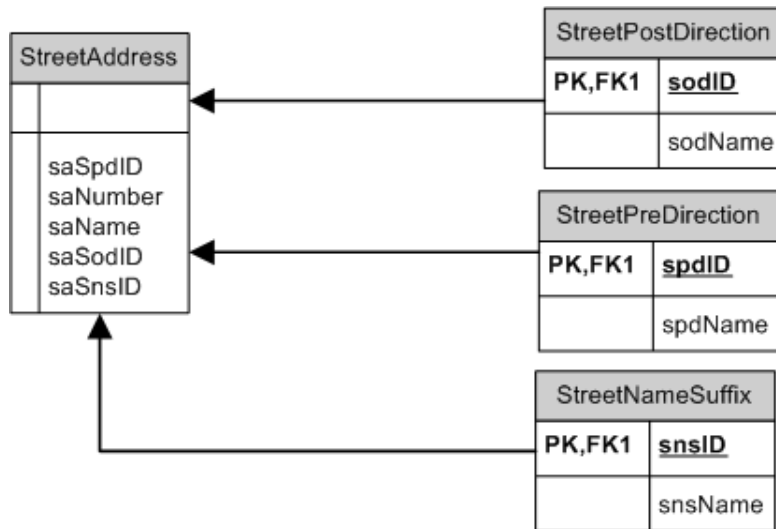
Figure 4.3: A well- (and perhaps over-) normalized database design.

Generally, the business reason behind this type of normalization is to ensure data accuracy. By providing lookup tables for values such as Street, Road, Circle, Drive, and so forth, the database's designer can ensure that only acceptable values will be entered into the database. Suppose that the street name suffix table contains the values that Table 4.1 shows.

SnsID	SnsName
1	Street
2	Drive
3	Circle
4	Parkway
5	Road
6	Way

Table 4.1: The street name suffix table.

Then the central table in the database might contain the following values (see Table 4.2).

saSpdID	SaNumber	SaName	saSnsID	saSodID
0	3600	Elm	1	0
0	1	Microsoft	6	0

Table 4.2: An example of the central table in the database.

The first row in the table represents 3600 Elm Street and the second represents 1 Microsoft Way; the saSpdID and saSodID columns would be used to represent optional street directions such as NW or West. Although this type of design definitely meets the rules of the various forms of normalization, it makes it tough on SQL Server to retrieve a particular street address, especially if you consider the fact that a production version of this table would also need to contain columns for the city, state, postal code, and so forth. A query to this type of table might look like the following query:

```
SELECT spdName, saNumber, saName, snsName, sodName FROM
StreetPreDirection INNER JOIN StreetAddress ON spdID = saSpdID
INNER JOIN StreetNameSuffix ON saSnsID = snsID INNER JOIN
StreetPostDirection ON sodID = saSodID
```

That's a pretty heavy-duty query, even before you start adding ORDER BY or WHERE clauses.

One way to reduce query times on this table while maintaining data consistency is to deliberately denormalize the table. You could use the same lookup tables to provide lists of acceptable values, but store the actual values within the main table, rather than ID numbers that reference the lookup tables. In this design, the central table's contents might look like that shown in Table 4.3.

SpdName	SaNumber	SaName	SnsName	sodName
NULL	3600	Elm	Street	NULL
NULL	1	Microsoft	Way	NULL

Table 4.3: An example of the central table's contents.

Client applications that modify the central table can be written to use the lookup tables to generate drop-down list boxes of acceptable values, but the values themselves, rather than cross-referencing ID numbers, are stored in the table. The new query is vastly less complex:

```
SELECT spdName, saNumber, saName, snsName, sodName FROM
StreetAddresses
```

Even if you didn't use any indexes, this query would execute more quickly simply because fewer disk operations and joins would be required.

☞ In this set of examples, I've used a naming convention that prefixes each column name with an abbreviation of the table name. This technique ensures that column names are always unique within a table and eliminates the need to alias table names within queries.

Also notice that my SELECT statements include a full list of the columns that the query should return. Even if you want the query to return all available columns, avoid using the shortcut SELECT *. When you use the * operator, SQL Server must internally build a column list that includes all columns. Having SQL Server do so is slower than if you just come right out and specify the columns yourself.

Using the Index Tuning Wizard

Query Analyzer also provides access to the Index Tuning Wizard. Simply press Ctrl + I to activate it, or select Index Tuning Wizard from the Query menu. The Index Tuning Wizard can provide suggestions to create the proper indexes for the queries that you run. For example, consider the execution plan that Figure 4.4 illustrates. This plan shows three table scan operations, which are fairly high-cost operations.
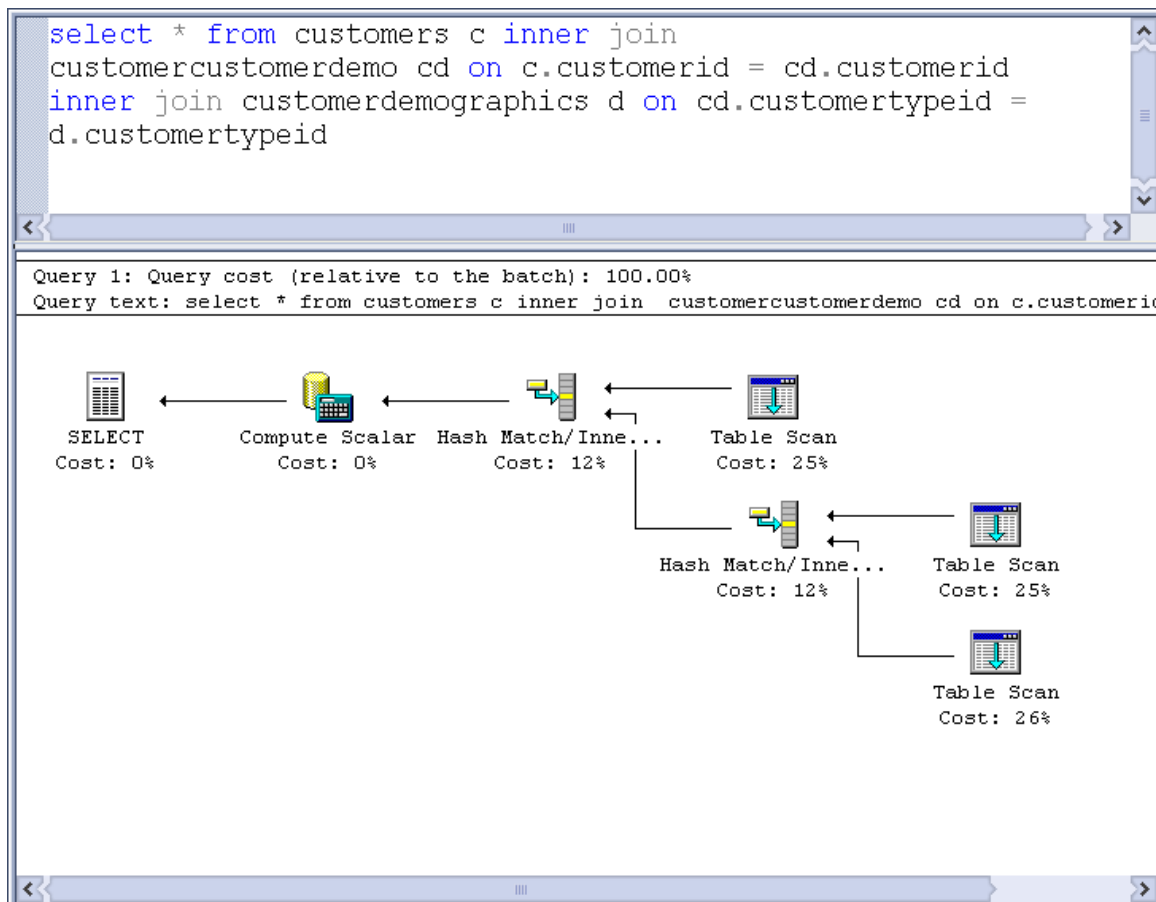
Figure 4.4: Table scan operations indicate a less-than-optimal query.

I could spend some time hunting through the plan to find out which indexes would offer a performance improvement. It's far easier, though, to let the Index Tuning Wizard make some suggestions first. Pressing Ctrl + I activates the Index Tuning Wizard and lets me select the server and database that I want to tune, as Figure 4.5 shows.
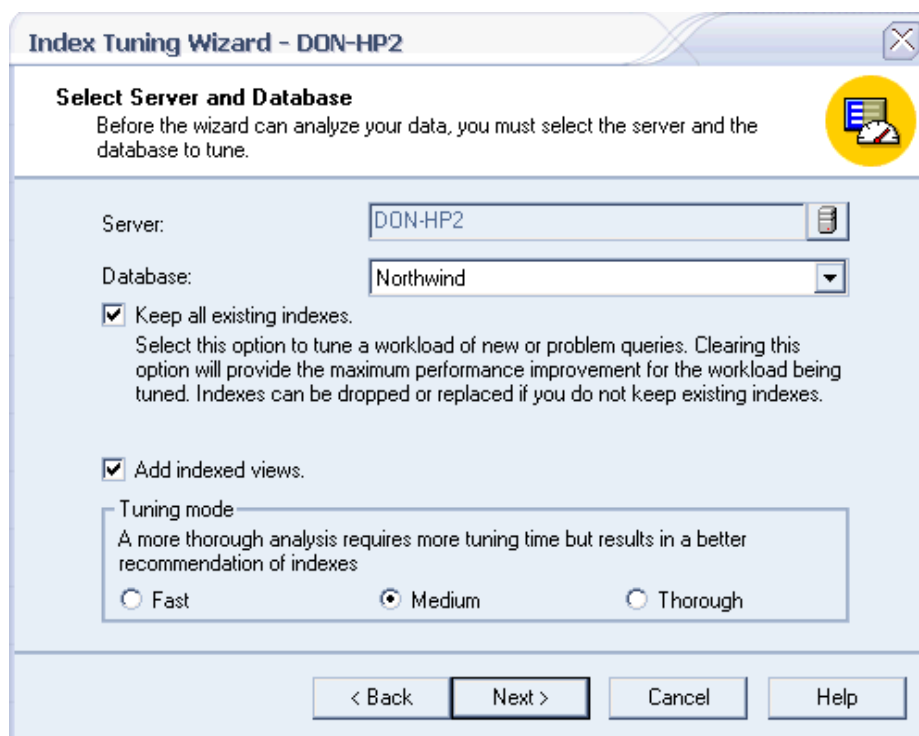
Figure 4.5: Starting the Index Tuning Wizard.

As Figure 4.5 shows, you can configure the Index Tuning Wizard to keep any existing indexes. Feel free to clear the *Keep all existing indexes* check box; some queries run faster *without* indexes in place (INSERT, UPDATE, and DELETE), and the Index Tuning Wizard can recommend indexes to drop.

> 💣 The Index Tuning Wizard can be used to analyze a single query from Query Analyzer or a whole batch of queries profiled by SQL Profiler. Be very careful when dropping indexes based on the Index Tuning Wizard's analysis of a single query. Although the dropped indexes might not be helpful for that particular query, the indexes might be helping the performance of other queries that run against the table. In general, you shouldn't drop indexes unless you're giving the Index Tuning Wizard a truly representative sample of your database's total workload to analyze.

The window that Figure 4.6 shows lets you select the workload that you want the Index Tuning Wizard to analyze. The default workload when you launch the wizard from Query Analyzer will be a single query.
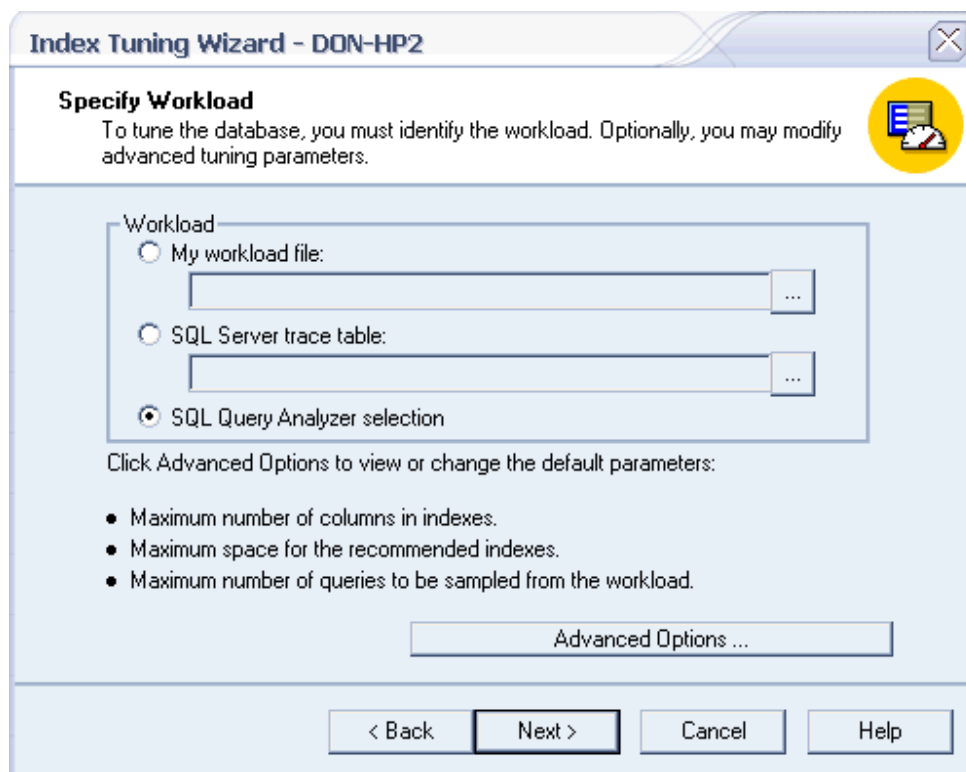
Figure 4.6: Selecting the workload for the Index Tuning Wizard to analyze.

You can also have the Index Tuning Wizard analyze the results of a SQL Server trace, which you can capture with SQL Profiler. Traces can include literally thousands of queries, allowing the Index Tuning Wizard to, for example, analyze an entire day's workload and make indexing recommendations.

> 📖 I discussed SQL Profiler in Chapter 2.

Figure 4.7 shows the Index Tuning Wizard's selection screen. I usually have the Index Tuning Wizard analyze all tables in the database for optimization; if you're only interested in optimizing a table or two, you can select them here. In the Projected column, you can enter your own values. This column lets you tell the Index Tuning Wizard how many rows your table will likely have in the future, and the Index Tuning Wizard will make its recommendations based upon your estimates. This handy feature lets you get indexing recommendations even when your tables don't yet contain enough data to *need* those indexes. After all, it's far better to have the indexes in place now than to not have them in place when your database really needs them.
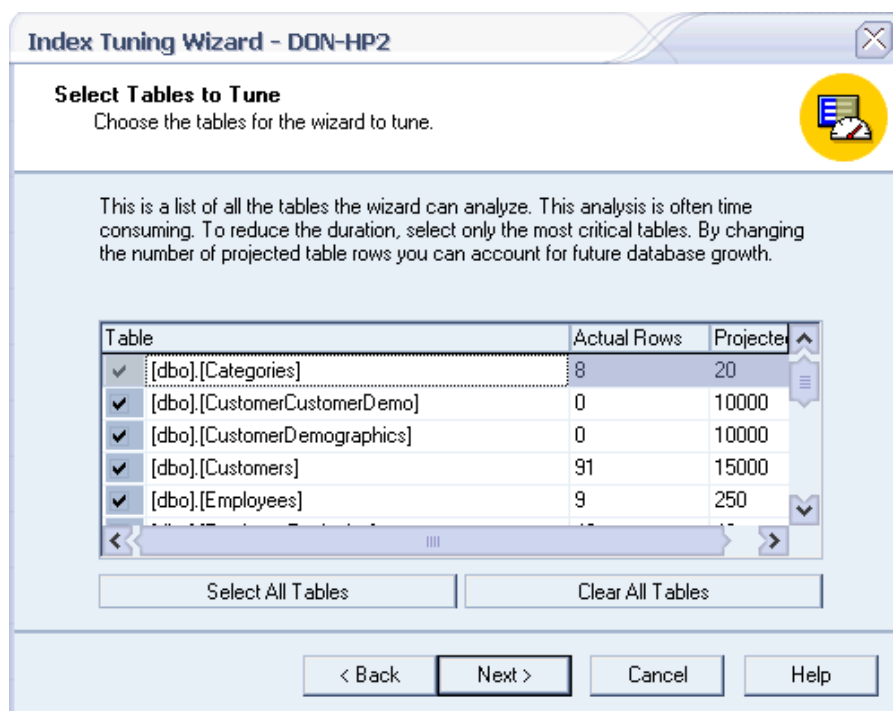
Figure 4.7: Selecting tables and entering projected table sizes.

When you click Next, the Index Tuning Wizard will analyze the workload you've given it. For a single query, the process usually only requires a second or two; if you're analyzing a day's worth of queries from a trace table, the analysis can easily take several minutes to an hour. Be patient; when the Index Tuning Wizard has finished, it will display its recommendations. You can click a single button to have the Index Tuning Wizard create the recommended indexes, and drop any indexes that are in place but not needed by the workload you provided. As you can see in Figure 4.8, the execution plan for my sample query changed pretty drastically with the Index Tuning Wizard's recommendations, and now uses clustered indexes, rather than lengthier table scans, to quickly seek the necessary data.
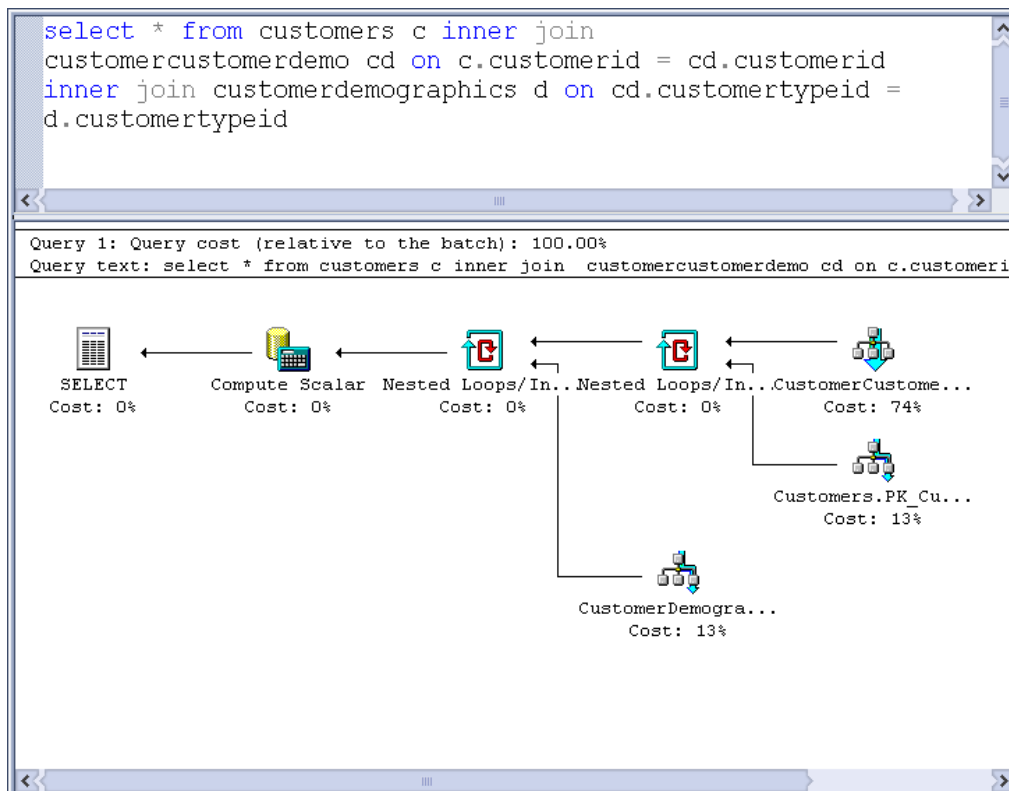
Figure 4.8: Revised execution plan after implementing the Index Tuning Wizard's index recommendations.

This process of checking the revised execution plan after making changes is actually part of my recommended query tuning methodology, which I'll discuss next.

Query Performance Tuning Methodology

Just about the worst thing you can do to your queries is to make changes willy-nilly without some kind of methodology to keep things running smoothly.

I recommend the following best practices:

- After you create a table, immediately run sample queries through Query Analyzer, and document the execution plans displayed. These are your baseline plans, and they might not be very pretty.

- Use the Index Tuning Wizard to generate index recommendations based upon your projected table sizes, and accept the recommendations. Rerun the queries in Query Analyzer, and document the execution plans. These will become the second part of your baseline. Without any data in the tables, the plans might not differ from those in the first step.

VERITAS™

- Every so often, rerun the queries in Query Analyzer and document the execution plans. With real data in the table, you'll see the plans change, and you should document them. Ensure that the plans are providing the best possible performance. How often you rerun the queries to generate new plans depends on how quickly the database is growing; plan to rerun after every 1000 rows or so, to make sure that database growth is supported by the indexes you have in place.

- When you spot a performance problem, analyze the execution plan to look for weak spots, and make changes in indexes, database design, or queries. Immediately rerun your queries to see the changes in the execution plan, and document the changes. Continue to rerun the queries as the database grows to ensure that your changes have a positive effect in the query execution plan.

Notice that I don't worry much about the query's overall execution time. That's a useful indicator of a query that's not performing as well as it could, but a better indicator is the query execution plan itself. If the plan doesn't include costly operations such as table scans, lots of bookmark lookups, and so forth, the query time will be right on target. Execution plans with costly operations, however, represent bad queries, even if the total query execution plan cost doesn't seem too high. For example, one small but poorly written query might execute in two-tenths of a second, which doesn't seem like a long time. However, optimized to have a better execution plan, the query might only require one-tenth of a second. Spread out over a day of production workload, those extra tenths of a second can add up to a major performance difference!

Summary

In this chapter, I introduced you to SQL Server's query optimizer. We explored how you can begin to tune and optimize execution plans by understanding how the optimizer works and why it chooses an execution plan.

Simple execution plans can be simple to optimize—an index here and there, rearranged query syntax, and so forth. Complex query execution plans, however, can require hours of study to determine exactly what SQL Server is doing and why. I'll dive into more complex plans in the next chapter, and show you how to read them and how to take corrective actions to make them more efficient.

VERITAS™

Chapter 5: Understanding Complex Query Execution Plans

Complex queries—those involving many JOIN clauses or other clauses—will generally result in a complex query plan. Troubleshooting simple queries is usually straightforward: Add an index or two, and you're usually running at maximum performance. Complex queries, however, require a lot more time and effort to understand what SQL Server is doing with the query. Only when you know how SQL Server is handling the query can you decide whether SQL Server is taking the best possible course of action and take steps to change the way SQL Server carries out the query.

Query Plan Icons

To understand everything that SQL Server can do with a query, you need to be familiar with the various physical and logical operations that SQL Server is capable of performing. I introduced you to a handful of these, such as Table Scan, in the previous chapter. To begin this chapter, I've provided the entire list, along with a brief explanation of what each operation does and the icon that Query Analyzer uses to represent the operation in a graphical execution plan. I've presented the long list alphabetically so that you can use the list as a reference when you're reading execution plans.

> 🖉 Icons, icons, icons! I'll include the icons for physical operations. These are operations that have some physical effect on the database, such as scanning through rows or deleting a table. The other operations listed are logical operations, which produce a result that's only used within the execution plan and has no actual effect on the database.

- Assert—This operator validates referential integrity, check constraints, and other conditions. The operator returns a NULL value if no conditions have been violated; otherwise, it raises an error and aborts the query. You'll usually see this operator in conjunction with an INSERT or UPDATE query.

- Aggregate—This operator calculates aggregate expressions, such as MIN, MAX, SUM, and so forth. Before calculating, this operator groups its input by a set of columns, making the aggregate calculation easier to perform.

- Bookmark lookup—This operator uses a row ID number, or *bookmark*, to look up the corresponding row in a table. You'll usually see this operator used in conjunction with nonclustered indexes: After SQL Server locates the correct index row, it uses a bookmark lookup to locate the matching data row in the table. Bookmark lookups aren't necessary when a covering index is available, as I discussed in the previous chapter.

- Clustered index delete—This operator deletes rows from a clustered index.

- Clustered index insert—This operator adds rows to a clustered index.

- Clustered index update—This operator updates the rows in a clustered index.

- Clustered index scan—This operator indicates a scan of a clustered index. Scans are fairly inefficient, indicating that the query engine is simply examining each row of the index in order, looking for the appropriate rows.

- Clustered index seek—This operator indicates a seek of a clustered index. Seeks are more efficient than scans, letting the query engine move directly to the desired rows. Seeks are indicative of a useful index.

- Collapse—This operator optimizes the processing of UPDATE queries. UPDATE queries can usually be split (using the split operator) into a DELETE query and a corresponding INSERT query; if the query includes a GROUP BY clause, the query engine groups by the specified columns to eliminate any temporary intermediate steps that aren't necessary to complete the query.

- Compute scalar—This operator evaluates mathematical expressions to produce computed scalar values.

- Concatenation—This operator scans through multiple input tables, returning each row scanned. In effect, the operator merges (or *concatenates*) the two tables into a single result. This action is a fairly inefficient operator, and one to avoid if possible.

- Constant scan—This operator adds a constant row to a query. The operator returns either zero rows or one row, which doesn't usually contain columns. Generally, you'll see a compute scalar function used to add columns to that empty row.

- Cross join—This operator joins each row from the first table with each row from the second table.

- Delete—This operator removes rows from a table.

- Deleted scan—This operator runs within a trigger, and scans the special Deleted table, which contains the rows that were deleted to fire the trigger. For more information, review the SQL Server documentation about triggers.

- Distinct—This operator scans through its input, removing duplicates. The result is output that contains unique values.

- Distinct sort—This operator functions much like the distinct operator, but, in addition, sorts the specified columns in the output.

- Distribute streams—This operator is used only in parallel query plans, and converts a single input stream into multiple output streams. In effect, this operator is the start of parallel query processing.

- Eager spool—This operator stores each input row in a hidden temporary table. When multiple nested operations are performed against a row set, this operator can allow successive operations to execute more quickly by using the temporary table rather than rescanning the original input.

- Filter—This operator scans its input, returning only the rows that satisfy the filter expression.

- Flow distinct—This operator works much like the distinct operator. However, the distinct operator scans all input before producing any output; whereas, this operator evaluates and returns each row as it is received from the input.

- Full outer join—This operator returns each row satisfying the join condition from the first table joined with each row from the second table. This operator produces a single output row for each row in the first table and each row in the second table, regardless of whether the rows match. Any output row not matching both tables will contain NULL for the non-matching values.

- Gather streams—This operator is used in parallel execution plans, and is the opposite of the distribute streams operator. This operator gathers multiple input streams into a single output stream.

- Hash match—This operator builds a hash table by computing a hash value for each input row. Then the operator computes a hash (using the same hash function) for each probe row, and uses the hash table to quickly locate probe rows in the input row set. This operator is often used in JOIN queries by using the first table in the join to build the hash table, then using the rows from the second table as probe rows. This operator can also be used with distinct or aggregate operators to quickly remove duplicates. Because hashes operate on an entire table one row at a time, they're not always the most efficient operation; indexes can be faster. For example, with a unique index in place, a distinct operation can utilize the index to generate unique output rather than building a hash table.

- Hash match root—This operator coordinates multiple hash match team operators, forcing all subsidiary matches to use identical hash functions.

- Hash match team—This operator is part of a team of connected hash operations that share a common hash function.

- Index delete—This operator deletes rows from a nonclustered index.

VERITAS™

- Index insert—This operator adds rows to a nonclustered index.

- Index scan—This operator scans through the rows of a nonclustered index. This action is less efficient than an index seek because the index scan operator must process all rows in the index.

- Index seek—This operator seeks to a specific row in a nonclustered index, and is more efficient than a scan operation.

- Index spool—This operator scans its input rows and places a copy of each row in a hidden temporary table. The operator then indexes the temporary table and uses it to perform subsequent seek operations. As with most spool operators, index spool is something you want to avoid if possible because temporary tables are inherently inefficient, not to mention the inefficiency of the spool operation itself.

- Index update—This operator updates the rows in a nonclustered index.

- Inner join—This operator accepts two input row sets and returns all rows from both sets that match the join condition.

- Insert—This operator inserts each row from its input into an object. The associated physical operator will either be table insert, index insert, or clustered index insert.

- Inserted scan—This operator scans the special inserted table within a trigger. See the SQL Server documentation about triggers for more information about how this special table works.

- Lazy spool—This operator functions similar to the eager spool operator. However, the eager spool operator builds its spool table all at once; whereas, the lazy spool operator builds its spool table as rows are requested by the operator's parent operator.

- Left anti semi join—This operator returns each row from the first table where there is no matching row in the second table. If there is no specified join condition, then each row is a matching row.

- Left outer join—This operator returns each row from the first table that matches a row in the second table. It also returns any rows from the first table that have no matches in the second table, filling in the nonmatching rows with NULL values.

- Left semi join—This operator returns each row from the first table that matches a row in the second table.

- Log row scan—This operator scans the transaction log.

- Merge join—This physical operator performs the inner join, left outer join, left semi join, left anti semi join, right outer join, right semi join, right anti semi join, and union logical operators.

- Nested loop—This operator performs the inner join, left outer join, left semi join, and left anti semi join logical operators. This operator performs a search on the inner table of the join for each row of the outer table in an iterative process, usually utilizing an index.

- Parallelism—This operator performs the distribute streams, gather streams, and repartition streams logical operations for parallel query execution.

- Parameter Table scan—This operator scans a table that is acting as a parameter in the current query. Usually, you'll see this used for INSERT queries within a stored procedure.

- Remote delete—This operator deletes the input rows from a remote object.

- Remote insert—This operator adds rows to a remote object.

- Remote query—This operator submits a query to a remote source and returns the results.

- Remote scan—This operator scans a remote object.

- Remote update—This operator submits updates to a remote object.

- Repartition streams—This operator is found only in parallel queries. This operator takes multiple input streams and returns multiple output streams, rebalancing the workload across the multiple execution threads running within the parallel query. You can think of this operator as a kind of combined gather streams and distribute streams operator, or perhaps a "redistribute streams" operator.

- Right anti semi join—This operator outputs each row from the second table that does not match a row in the first table—the flip of the left anti semi join operator.

- Right outer join—This operator is the flip of the left outer join operator, returning all rows from the second table in the join. For rows matching a row in the first table, the values from the first table are returned; otherwise, NULL values are returned.

- Right semi join—This operator is the flip of the left semi join operator, returning rows from the second table that match rows in the first table.

- Row count spool—This operator scans the input, counts the rows, and returns the same number of empty rows. This operator is useful when a query needs to count rows matching a particular condition without actually using any of the data in the rows. You'll often see this operation used along with nested loops, as the row count spool operation lets SQL Server count the number of rows that will need to be processed within the loops.

- Sequence—This operator executes each input in sequence. Inputs are usually updates of an object.

- Sort—This operator sorts all incoming rows and outputs all rows in sorted order.

- Split—This operator optimizes update processing by splitting each update operation into a delete and a corresponding insert.

- Stream aggregate—This operation requires input that is ordered by the columns within its groups, and calculates aggregate expressions.

- Table delete—This operator deletes rows within a table.

- Table insert—This operator inserts rows into a table.

- Table scan—This operator scans a table's rows, and is generally a fairly inefficient operation, especially on larger tables. The result of this operation is all rows matching the query conditions.

- Table spool—This operator copies a table into a hidden temporary table, allowing subsequent operators to use the table's data without requerying the table from the database.

- Table update—This operation updates data in a table.

- Top—This operation scans the input, returning the first specified number (or percent) of rows. See the SQL Server documentation about the TOP operator for more details.

- Union—This operator scans multiple input row sets, and outputs each row after removing duplicates.

VERITAS™

- Update—This operator updates each row from its input in the specified object. The corresponding physical operator is table update, index update, or clustered index update.

You'll notice that many of these operators refer to an input or output. All operators require an input and provide an output; generally, the input is a table (or a set of tables), and the output is another table or set of rows. Remember that SQL Server processes execution plans sequentially, so ideally, each successive operator should produce a smaller output than its input; gradually reducing the size of the row set and eventually producing the desired query result. After you understand what each of these operators does, you can read query execution plans more intelligently, looking for inefficient operators and modifying your query or your database to eliminate them when possible.

Reading Text-Based Query Plans

Especially complex graphical execution plans can be difficult to read, and you might find a text version easier to deal with. To force SQL Server to display a text-based execution plan, execute the SET SHOWPLAN_TEXT ON command. Figure 5.1 shows a fairly simple query and its graphical execution plan.
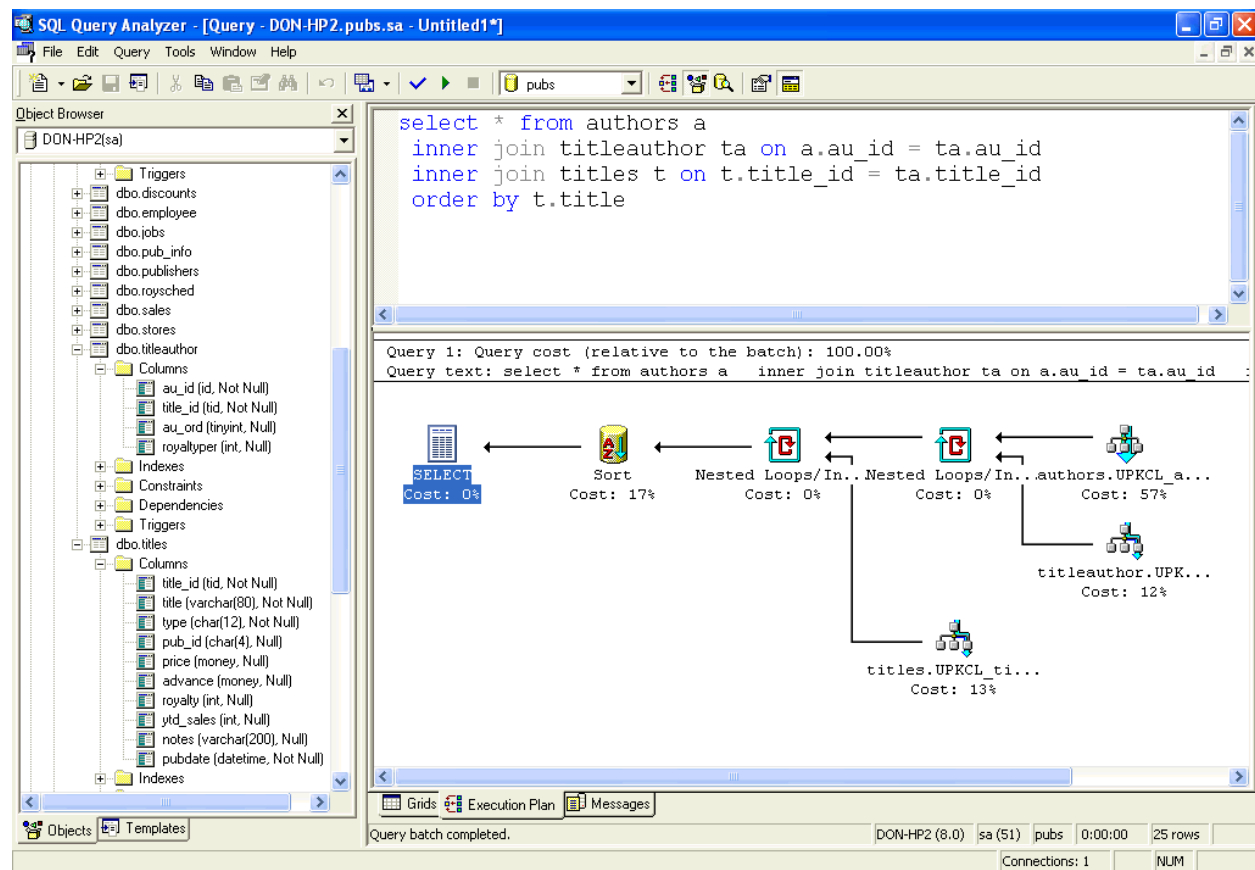
Figure 5.1: A basic query and its graphical execution plan.

Figure 5.2 shows the corresponding text execution plan, which can be a bit easier to read.
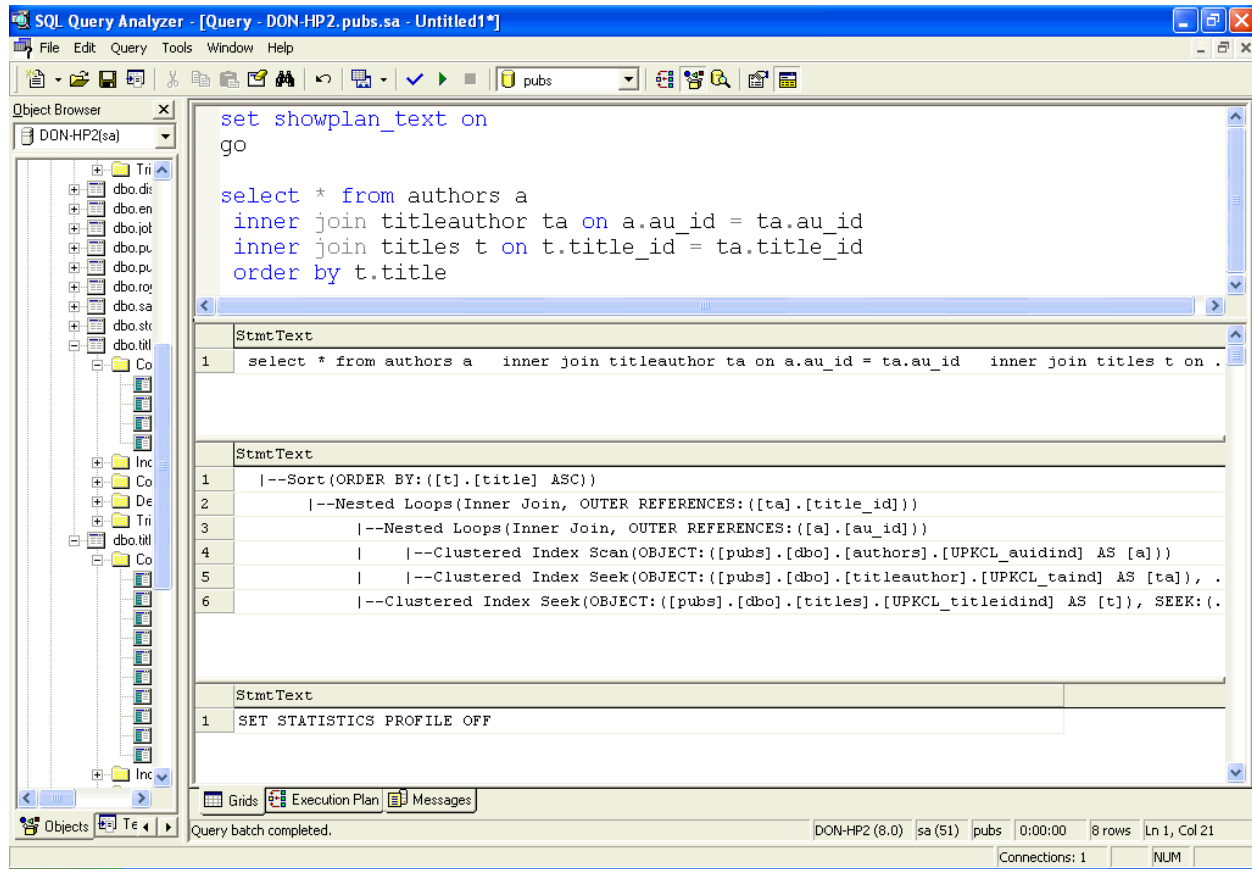
Figure 5.2: The same query and its text-based execution plan.

Even text-based execution plans can become unwieldy for larger queries, especially those with multiple joins. You might need a wide-format printer, such as a plotter, to create hardcopies of these execution plans, rather than attempting to review them on the screen.

Let's briefly review the text-based query plan that Figure 5.2 shows. Listing 5.1 shows this query plan in a slightly different formatting for easier reading:

```
|--Sort(ORDER BY:([t].[title] ASC))
 |--Nested Loops(Inner Join, OUTER REFERENCES:([ta].[title_id]))
      |--Nested Loops(Inner Join, OUTER REFERENCES:([a].[au_id]))
      |     |--Clustered Index
Scan(OBJECT:([pubs].[dbo].[authors].[UPKCL_auidind] AS [a]))
      |     |--Clustered Index
Seek(OBJECT:([pubs].[dbo].[titleauthor].[UPKCL_taind] AS [ta]),
SEEK:([ta].[au_id]=[a].[au_id]) ORDERED FORWARD)
      |--Clustered Index
Seek(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind] AS [t]),
SEEK:([t].[title_id]=[ta].[title_id]) ORDERED FORWARD)
```

Listing 5.1: Example text-based query plan.

> ☞ Try it yourself! You can execute this query yourself against SQL Server's default Pubs table. Just type the following into Query Analyzer, make Pubs the current database, and click the Execute button:
>
> ```
> Set showplan_text on
> go
> select * from authors a
>  inner join titleauthor ta on a.au_id = ta.au_id
>  inner join titles t on t.title_id = ta.title_id
>  order by t.title
> ```

The outer operation is a sort, which is executing the ORDER BY t.title clause in the original query. That operator is receiving its input from a nested loop, which is executing the first INNER JOIN clause. The outer reference for the clause is the title_id column of the TitleAuthor table. The next operation is also a nested loop, performing the second INNER JOIN with an outer reference of the au_id column in the Authors table.

That join is executed by using a clustered index scan against the Authors table's primary key, and a clustered index seek against the TitleAuthor table's primary key. The scan operation returns all rows for the Authors table, and the seek operation returns the matching rows from the TitleAuthor table. In this case, the scan is no problem because I wanted all rows returned.

The second join takes the results of the first and uses a clustered index seek against the Titles table's primary key, returning the rows that match the rows already in the first join's results. Finally, the results of the second join are passed to the sort operation where the ORDER BY clause is implemented.

This query is fairly efficient. Most of the query's cost is taken up in the clustered index scan, which, in this case, is unavoidable given the desired results. But suppose I dropped the clustered indexes that the query is currently using and updated the tables' statistics? You'd see an execution plan like the one that Figure 5.3 shows.

Figure 5.3: Revised query execution plan.

Notice that the clustered index scans and seeks are gone, replaced by more costly table scans. The one index still in use is on the Titles table, and that is a nonclustered index, requiring a corresponding bookmark lookup to locate the actual data row. Also notice that the Query Optimizer has eliminated the sort operator because the data is being selected in the correct order thanks to the ordered index scan. However, this query is much more costly than the original query, taking nearly twice as long to run. If you were to look at the corresponding graphical execution plan, you'd see that the table scans are each generating about 30 percent of the query's execution time, the bookmark lookup about 5 percent, and the index scan another 30 percent.

Improving Complex Queries

Most complex queries involve joins, which require the Query Optimizer to make a lot of decisions. The optimizer basically has three choices: A nested loop join, a merge join, or a hash join. The optimizer chooses between these based on the size of the data involved and the availability of indexes—making it especially important for the optimizer to have updated statistics upon which to base its decisions.

Nested Loop Joins

These joins use one join table as the outer input table, which is shown at the top in the graphical execution plan. The second input is the inner input table, shown at the bottom of the graphical plan. The inner loop is executed once for each row in the outer input, searching for matching rows in the inner input. Nested loop joins are most effective if the outer input is fairly small because that requires fewer iterations of the inner loop. Also, a large, indexed inner loop is efficient because the index can be exploited for seeks, rather than scans. Nested loop joins are also more effective with smaller sets of data. Larger queries can often benefit from merge or hash joins because either method can be faster than multiple iterations through large sets of data.

Merge Joins

These joins require both inputs to be sorted on the merge columns. Typically, the Query Optimizer scans an index, if one exists, or places a sort operator below the merge join. Indexes are, of course, far more efficient than a sort. Because each input is sorted, the merge join operator can retrieve a row from each input and compare them. Whether the rows are returned depends on the join; in an inner join, for example, the rows are returned if they match.

In a many-to-many operation, the merge join operator constructs a temporary table to hold rows. If there are duplicate values from either input, one of the inputs will have to be rewound to the start of the duplicates as each duplicate from the other input is processed. This process ensures that each row in one input is compared with potential matches in the other input.

Merge joins are very fast; although if they have to use a sort operator, they can slow considerably. Indexes on the merge columns eliminates the need for a sort operation, making merge joins one of the most effective decisions for large volumes of data. Note that merge joins don't require the iterative loops of a nested joins loop, which accounts for the speedy execution of merge joins.

Hash Joins

Hash joins have two inputs. The first input is the *build input*, which is used to build a hash table. The *probe input* is compared with the hash table to find duplicates. Here's how it works: SQL Server uses the build input to construct a hash table. *Hashes* are mathematical algorithms that produce a deterministic output. In other words, given the same input, the hash function will always produce the same output. The output of the hash is similar to a checksum, and is a value that uniquely represents the input.

SQL Server then loops through the probe input, hashing each row. Because the hash table is sorted, SQL Server can very quickly determine whether the hashed probe rows are matched in the hash table, generating the join results.

Suppose, for example, that you have one table with three rows. The primary keys for those rows have the values Arthur, Bob, and Clancy. Hashed, these values might be A41, 5B2, and CA8. The table you're joining with also has three rows, and the join column contains the values Arthur, Robert, and Clancy. Hashed, these probe inputs will be A41, R8E, and CA8. SQL Server will hash each probe input row and locate any matches in the hash table. In this case, A41 and CA8 match, producing the join results Arthur and Clancy.

SQL Server has a few hash mechanisms to work with:

- In-memory hash joins build the hash table in memory. This method is used only if the entire hash table can fit within the available memory.

- Grace hash joins break the build table into a series of partitions, which are stored in multiple files. Because the same hash function is used throughout, SQL Server can guarantee that any two matching hashes will be located in the same file.

- Recursive hash joins are used for especially large inputs, and multiple levels of partitioning are used. Large, asynchronous disk operations are used to generate the multiple files as quickly as possible.

SQL Server can't always tell upfront which hash type will be used. Therefore, it usually starts with an in-memory hash and uses the other types as necessary. Sometimes, the optimizer might incorrectly guess which of the two inputs is the smaller (and therefore should have been the build input rather than the probe input). If this occurs, SQL Server will dynamically flip the inputs, ensuring that the smaller of the two inputs is used as the build input, to reduce the size of the hash files as much as possible.

Optimizing

So which join type should you shoot for? For smaller inputs, a nested loop or hash join isn't a big deal. For larger inputs, merge joins are usually the most desirable, except for especially large inputs for which multiple merges would be required. You can usually get SQL Server to use a desired join type simply by making the appropriate indexes available. However, you can go too far—don't create indexes that SQL Server subsequently refuses to utilize, as you'll just be slowing table updates for no good reason. Examine your query execution plans before and after changes to determine what SQL Server is actually utilizing.

Using Query Hints

You can often force SQL Server to use a particular query strategy by using query hints. These hints override SQL Server's default behavior.

💣 Be careful when using hints! I don't recommend using hints without extreme caution. SQL Server's Query Optimizer is designed to change its decisions based on your data's growth and other changes. When you use hints, you're overriding that dynamic behavior, forcing SQL Server to use a particular method whether it's appropriate or not. If you decide to use hints, make sure you review your query execution plans frequently so that you can modify your hints as appropriate to reflect changes in your data.

When are hints appropriate? Sometimes, SQL Server will simply refuse to select a query strategy that you know is better. I've seen SQL Server ignore indexes on huge tables and execute hashes and table scans in defiance of all logic. In these cases, the database administrators (DBAs) responsible for the server were forced to implement query hints so that SQL Server would use more efficient merge joins. The administrators verified that the hinted queries executed many times faster than the unhinted queries, and they monitored the queries' performance over time to make sure performance remained acceptable.

You can force SQL Server to use loop, hash, or merge joins by specifying the LOOP, HASH, or MERGE keyword in a join condition. Table hints can be used to force SQL Server to use a specified index that you've created, as the following examples illustrate:

- SELECT * FROM *tablename* WITH (INDEX(*indexname*))—This forces SQL Server to use the designated index to execute the query.

- SELECT * FROM *tablename* INNER LOOP JOIN *tablename*—This forces SQL Server to execute a nested loop join; replace LOOP with MERGE or HASH to force the corresponding join strategy.

SQL Server also supports some miscellaneous hints, which you can specify by using the OPTION *optionname* clause within your query. The available options include:

- EXPAND VIEWS—Forces the optimizer not to use any indexed views in the execution plan.

- FAST *n*—Forces the optimizer to return the specified (*n*) number of rows as fast as possible, even if doing so hurts the query's overall performance. You can use this option to create the illusion of speed for the user, quickly returning the first set of rows while SQL Server works to churn out the rest.

- FORCE ORDER—Forces the optimizer to join tables in the query in the order in which they are specified rather than attempting to reorder them for optimal execution.

- KEEPFIXED PLAN—Forces the optimizer to use a stored or cached query execution plan regardless of any changes to statistics that might render the plan outdated.

- KEEP PLAN—Forces the optimizer to reduce the recompile threshold for a query. Doing so makes the optimizer less willing to recompile a plan, but still leaves recompilation as an option if the plan is seriously out of date.

- MAXDOP *n*—Tells the optimize to override the maximum degrees of parallelism server configuration setting for this query, specifying the maximum number (*n*) of CPUs that can be used for parallelism. Doing so can be useful for temporarily turning off parallelism for queries that you know won't run as well.

- ROBUST PLAN—Forces the optimizer to create an execution plan that works for the maximum potential row set size, even if that means hurting performance. You'll usually only use this hint if you receive an error about exceeding the row size limit, which can occur for very large tables with very large varchar columns.

If you decide to use query hints, be sure to document your before and after execution plans and to carefully monitor the performance of your hinted queries in the future. The more you override the Query Optimizer, the less your queries will be able to respond to changing server and data conditions automatically, and the more you'll need to manually babysit them to keep performance high.

Using Indexed Views

Views are often implemented as a shortcut, to create a virtual representation of one or more tables that can be queried without repeated complex queries. Views are built from an underlying query that specifies the tables and columns that will be contained within the view. Normally, views have no impact on table updates; the view is dynamically reconstructed from the base query each time the view is referenced.

SQL Server 2000 supports the creation of indexes on views, which can have an impact on performance:

- Whenever a table is updated, its indexes must be updated to reflect the changes. In addition, the indexes on any indexed views that reference the table must also be updated. In write-heavy tables, indexed views can severely compound update overhead and reduce performance.

- In a query, the Query Optimizer will use any indexes on views that might help accomplish the query at hand, even if the query at hand doesn't reference the view. In other words, any index built upon a view can be used to help execute queries on the underlying table as well.

🔴 Using Enterprise Edition? Only the Enterprise Edition of SQL Server will automatically choose to use indexed views to execute queries. Other editions of SQL Server support indexed views but will not consider them in the query optimization process. You can force them to be considered by using the NOEXPAND query hint.

Normally, SQL Server doesn't store the results of a view. As I've described, it simply reconstructs the view each time the view is referenced. Thus, SQL Server must re-optimize the view's underlying query, which can involve a lot of overhead if the query is fairly complex. You can improve the performance of the view by creating a unique clustered index on the view. Doing so forces SQL Server to execute the view and store the result set in the database, in the same way that a regular table is stored. With the unique clustered index in place, you can create additional nonclustered indexes on the view if desired.

☞ Use the Index Tuning Wizard! The Index Tuning Wizard will provide suggestions for creating indexed views, if you let it. Simply select the appropriate option when you run the wizard to have it consider the impact of indexed views on your query workload.

To use indexes on a view, you have to meet an array of requirements:

- The ANSI_NULLS and QUOTED_IDENTIFIER options must have been set to ON when the original CREATE VIEW statement was executed.

- The ANSI_NULLS option must have been ON when you executed the CREATE TABLE statement to create all tables referenced by the view.

- The view must reference only base tables and not other views.

- All tables referenced by the view must be in the same database and have the same owner as the view.

- The view must be created with the SCHEMABINDING option, binding the view to the underlying tables' schema. See the CREATE VIEW command in SQL Server's documentation for more details about this option.

- Any user-defined functions referenced by the view must also have been created with the SCHEMABINDING option.

- All tables and user-defined functions must be referenced by two-part names only (owner.objectname). One-, three-, and four-part names are not permitted.

- Any functions referenced by expressions in the view must be deterministic. In other words, each function must return the same output given the same input. The INT() function is an example, as INT(2.4) will always return 2. The DATE() function, however, will return different output from day to day, and is therefore not deterministic.

- Finally, the SELECT statement used to build the view cannot contain:

  - The * operator to specify all columns. You must specify a column list.

  - You cannot specify a column name twice in the column list. For example, Column1, Column2 is fine; Column1, Column2, SUM(Column2) is not allowed.

  - A derived table.

  - Rowset functions.

  - The UNION operator.

  - Subqueries.

  - Outer or self joins.

  - The TOP clause.

  - The ORDER BY clause.

  - The DISTINCT keyword.

  - The COUNT(*) function.

  - The following aggregate functions: AVG, MAX, MIN, STDEV. STDEVP, VAR, or VARP.

  - Any SUM function that references an expression whose results could be NULL.

  - The CONTAINS or FREETEXT predicates.

  - The COMPUTE or COMPUTE BY clauses.

It's a hefty list of restrictions, but all conditions must be met for the indexed view to be possible. Finally, when you issue the CREATE INDEX command to create the index on the view, the following options must be set to ON: ANSI_NULLS, ANSI_PADDINGS, ANSI_WARNINGS, ARITHABORT, CONCAT_NULL_YIELDS_NULL, and QUOTED_IDENTIFIERS. The view can only contain text, ntext, or image columns.

VERITAS™

✎ Check these conditions before running the Index Tuning Wizard. If you're relying on the wizard to recommend indexed views, keep in mind that it will only do so if the previously listed constraints are met. You should check these conditions prior to running the wizard or it might decline to create indexed views simply because of these constrains and not because the views wouldn't be helpful to your query workload.

Where will indexed views get you the most bang for your buck? Data warehouses and other decision support queries are the most obvious choice. Also, any existing view that joins two or more large tables, as the indexed view will execute more quickly than a non-indexed view.

Also keep in mind that an indexed view offers the opportunity to create multiple clustered indexes on a table. If you think a table might benefit from somehow being in two physically different sort orders at the same time, indexed views offer that mind-boggling opportunity. However, don't underestimate the performance impact of adding an indexed view to a write-heavy table, and make sure that you keep an eye on your before and after performance numbers.

Using Stored Procedures

As I've mentioned, stored procedures can have an immense performance benefit on almost any application. I've worked in many shops for which the policy was clear: All queries will be executed from stored procedures and never submitted as ad-hoc queries. Why? Simple: SQL Server saves the execution plan for stored procedures, meaning it doesn't have to re-optimize the procedure every time the procedure is executed. The optimizer has a threshold, and will recompile procedures after a certain amount of time so that their execution plan reflects the current condition of the database's statistics, but recompiling every so often is much better than re-optimizing every single time. In addition, stored procedures offer the following less–often–considered benefits:

- Clients transmit less data to execute a stored procedure than to execute an ad-hoc query. Generally, clients only transmit the name of the procedure and a few parameters rather than an entire query statement (or batch of statements).

☞ Stored procedures *really* reduce network traffic. When clients execute a stored procedure, SQL Server responds with the procedure's results, if any, and a count of the number of rows the procedure affected. Applications rarely use this information, and you can tell SQL Server not to execute it. Simply place the SET NOCOUNT ON command at the start of each stored procedure. Or, to turn off the row count server-wide, execute the following:

```
SP_CONFIGURE 'user options', 512
RECONFIGURE
```

Keep in mind that performing this server-wide reconfiguration will affect all transactions on the server.

- Stored procedures encourage modular programming and code reuse, which reduces maintenance and makes applications more efficient.

- Stored procedures can increase data security by reviewing data for correctness before changing the database.

VERITAS™

☞ Watch stored procedure performance! Stored procedures aren't guaranteed to perform better than ad-hoc queries; it depends on how they're written. SQL Profiler includes a built-in trace template, *Profile the Performance of a Stored Procedure,* that can monitor a procedure's performance in production (real or simulated) conditions. Keep in mind that SQL Server doesn't always pick the best execution plan for stored procedures, so examining those execution plans is a great idea.

Stored procedures can be parameterized, allowing them to accept required and optional parameters to customize their behavior. Don't get carried away with parameters—the fewer the better. In addition, avoid optional parameters entirely, if possible, because they can force otherwise unnecessary procedure recompilation. If you want to write a query that can accept either two or three parameters, do so with two stored procedures. One can accept two parameters, and the other can accept three. Each stored procedure might even call many of the same sub-stored procedures to keep your program code nice and modular.

☞ You might have occasion to create server-wide stored procedures that you save in the Master database. If you do, name them with an sp_ prefix. Do not, however, name other stored procedures with the sp_ prefix. SQL Server always looks for sp_ procedures in Master first, and only looks in your current database if it can't find the procedure in Master. You'll speed things by using a different procedure name so that SQL Server won't jump to Master first.

SQL Server will automatically recompile a stored procedure's execution plan in a variety of circumstances. You can use these to control recompilation to meet your needs:

- Whenever you include a WITH RECOMPILE clause in a CREATE PROCEDURE or EXECUTE statement.

- If you run sp_recompile for any table referenced by a procedure.

- If any schema changes occur to any object referenced within the stored procedure. Schema changes include dropping or adding rules, defaults, and constraints.

- If you restore a database that includes a stored procedure or any objects referenced from within a stored procedure.

- If an index used by a stored procedure's execution plan is dropped or rebuilt.

- If a table's statistics change a lot (which will happen whenever a large number of data rows are added, changed, or deleted).

- If the stored procedure contains any data definition statements along with data manipulation statements (such as SELECT). The procedure will actually recompile in mid-run after executing data definition statements, then encountering data manipulation statements; the theory being that the definition statements changed the underlying tables that the manipulation statements were using. In general, avoid changing database objects and querying them in the same procedure.

- If the stored procedure performs almost any action with a temporary table. This behavior is a major reason why temp tables are so despised by DBAs: They almost completely defeat the ability of SQL Server to accurately cache the procedure's execution plan, and they force a recompilation at every execution. The only exception is for temp tables created within the stored procedure and never referenced outside of it, assuming you first declare the temp table before attempting to reference it.

☞ Avoid creating temp tables within a control-of-flow construct, such as an IF…THEN block. If you *might* need to use a temp table within a stored procedure, create the table outright at the beginning of the procedure, and drop it at the end. That way, the table will exist no matter what flow your procedure follows, and the procedure won't have to recompile every time it's executed.

Combining SQL Profiler and the Index Tuning Wizard

When you're trying to fine-tune really complex queries, using SQL Profiler with the Index Tuning Wizard can be very effective. The Index Tuning Wizard can analyze all the workload represented in a SQL Profiler trace database, effectively offering suggestions for a realistic production workload.

To get things started, you'll need to open SQL Profiler and start a new trace. As Figure 5.4 shows, you'll want to configure the trace to save to a database table. *Do not* save the trace to the same SQL Server that you're profiling (even though I did so in this example); the activity generated by the trace can throw off the statistics that the Index Tuning Wizard analyzes.
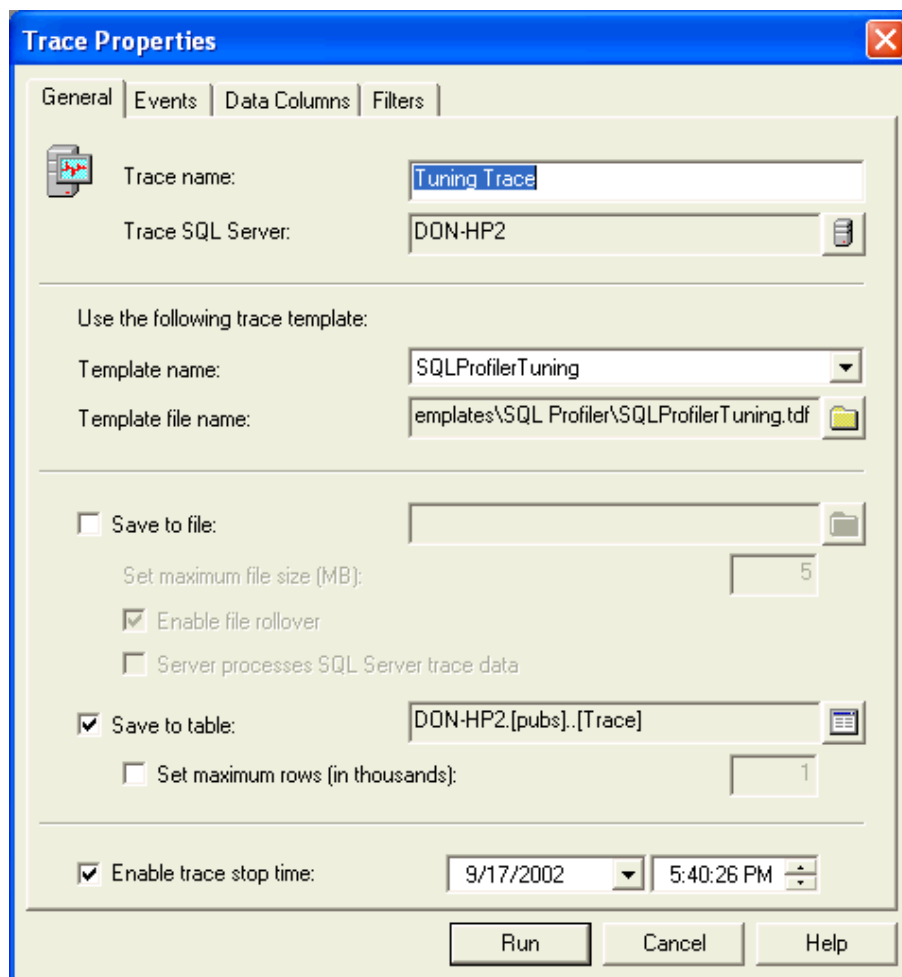
Figure 5.4: Saving the trace to a database table lets the Index Tuning Wizard work with the trace data.

I like to start with the SQL Profiler's built-in SQLProfiler Tuning trace, although, as Figure 5.5 shows, I like to add the entire Performance event category to the event classes that are predefined in the trace template.
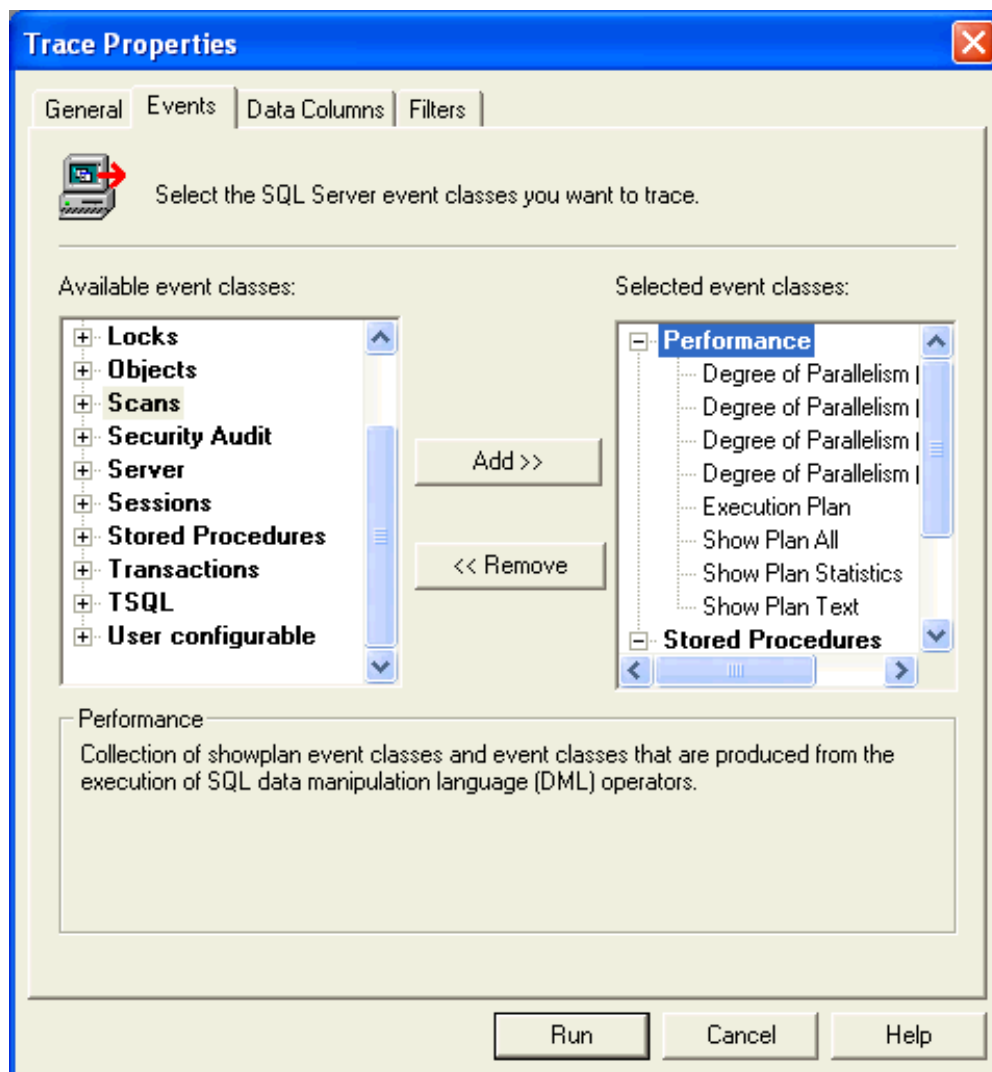
Figure 5.5: The additional events make it easier for you to spot performance problems (such as long-running execution plans) in the trace.

When you've got the trace configured, start it, and let it run while you perform a representative sample of work. For really complex queries, I like to run SQL Server through an entire process. For example, if you have a month-end process that works with a lot of complex queries, run that process while you're tracing the server.

☞ You might not want to kick off periodic processes on your production server. Instead, you might want to back up your production database, restore it to a development server, then run the processes on the development server while you trace it. This technique allows you to use SQL Profiler against production processes when it isn't necessarily convenient to actually run the processes in your production environment.

Figure 5.6 shows a completed trace. After you stop the trace, you can close SQL Profiler or you can launch the Index Tuning Wizard right from SQL Profiler's Tools menu.

Figure 5.6: You might want to scroll through the trace data to familiarize yourself with it before you run the data through the Index Tuning Wizard.

When you launch the Index Tuning Wizard from within SQL Profiler, the wizard configures itself to analyze a SQL Profiler trace. However, as Figure 5.7 shows, you'll still need to direct the Index Tuning Wizard to the server, database, and table that contains the trace data.

VERITAS™
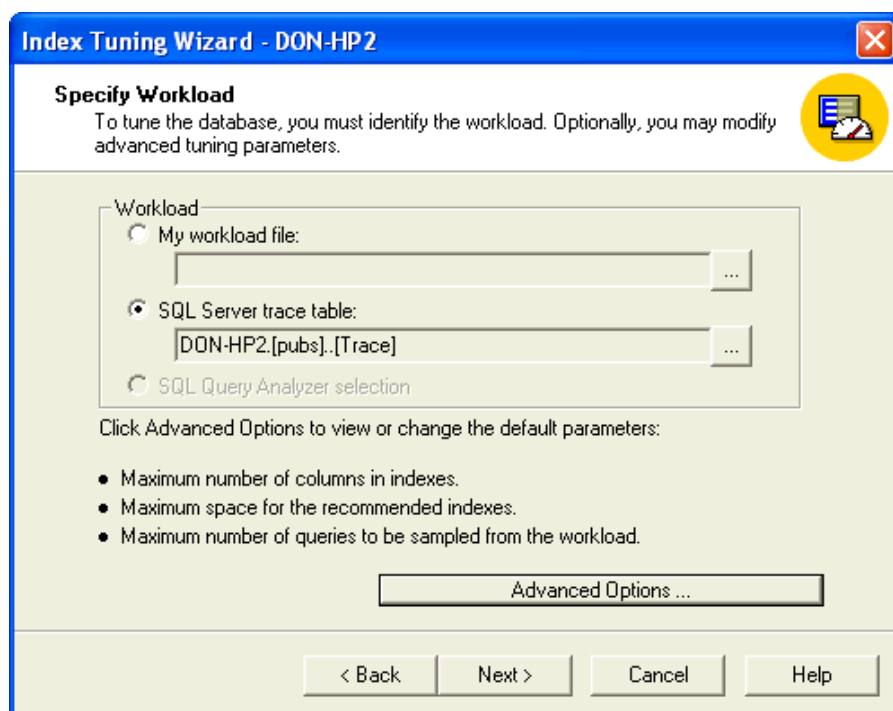
Figure 5.7: Pulling trace data from a database table lets the wizard analyze entire days' worth of query workload, if desired.

I also like to click Advanced Options and modify the default options. Usually, the Index Tuning Wizard won't recommend new indexes that would exceed 2MB in size. I usually bump that number up to 20MB or so, just to make the Index Tuning Wizard show me all of my possible options. That doesn't necessarily mean that I'll implement an index that large, but I'd like to see the recommendation anyway.

Step through the Index Tuning Wizard's remaining screens, which I described in the previous chapter. When the wizard finishes, it will display a recommendations screen similar to the one that Figure 5.8 shows. Notice that two new indexes were recommended and are displayed with a yellow star icon. These are the indexes that would have improved server performance had they been present when the trace was run.
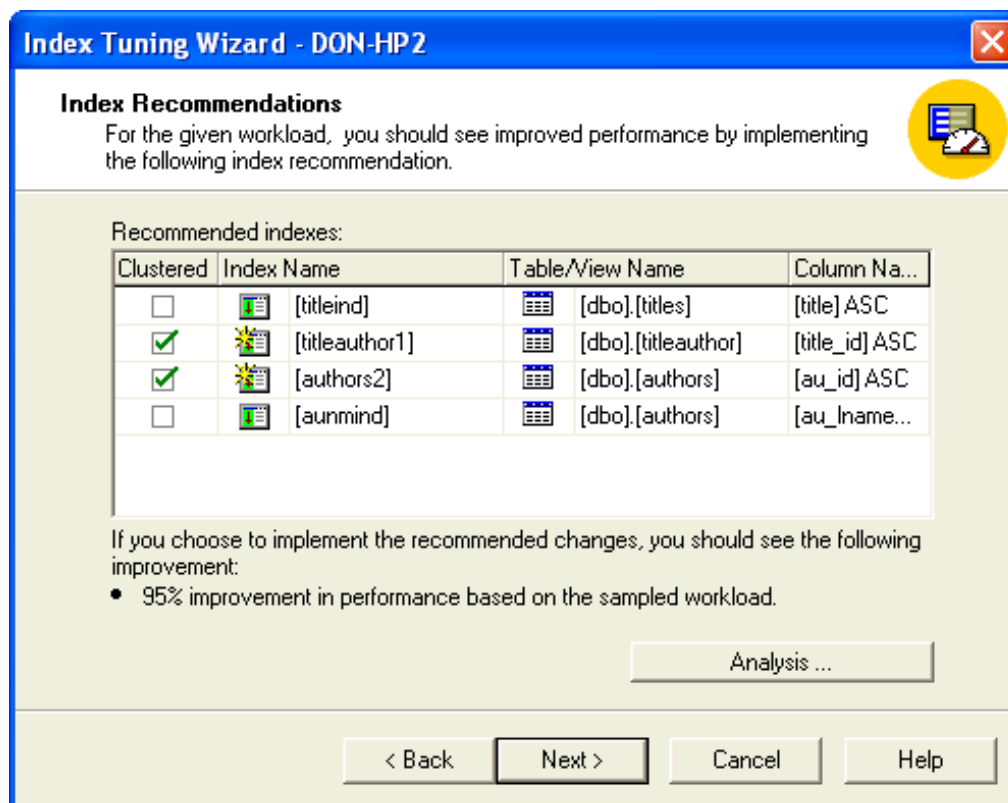
Figure 5.8: The Index Tuning Wizard's index recommendations.

You can click Analysis to see how each index—including the recommended indexes—was used or would have been used had it been present. This screen, which Figure 5.9 shows, can help you judge which indexes will actually offer the most benefit to your server's performance. Back on the main recommendations screen (see Figure 5.8), clear the check box next to any index you don't want to create, then click Next.
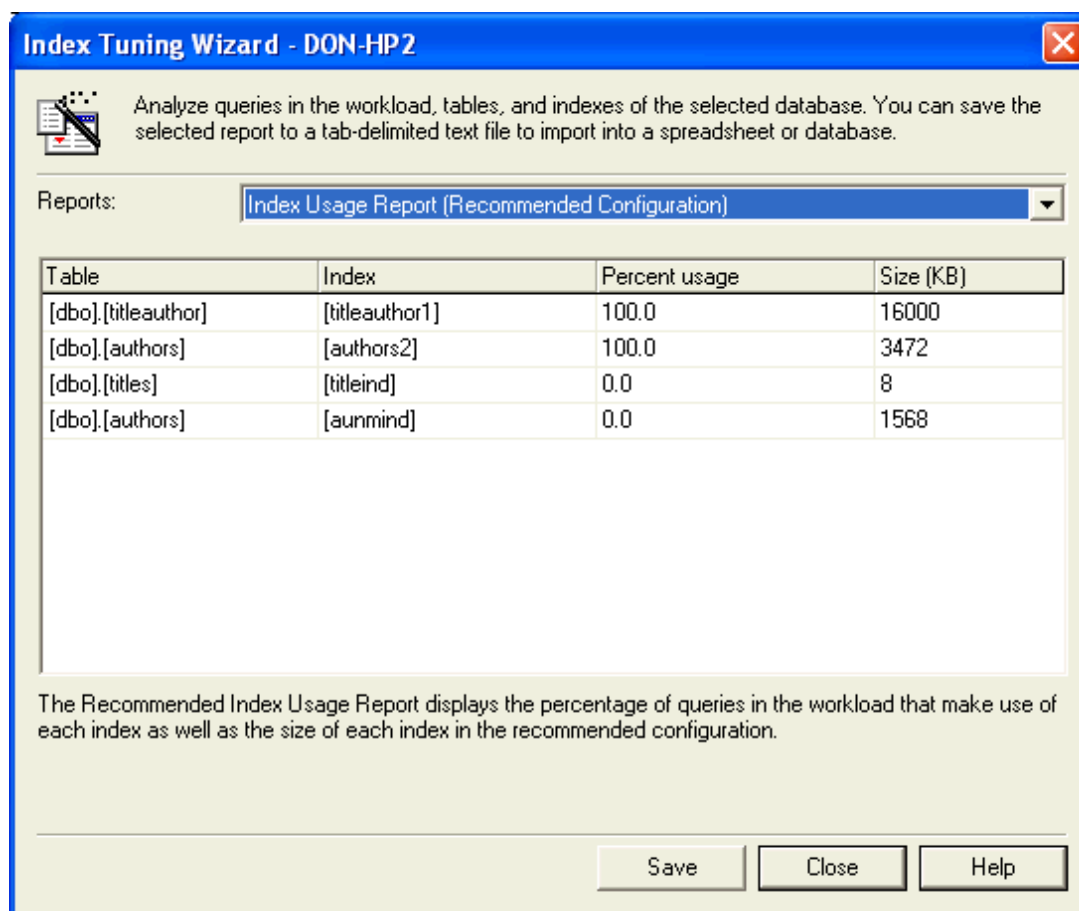
Figure 5.9: Indexes with a 70 percent or higher usage will make a major impact on your server's performance. Note that this screen shows an estimated size, in kilobytes, for each index.

Finally, on the Index Tuning Wizard's last screen, you'll have the opportunity to implement the wizard's recommendations. If you're running your trace on a development server, implementing the changes right away won't do any good. As Figure 5.10 shows, I prefer to save the recommended changes to a T-SQL script that I can then execute against any production servers that need the change. The script can also become part of my change-management documentation so that I (and other administrators) can see exactly what changes were made.
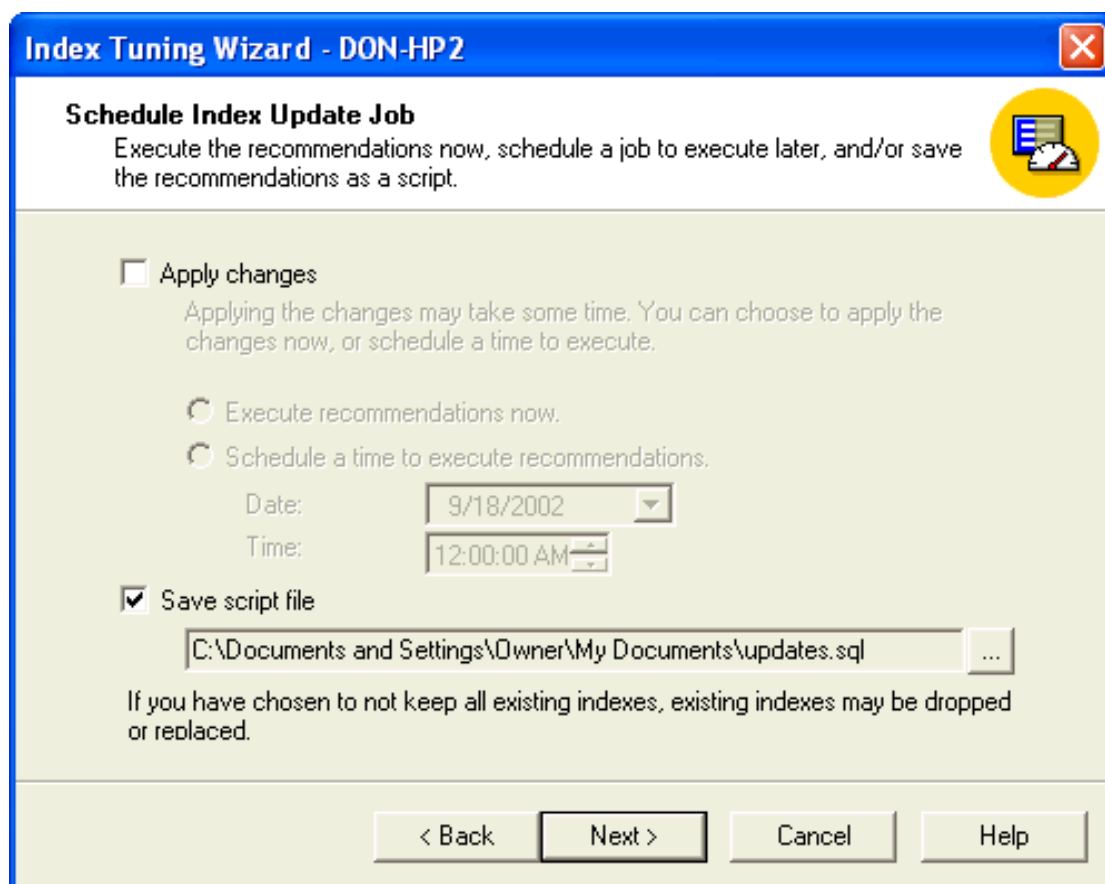
Figure 5.10: Saving the Index Tuning Wizard's recommendations to a script file.

Combined with trace files from SQL Profiler, the Index Tuning Wizard can be your best first shot at improving performance problems. At the very least, the wizard might recommend no additional indexes, meaning you'll need to look deeper into the trace data for possible improvements.

Summary

A key to getting the best performance from SQL Server is fine-tuning the queries you execute. You'll need to know how to read complex query plans and how to use SQL Profiler to capture the query plans from your production workload. In this chapter, I've shown you how to perform both of these important tasks. With a little bit of practice and patience, you'll be able to capture and interpret even the most complex query execution plans, and fine-tune those queries to achieve the best possible performance from SQL Server.

VERITAS™

Chapter 6: Scaling SQL Server for Performance

During your SQL Server administrative duties, you'll probably come to a point where, no matter how fine-tuned your queries are, your SQL Server computer simply isn't up to handling the work you need it to do. In other words, your server hardware simply isn't up to snuff, and you'll need to do something about it. You have two options: a bigger server or more servers. Getting a bigger server (or upgrading an existing one) is known as *scaling up,* while implementing additional servers is known as *scaling out.*

Scaling Up

Adding RAM, faster hard disks, and additional processors are all examples of scaling up—making an existing server more powerful, and therefore able to handle more work. SQL Server scales up quite nicely, taking advantage of additional processors, more memory, and so forth. However, there's still a hard-and-fast limit on how far scaling up can take you, because Windows itself has a limit on how much hardware it supports. That limit differs among Windows editions (Standard, Enterprise or Advanced, and Datacenter) and among Windows versions (Windows .NET Server 2003 can scale higher than Win2K, for example). You're also limited by available hardware, although you can usually find vendors selling servers that pack as much hardware as Windows can handle.

There are practical limits to scaling up, too. For example, if your database application makes heavy use of your network, adding more RAM or more processors to your SQL Server computer won't change the fact that your client computers are only connected by a 10MBps or 100MBps network. Network bandwidth is one of the most difficult components to scale up, simply because doing so requires changes to large numbers of client computers, to network devices (such as routers and switches), and to your actual servers.

Generally, though, scaling up is the easiest—if not always the cheapest—way to get more power out of a SQL Server application. Making an existing server more powerful, or simply buying a new, beefier server, allows you to continue using your database application's basic architecture. Price aside, scaling up can often be a quick fix for server performance.

Eventually, though, you'll hit a wall. Either your servers will be as powerful as you can make them, you'll have purchased the biggest servers money can buy, or you'll run out of money. Either way, you'll have scaled up as far as you can and the time will have come to scale out.

Scaling Out

Scaling out is typically a great way to improve the performance of an application. The goal of scaling out is to have multiple servers sharing a single workload, essentially making the servers work together as if they were a single, gigantic computer.

The most common example of scaling out is a Web farm. In a typical Web farm, you have multiple Web servers, each loaded with the exact same content. You can even think of the servers' content—Web pages and graphics—as a sort of database because the content is what users are requesting from the servers. Because each server has an identical copy of the data, it doesn't matter from which server users get data—users will get the same response from any of the servers. Web farm administrators use products such as Microsoft Application Center to keep the Web servers synchronized, ensuring that users get a consistent response no matter which server they happen to contact. Application Center can also perform load balancing, distributing user requests across the servers in the farm and making the farm seem like a single, gigantic Web server.

If performance in a Web farm starts to fall, you simply add more servers. Because Web serving isn't especially resource-intensive, many Web farms are built from small, inexpensive servers, making it quite practical to add a few servers when necessary. Figure 6.1 shows a typical Web farm, with many Web servers working together to handle user requests.
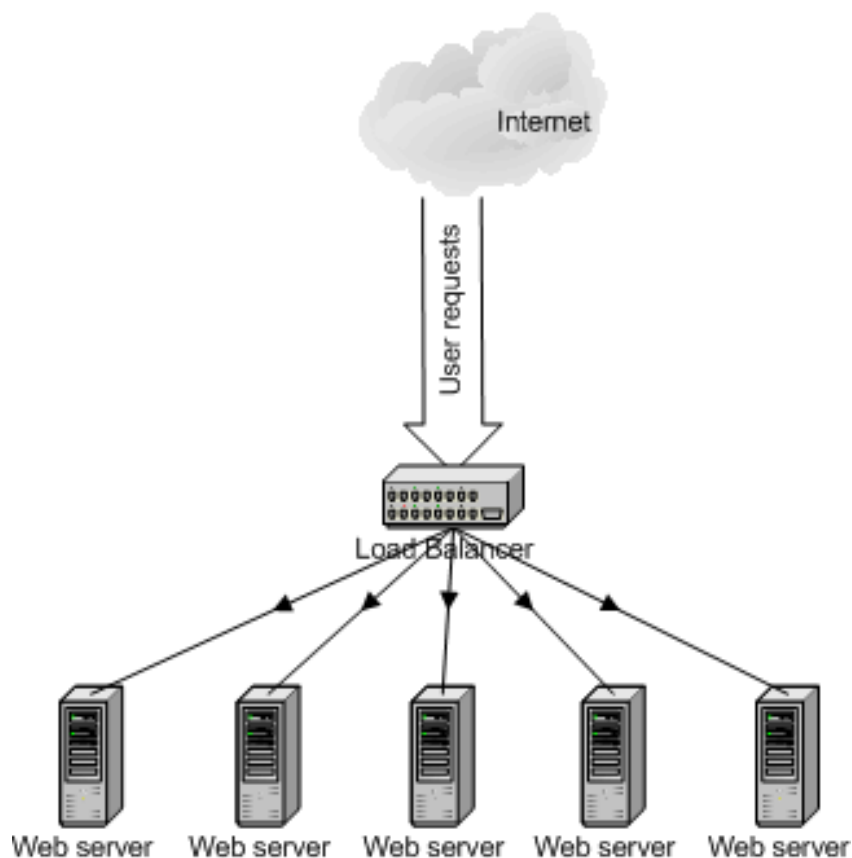
Figure 6.1: Web farms consist of multiple servers hosting identical content.

Unfortunately, the Web farm model doesn't work well for most database applications. The reason is that most database applications involve reading and writing data, while most Web sites simply involve reading data—Web pages—and sending it to users. Even a dynamic Web site doesn't allow users to directly change the site's content; instead, the site passes data changes through to a database server.

If you happen to have a database application that relies entirely on reading data, SQL Server is quite easy to scale out. Simply set up as many read-only SQL Server computers as necessary, and distribute requests for data across them. You can even use Windows' Network Load Balancing to distribute the user requests, if desired, or you can use external load-balancing solutions. You can use SQL Server replication to publish changes to your data from a centralized, writable server out to all the read-only servers. Figure 6.2 shows what such a SQL Server farm might look like.
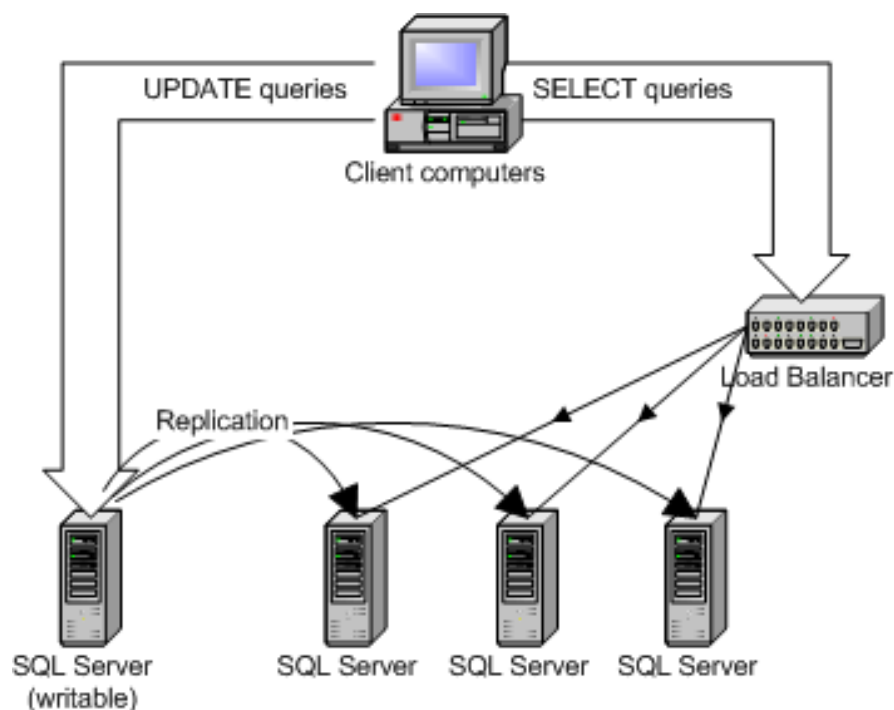
Figure 6.2: Creating a server farm for read-only database applications is a lot like creating a Web farm.

Sadly, not many of us deal with read-only applications. Most SQL Server applications require constant reading and writing of data, making scaling out more difficult. Why? Imagine that you decide to scale out by adding another, identical SQL Server computer to your network. You use a load-balancing solution, such as Application Center, to evenly distribute user requests across your two SQL Server computers. Everything will work fine so long as users are simply querying data; however, everything will go wrong as soon as someone *changes* data. The change will only be made to one SQL Server, resulting in the two servers having unsynchronized copies of the database. Users submitting queries will now receive inconsistent results, because the data on one server has changed. As users continue to modify data on one or another of the servers, the two servers will become more and more out of sync, resulting in ever-increasing inconsistency for user queries. (For information about why Windows Clustering isn't a solution for this problem, see the sidebar "Windows Clustering: Not for Scaling.")

You *could* use SQL Server replication between the two servers, setting things up so that changes on one server were immediately replicated to the other. In a busy environment, though, the constant replication between the two servers would quickly negate the performance benefits of scaling out to begin with. If you were to add a third or fourth server to the mix, performance would decline exponentially because each server would have to replicate changes with every other server in your farm. Figure 6.3 illustrates the replication topology for four SQL Server computers.
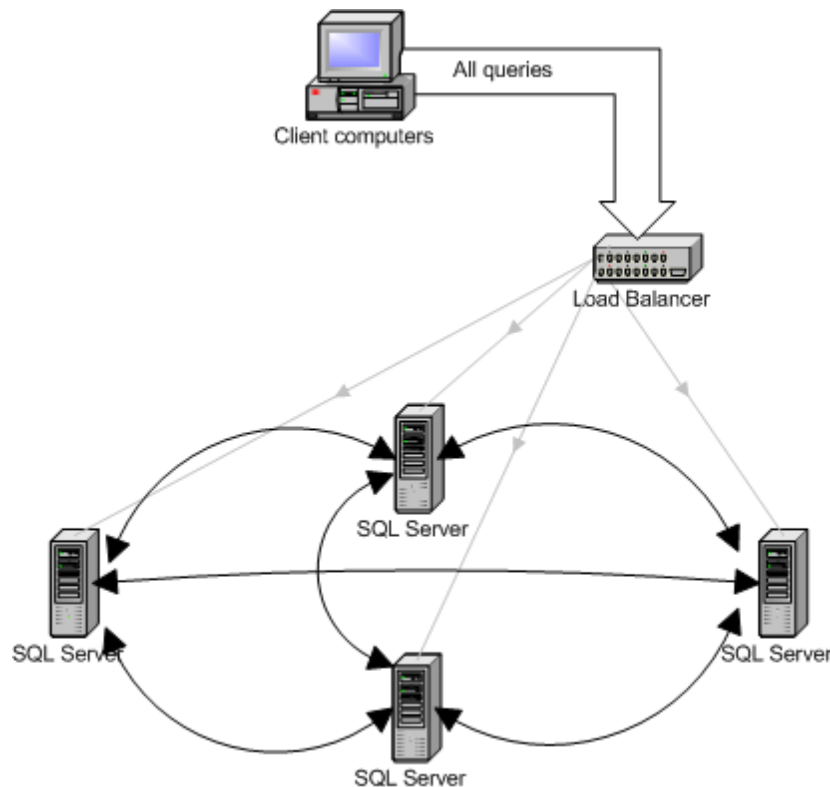
Figure 6.3: Replication of this kind would quickly become a performance problem.

Windows Clustering: Not for Scaling

Many administrators believe that Windows Clustering combined with SQL Server, Enterprise Edition (which supports clustering) is one way to scale out. Unfortunately, nothing could be further form the truth. Windows clustering *does* allow you to connect multiple servers, but those servers can't distribute any workload among themselves. Instead, each member of a cluster is essentially its own, independent SQL Server computer, and the cluster itself simply provides *reliability,* allowing cluster members to cover for a failed member.

If you plan to scale out by adding multiple servers, you can use Windows Clustering to add a level of reliability to those multiple servers. However, Windows Clustering itself doesn't provide a way to make two SQL Servers work together to handle a greater workload.

Techniques for Scaling Out

The problems with running a single database across multiple SQL Server computers are pretty complex. In fact, earlier versions of SQL Server didn't even try to address the problem, effectively limiting SQL Server's capacity to whatever a single server computer could handle. SQL Server 2000, however, offers a number of options that allow you to successfully scale out, distributing a single database across multiple SQL Server computers. These options include:

- Partitioning and federated databases—Partitioning is the only technique that has always been available to SQL Server administrators. Essentially, partitioning is simply dividing your database into logical parts and running each part on a dedicated SQL Server computer. Client applications have to understand which bits of the database live where, and maintain connections to each SQL Server that hosts a piece of the database. Partitioning is still the basic means of scaling out SQL Server, although new SQL Server features make it less painful. Federated databases are a special kind of partitioning.

- Linked servers—This SQL Server feature allows data on a remote server to be accessed as if it were local. Linked servers can help make partitioning less painful because clients can connect to a single server to retrieve all their data. When necessary, the main server contacts its linked servers to retrieve data from their parts of the database.

- Distributed views—Just as a regular view can combine columns from multiple tables into a single virtual table, distributed views can combine columns from multiple servers into a single, virtual table. Distributed views provide a way to logically recombine a partitioned database, allowing client applications to work with a single virtual table and not worry about how many servers are being accessed to actually obtain the data in the view.

Partitioned and Federated Databases

Applications such as Microsoft Commerce Server are designed with partitioning in mind. Commerce Server uses SQL Server to store Web site data, and its database is divided into distinct portions. For example, one set of tables maintains customer orders and the site's product catalog, another portion contains user profiles, while still another contains marketing campaign data. Because there are fairly few data relationships between these portions of the database, the different portions can easily be hosted on different servers. This type of partitioning is referred to as *vertical partitioning* because the database's columns are distributed but not the rows. In other words, all the columns relating to customer orders are kept on one server, and that server maintains all the rows for those columns. Columns relating to user profiles are stored on another server, which is responsible for all rows of data contained by those columns.

Commerce Server partitions its data by tables, and keeps closely related tables—such as customer orders and product catalogs—in the same partition. A more complex form of vertical partitioning is to split tables into multiple parts so that multiple servers handle the tables' data. This type of vertical partitioning is extremely complex and requires a great deal of additional programming to manage. Very, very few applications are built this way because of the extra overhead involved in managing the relationships between the distributed data.

> 🖉 Partitioning is a good thing! Commerce Server is just one example of an application that's built to handle a partitioned database. Microsoft BizTalk Server is another example, as it divides its data needs into four distinct databases that can be hosted on separate SQL Server computers.
>
> If you have any input into the design of new database applications, work hard to build in partitioning from the start. Reduce unnecessary data relationships and help software developers understand that their data might not always fit onto a single database server. Building applications that provide a built-in path for partitioning is a great way to ensure the best long-term application performance.

A more popular and easier-to-manage form of partitioning is *horizontal partitioning,* which is used to create *federated databases.* In a federated database, a group—or *federation*—of SQL Server computers is used to store a database. Each server contains every table and column for the entire database, but not every server contains every row. A simplistic example might include two SQL Server computers used to store customer information. The first server would store customers whose names begin with the letters A through M, and the second server would handle N through Z. Typically, a middle tier is used to process data and determine which of the federation member servers will handle a particular query. Figure 6.4 shows a typical application architecture utilizing a federated database
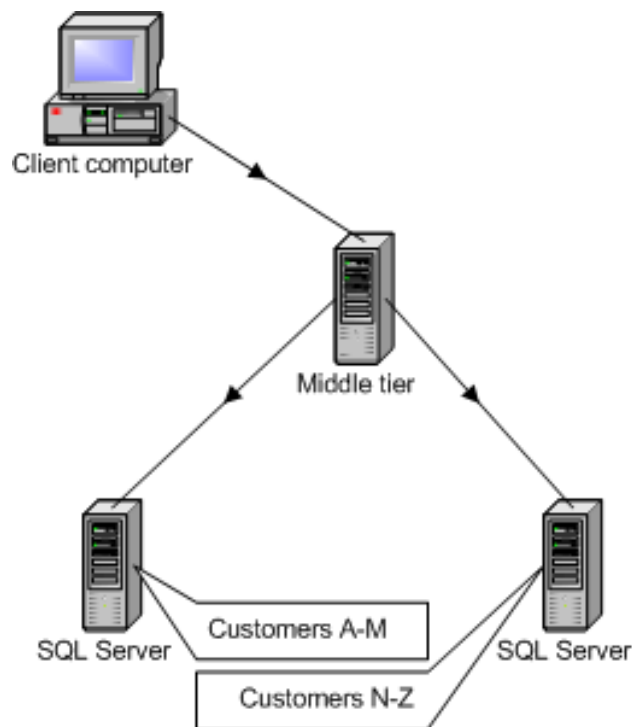
Figure 6.4: Federated database in a multi-tier application.

Distributed Views

With any kind of partitioned database, and most especially with federated databases, clients will need an easy way to work with data that is spread across multiple servers. SQL Server distributed views make this access possible, effectively creating a virtual table that contains data from multiple servers. Views used to combine horizontally partitioned data are referred to as *partitioned views,* and are new in SQL Server 2000. Figure 6.5 shows how a partitioned view can be used to logically recombine a federated database into a single virtual entity.
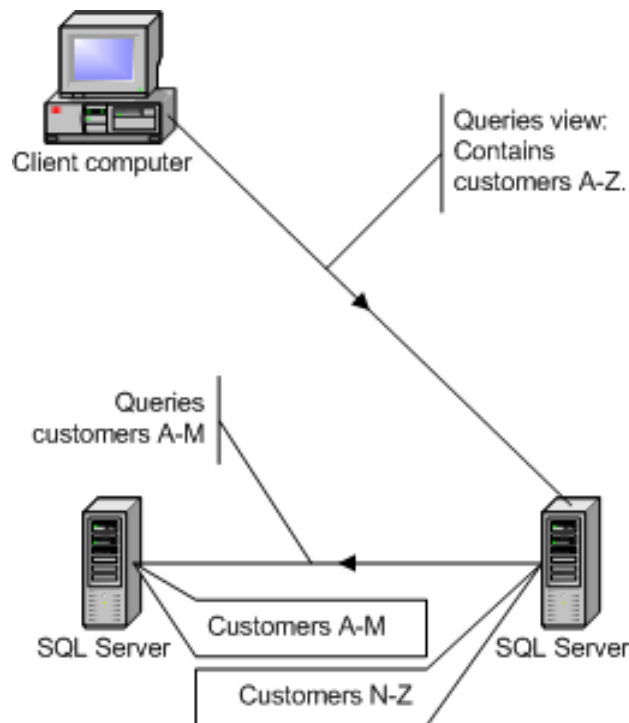
Figure 6.5: Using partitioned views to recombine federated data.

Linked Servers

SQL Server's linked servers feature allows you to create links to other OLE DB and ODBC data sources. Once linked, you can submit commands to the remote servers, include the remote servers' data in distributed queries, and so forth. The linked servers feature is typically used to link to other SQL Server computers (including older versions), Oracle databases, Access databases, and other ODBC sources.

> The linked servers feature basically turns SQL Server into a client as well as a database server. When a client application submits a distributed query, SQL Server fulfills as much of the query locally as possible. It then acts as a database client to the remote servers, querying data from them and using that data to fulfill the request from the application.
>
> Querying through linked servers isn't always very efficient. SQL Server has no control over what indexes or other techniques are used by the remote server to generate the required results.

VERITAS™

You can use SQL Server Enterprise Manager to create linked server definitions. Each definition represents a single remote server, and includes the security credentials necessary to access the remote server. To set up a new definition:

1. Open Enterprise Manager and expand a server group.

2. Expand a SQL Server and expand the Security folder.

3. Right-click the Linked Servers item, and select New Linked Server from the pop-up menu.

4. Type the name of the server to link to.

5. Select the appropriate server type.

Linked servers support a lot of functionality, including the ability to distribute transactions across remote servers. These capabilities make it easier to distribute a database across multiple servers, improving performance. However, linked servers can create a lot of performance overhead. Here are some tips for getting the most from this feature:

- Avoid distributed transactions across linked servers. Although this feature is convenient, it imposes a huge performance hit. Try to design your databases so that transactions can be completed locally.

- Ensure that all linked servers use the same collation type as your main server and set SQL Server's "collation compatible" option to True by using the sp_serveroption system stored procedure. Leaving this option to its default setting (False) will force the remote server to return an entire table in response to a query, and have the calling server apply any WHERE filters. Doing so is very inefficient; setting the collation option to True will allow the WHERE filter to be applied remotely before any data is transferred to the calling server.

- Avoid queries that include data conversion, the TOP clause, or datestamp or timestamp values. These operations cannot be completed on the remote server, and will force the calling SQL Server to bring back much more data than it would otherwise have to do.

Carefully used, linked servers can be one way to distribute data across multiple SQL Server computers. However, you'll often find that federated databases and distributed partitioned views will offer better performance when using multiple SQL Server computers to distribute query workload.

☞ Distributing workload: think big picture. Linked servers offer a means to distribute the work from a single query across multiple servers. Federated databases help distribute an application's overall workload across multiple servers; ideally, single queries are still handled by one server in the federation.

Database applications work in bulk, and distributing a single query isn't the best way to achieve a performance improvement. Using federated databases to evenly distribute an application's workload across multiple servers will result in a much better performance improvement and an ultimately more scalable architecture.

Other Techniques and Architectures

Depending on how your data is used, you might have other options for spreading the workload across multiple servers. For example, if you can partition your database and identify partitions that contain mostly read-only data, those read-only partitions can be spread across multiple SQL Server computers, and you can use load balancing to distribute work between them. One of the servers can host a writable copy of the database and distribute changes via replication to the other servers. This technique works best for data that doesn't change frequently, such as product catalogs, lookup tables, and so forth.

The key is to obtain a thorough understanding of how your data is used. Through this understanding, you can be creative in thinking of ways to spread your data across multiple servers.

Implementing Scale-Out Techniques

Although SQL Server offers a number of techniques for scaling out, I'm going to focus on federated databases. Federated databases are SQL Server's ultimate scale-out technology, and they let you create databases that can literally handle any amount of workload. In the next few sections, I'll show you how to design a federated database, create partitions on multiple SQL Server computers, create partitioned views, and modify your client applications to work with the new database architecture. In the end, you can have a highly scalable SQL Server architecture that provides top performance for even the most demanding applications.

Designing Partitions

The first step in creating a federated database is to intelligently partition your data. *Intelligently* doing so is a key requirement in partitioning, and you'll need to understand exactly how your database relations work, how users access your database, and so forth to make the right partitioning decisions.

Database architecture describes two types of partition schemes: *asymmetric and symmetric.* In symmetric partitioning, each server in the federation works about as hard as the others and contains about the same amount of data. For example, a database of customer information might be evenly divided among three SQL Server computers, creating a nicely symmetric division of labor. New records could be created in round-robin fashion so that each server maintains a roughly equal percentage of the total application data.

Symmetric partitioning is easy with unrelated tables but becomes more complicated in large, relational databases. Generally, what you want to distribute between the federation members are queries, and each query received by a member should be fulfilled only by that member. A good guideline is to have each member capable of handling 80 percent of the queries it receives without having to retrieve data from another member.

Suppose that in addition to your customers database, you have a database of orders placed by those customers. Simply dividing the orders evenly across three servers won't guarantee a symmetric partitioning of the data. For example, if a customer's record is on one server, the customer's order data might be spread across all three, resulting in less efficient queries. Figure 6.6 illustrates the problem.
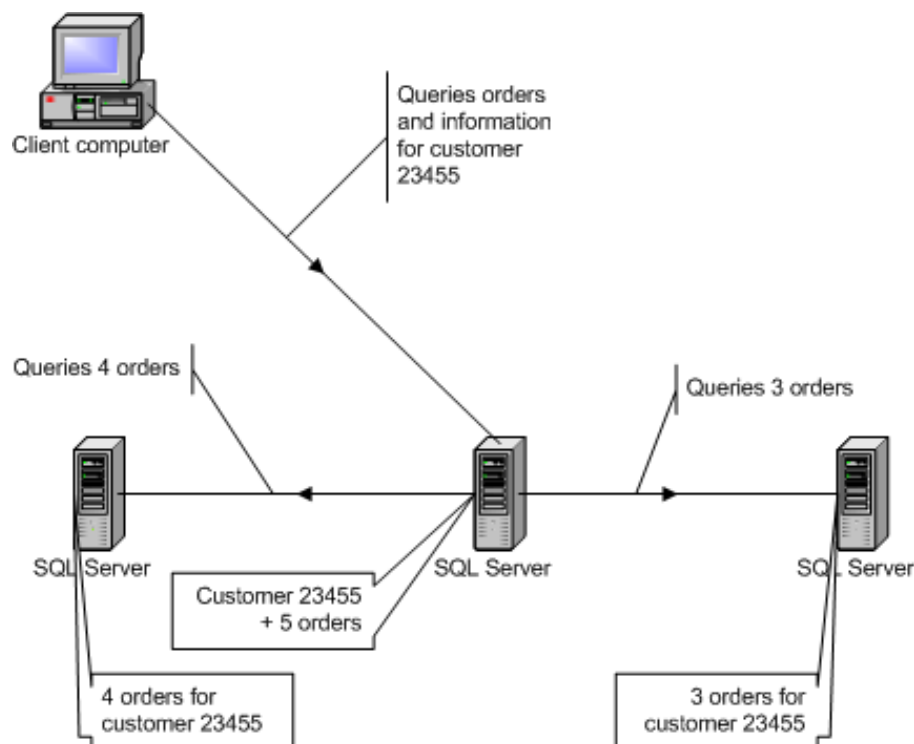
Figure 6.6: Unintelligently partitioned data will result in less efficient querying.

The solution would be to partition the orders database in such a way that each customer's orders are housed on the same federation member that has the customer's own database record. Then queries for a particular customer's orders could be handled by a single server. If the customers were well-distributed across the three servers, the application's overall workload would be evenly distributed, resulting in the best performance. Figure 6.7 illustrates this ideal solution.
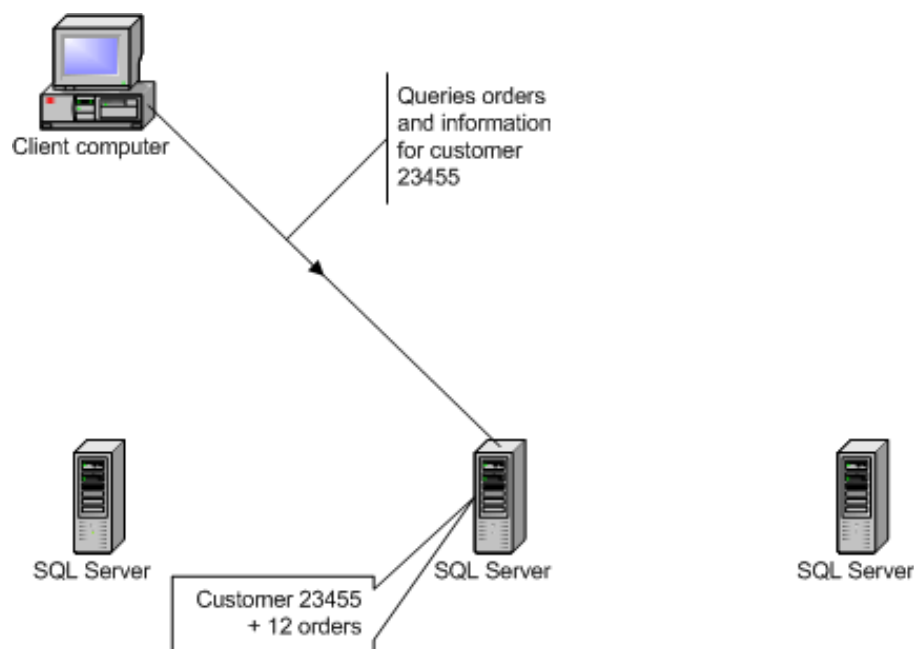
Figure 6.7: Intelligently partitioned data results in more efficient querying.

Suppose that you now add a products database to the mix. After all, orders are usually just an intersection between customers and products, with each order representing a specific customer's desire to obtain one or more specific products. How could you intelligently partition the products database among the members of your federation? You have three options:

- Somehow divide up customers by the product they order, and store those customers and their orders on the same server with a partition of the products database that contains the correct products. In reality, this setup will be close to impossible. Over the course of time, customers can potentially place orders from your entire product catalog, making it impossible to group customers with a subset of the product catalog.

- Don't partition the products database. Instead, settle for an asymmetric design in which one server hosts the products database. Build that server with better hardware to handle the extra load.

- Designate one server as the master, and give it a complete copy of the products database. Make that copy writable, and use replication to place full read-only copies on the other two servers. This option is probably the best solution of the bunch, as it provides each federation member with a complete copy of the products database and the ability to handle most queries locally. Because product databases aren't usually as write-intensive as customer or order databases, the replication traffic should be minimal. Figure 6.8 shows the proposed solution.
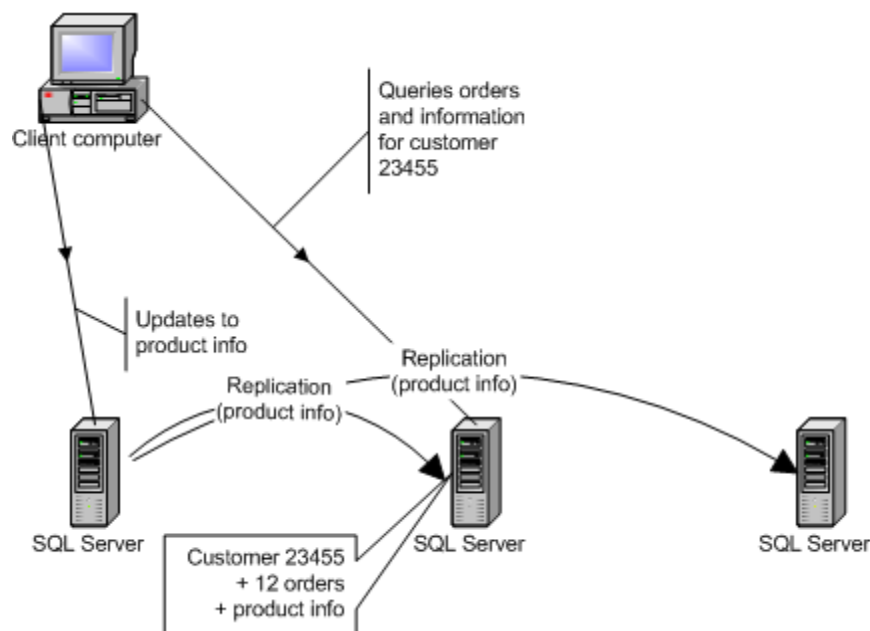
Figure 6.8: Distributing a read-only database to help improve local querying efficiency.

You'll also have to come up with some scheme for partitioning your data. Which customers will go on which server? In the case of customers, you might break them down alphabetically, with an even distribution on each federation member. Typically, though, you'll partition based on ID numbers. For example, you could designate the first server in the federation for customer ID numbers 1 through 10,000, the second for customers 10,001 through 20,999, and the third for 21,000 and higher. New records could be created in a round-robin fashion, alternating among the servers.

☞ Leave room for growth on all members. If you partition your tables by ID number, leave room for growth. For example, if your database currently contains half a million rows, and you will have four servers in your federation, assign each server to handle a half million IDs. Distribute the existing records equally among the four servers. This technique will fill each server's ID range by 25 percent, leaving plenty of room for evenly distributed growth.

Creating Partitions

Physically partitioning your data is usually pretty easy, especially if you leverage SQL Server's Data Transformation Services (DTS). For example, suppose you're starting with all your data on a single server and you want to distribute half of it to a new server, creating a federated database. You decide to distribute all even-numbered records to the new server, leaving odd-numbered records on the original server. With DTS, you can create a *package* that creates the appropriate database objects on the new server, copies the appropriate rows, then removes those rows on the original server. The package can be saved and scheduled for an off-hours execution, minimizing impact on productivity and eliminating the need for you to hang out at the office all night.

🖉 Partitions aren't anything special. Remember that partitions aren't special types of databases. Partitions are simply a logical division of your data, like having all the boys and girls go to opposite sides of the room at the school dance. The partitioned data is stored in normal databases, so you can create partitions by using any tools—including DTS—that can copy data from one database to another. Other tools you might use include SQL Server Bulk Copy, the Import/Export Wizard (which is really just a lightweight user interface for DTS), or even SQL Server replication.

Creating Partitioned Views

Suppose you've partitioned your customers database across four servers. Each server handles 25 percent of the customers and uses ID number ranges to determine which server has a particular record. Server1 handles ID numbers up to 25,000, Server2 up to 50,000, Server3 up to 75,000, and Server 4 up to 99,999. Each server hosts the data in a database named Customers.

☞ Using the same database name on each federation member makes querying easier.

When an application needs to query data, it would normally have to first figure out which server contained the data it wanted. For example, if the application needed customer record 89901, the application would have to dynamically build a query and submit it to Server 4. Building applications in this fashion is time-consuming and very inefficient. Fortunately, partitioned views provide a much better way of handling the problem. Consider the view created on Server1 by the code in Listing 6.1.

VERITAS™

```
CREATE VIEW AllCustomers
AS
 SELECT * FROM Customers_A
  UNION ALL
 SELECT * FROM Server2.Customers.dbo.Customers_B
  UNION ALL
 SELECT * FROM Server3.Customers.dbo.Customers_C
  UNION ALL
 SELECT * FROM Server4.Customers.dbo.Customers_D
```

Listing 6.1: Creating a partitioned view on Server1.

The resulting view on Server1 seems to contain all the customer records from each of the four servers. The view can be treated as a table and queried normally:

```
SELECT * FROM AllCustomers WHERE CustomerID = 89901
```

This query will retrieve the desired row of data regardless of which server actually contains it.

Some key terminology when dealing with partitioned views:

- A *local partitioned view* combines data from multiple tables on the local server.

- A *distributed partitioned view* combines data from multiple servers. The referenced tables are called *member tables,* and the servers containing the tables are *member servers.* The tables are contained within a *member database.*

Generally, a federated database will have the same distributed views on each member server, allowing applications to contact any one server and receive the same query results. Also, while SQL Server 7.0 supports distributed views, SQL Server 2000 allows those views to be *updatable,* meaning the view functions as a table and can accept INSERT, UPDATE, and DELETE queries. These updatable distributed views are crucial to truly federated databases, as it allows any member of the federation to accept update queries. If you want to be able to submit updates to any server in the federation, the view must exist on all of those servers; you can house the view on a single server if you're willing to submit all updates to that server only.

☞ Convenience vs. Performance. Distributed views make it easy to create federated databases with very little change to your client applications. However, having applications blindly fire queries at a particular server isn't the most efficient way to work with a federated database. Ideally, applications should have some logic that tells them which server physically contains most of the data the application needs, allowing the application to send queries directly to the appropriate server and minimizing the amount of distributed querying SQL Server has to perform.

Modifying Client Applications

You might not have to make a lot of changes to your client applications in order to take advantage of a federated database. For example, suppose you currently have a single SQL Server computer named Server1 and several client applications that send queries to it, as Figure 6.9 illustrates.

VERITAS™
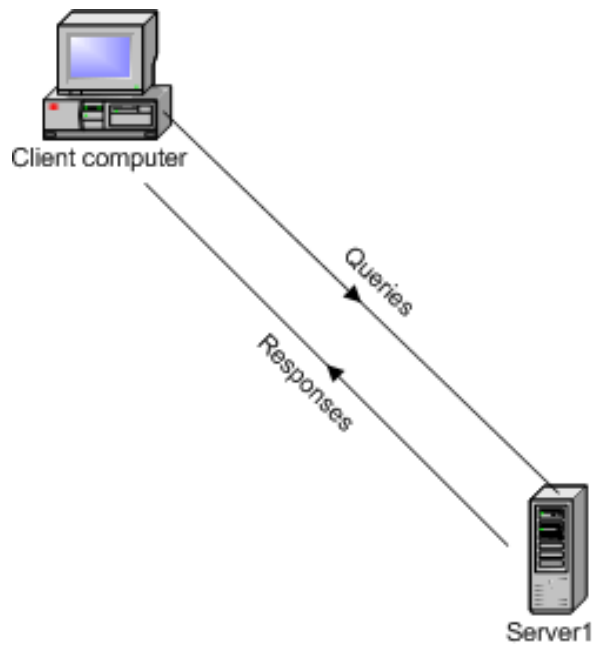
Figure 6.9: Basic client-server application architecture.

All of your clients are hard coded to submit their queries to Server1, so when you create your federated database, you'll want to create distributed partitioned views on Server1. Those views should have names that correspond to the old table names, which your client applications are used to querying. The distributed partitioned views will substitute for the original tables, pulling data together from Server1 and the new servers in the federation, as Figure 6.10 shows.
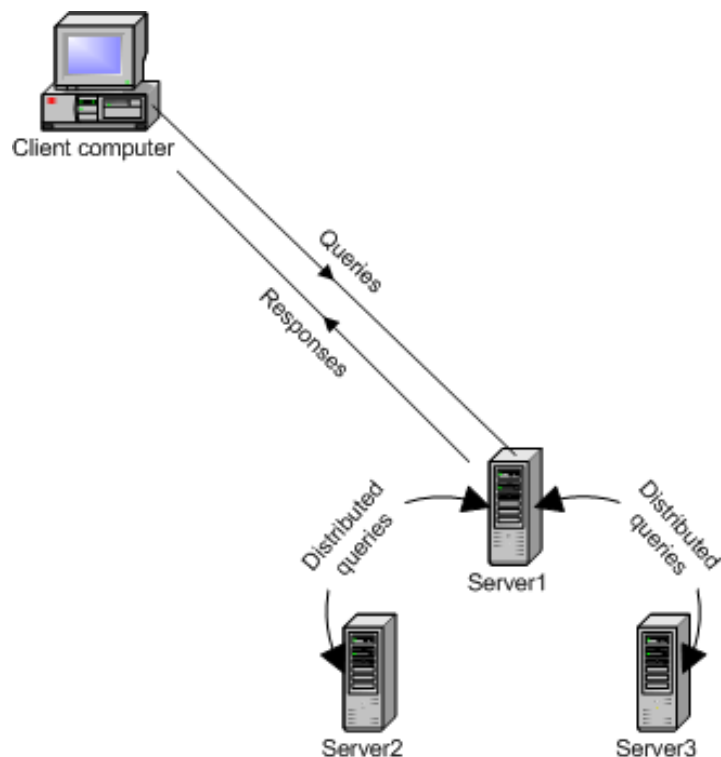
Figure 6.10: Using views to mimic a client-server architecture in a federated database.

VERITAS™

One of the biggest challenges will be having applications determine which servers will receive newly created records. One way to handle this problem is to create a master table of available ID numbers along with the server that "owns" each ID number. When a new record needs to be created, the application can retrieve the next ID number in sequence, which will tell the application on which server the new record should be created. Obviously, implementing this behavior through some central means—such as a middle-tier component or even a stored procedure—will be easier than coding the behavior into a client application that must be distributed to your users.

Without changing your client application, though, you can still maintain a federated database. All new records will be created on Server1 because that is where your clients are sending their queries. However, you can run evening SQL Server Agent jobs that run DTS packages to move new data to the appropriate federation members. Automating the process requires just a few extra steps:

- Add a Boolean column to each table with a default value of True. When new rows are created, this column will be automatically filled in, allowing you to easily spot new rows.

- Create a DTS package that moves new rows—based on the value of the Boolean column—to the appropriate federation member, based on ID numbers or whatever other criteria you used to partition the database in the first place. The DTS package should delete the row on Server1 once the row is copied to another server.

- Create a SQL Server Agent job that runs the DTS package each evening, once a week, or however often is appropriate. The job should also set all of the Boolean columns to False after all new rows have been processed.

Figure 6.11 shows how the data might flow in such a process. With a little creativity, you can have a true federated database with little or no changes on the client application.
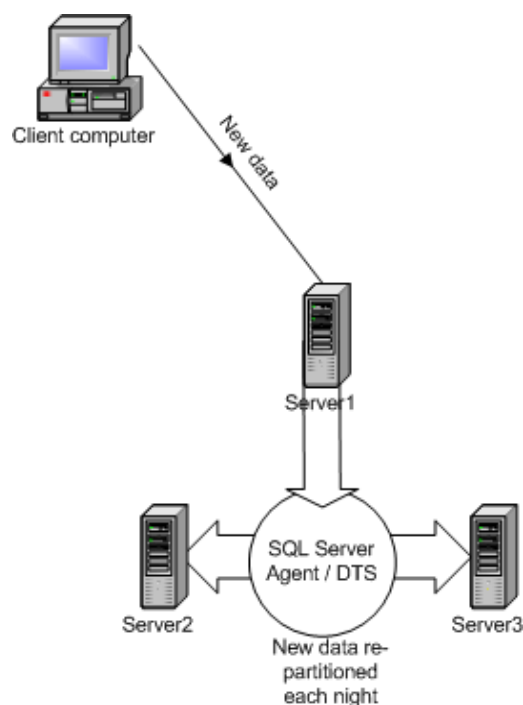
Figure 6.11: Automating the partitioning of new data.

You can improve the performance of your application even more by adding a middle-tier layer, as Figure 6.12 shows. Having applications submit queries to the middle tier (which can be independently scaled to support larger numbers of clients) allows the middle tier to have built-in business logic that determines which of the federation members should receive the query. Essentially, the middle tier acts as a distributor for queries, accepting queries from clients and distributing those queries to the appropriate back-end SQL Server computers. The middle tier accepts the data results from SQL Server and returns them to the clients.
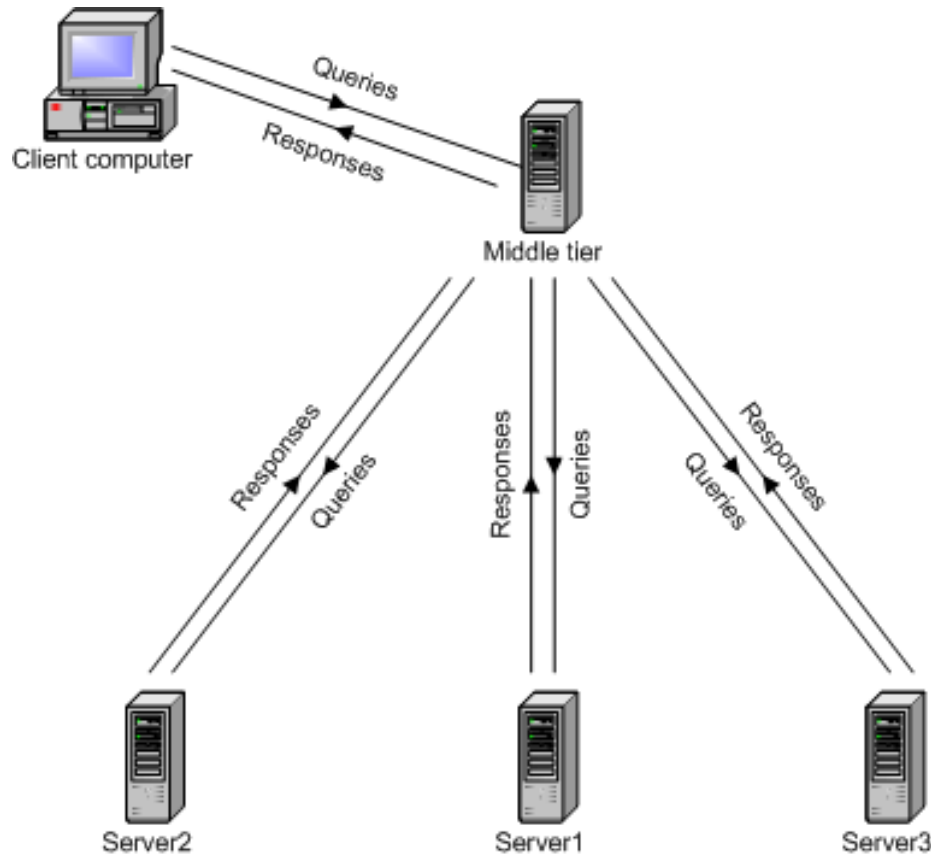
Figure 6.12: Using a middle tier to make a federated database more efficient.

Because the middle tier can have a centralized set of business rules, it can direct queries to the back-end servers that physically contain the requested data, if possible. Doing so will reduce the amount of distributed queries that each SQL Server computer has to process, increasing overall application performance. For queries that require distributed querying, such as a report that includes data from across the federation members, the middle tier can still utilize distributed partitioned views to create a single virtual table to work with.

Perhaps most importantly, a middle tier's business rules can distribute newly created data across the federation. This functionality maintains the level of partitioning originally planned for in the federation's design.

Reliability

The distributed architectures I've presented in this chapter can go a long way toward improving the overall performance of a database application. However, none of them particularly address reliability. For example, in a federated database, the failure of any server in the federation can make the entire database completely useless. You'll need to carefully consider your business and reliability needs when building any distributed solution.

You might, for example, decide to build a federated database and architect each federation member as a server cluster, eliminating any single point of failure. Or, for less money, you might maintain a "hot spare" server that can stand in for any single member of the federation that fails. Whatever you decide, make sure that your quest for improved database application performance doesn't ignore the business need for reliable access to your data.

Summary

In this chapter, you learned that scaling up is an easy way to improve SQL Server performance, but that scaling up has some very definite limits due to Windows' hardware support, cost, and hardware availability. Ultimately, scaling *out* is the best means of improving SQL Server performance, by having multiple servers work together to handle your workload. SQL Server 2000 includes a variety of technologies to make scaling out easier, the most effective of which is federated databases.

A federated database allows you to involve multiple SQL Server computers in a single database by partitioning that database's data across the available servers. Distributed partitioned views act as virtual tables, effectively recombining the distributed data into a single entity and making it easier to create federated databases with minimal client application reprogramming.

VERITAS™

Chapter 7: Improving Performance from the Client Side

SQL Server performance isn't something you can always adjust server-side. After all, SQL Server is simply accepting requests from client applications and processing those requests; how the requests are structured and submitted are an important factor in how well SQL Server performs. In this chapter, I'll help you identify some of the most common performance problems to which client applications contribute, and offer some tips about how to improve client applications to provide better SQL Server performance.

Many of the suggestions I'll offer are new techniques designed to reduce communications with SQL Server and to use Microsoft's data access technologies more efficiently. Some of these performance-improvement tips will help you understand the technologies' internal workings more clearly so that you can work around any internal gotchas. Still other suggestions won't necessarily improve actual performance but might help improve the *perceived* performance of your applications.

> ✎ Improving client application performance can require reprogramming! Unfortunately, squeezing better performance from a client application can often require a great deal of reprogramming, depending on how bad the client application is written to begin with. Although performance-improvement reprogramming might be more work than you're able to invest, you can at least use the tips and suggestions in this chapter to improve your client applications in their next major versions, or to improve new client applications as you're writing them.

Identifying Client Bottlenecks

Figuring out which portion of a client application is causing performance problems can be incredibly difficult. You'll generally find yourself relying on comments by the application's users, which can offer a completely different outlook on performance than you might expect (see the sidebar "Perception = Performance" for more information).

Client bottlenecks can come in several shapes and sizes:

- Network performance—This type of bottleneck occurs when you try to transmit too much data across your network infrastructure. Alleviate the problem by fine-tuning queries to return less data or to use design techniques that avoid the need to query data from SQL Server.

> 📖 See "Store Lookup Tables Client-Side" later in this chapter, for an example of how to avoid querying data.

- API selection—Older applications might be using older data access APIs, such as Microsoft's Data Access Objects (DAO). These can create several performance bottlenecks that can be alleviated only by moving to a more modern API, such as ActiveX Data Objects (ADO) or ADO.NET.

- API behavior—Neither ADO nor ADO.NET are perfect; they can both be used in sub-optimal ways that result in poorer performance. In this chapter, I'll highlight some of the more common "worst practices" and show you how to avoid them.

Perception = Performance

Most of the time you'll probably find yourself investigating client application performance only when the application's users complain that the application is running slowly. Be careful in those situations, as user perceptions might require a completely different performance tuning methodology than a systems administrator's perceptions of performance problems would require. For example, suppose your application executes a SQL Server query that returns a thousand rows of data into a list, which users can browse through. Users complain that the list takes too long to populate. You've fine-tuned the query as best as you're able, and yet users are still complaining about the application's performance. There could be a couple of things going on. One problem could be that the client application language, such as Visual Basic, simply takes a long time to load a thousand items into a list box. No amount of query tuning will improve that problem! Another problem could simply be the amount of time required to transmit a thousand rows over the network from the SQL Server, and again, no query tuning can help you.

You could, however, modify the application to query only a hundred rows or so, using SQL Server's TOP clause. For example, instead of executing a query such as SELECT CustomerName FROM Customers, you might execute SELECT TOP 100 CustomerName FROM Customers. This query would allow you to populate the list box more quickly. You would need to write additional code that queried another hundred rows when the user scrolled to the bottom of the first hundred. Effectively, you're still querying the original thousand rows, but you're doing so in hundred-row chunks, so your client application appears to be performing more quickly.

This new query technique might not be the most efficient for SQL Server to process because it now has to optimize and execute ten queries instead of one. Your users, though, will get a response back more quickly, so the application will *appear* to be performing faster. Little tricks like this can often make users happier. I'll cover more such techniques in "Performance Tips for Client Applications" later in this chapter.

Performance Hits with DAO and ODBCDirect

Older client applications written to use Microsoft's DAO and ODBC should be rewritten. DAO and ODBC used in conjunction with SQL Server result in particularly poor client-side performance for a number of reasons. First, bear in mind that DAO and ODBCDirect were among Microsoft's first data access technologies and were design for use with Jet (Microsoft Access) databases. They were never designed to work with powerful relational database management systems such as SQL Server and they don't take advantage of many of SQL Server's features. ADO, however, takes full advantage of SQL Server's array of features and provides the best performance for using low-overhead features such as stored procedures.

Because DAO was designed for Jet, it works great for Microsoft Access databases. However, DAO doesn't include any built-in capability to talk to SQL Server. It does have the ability to talk to generic ODBC data sources, and Microsoft provides an ODBC driver for SQL Server, so DAO can access SQL Server data. To do so, however, DAO has to use the Jet engine to translate its internal workings into ODBC calls, which must pass through the SQL Server ODBC driver and be translated to SQL Server-specific calls. The data returned must likewise pass through the ODBC and DAO layers back to your application. The result is slow performance.

Microsoft came up with an alternative, Remote Data Objects (RDO), which allow programmers to directly access ODBC. ODBCDirect is a set of extensions to DAO that allows DAO to utilize RDO in the background, making it easier for DAO programmers to access SQL Server data without using the Jet database engine as an intermediary. Unfortunately, this acronym soup of DAO, RDO, and ODBC still requires a lot of layers and doesn't provide optimal performance with SQL Server.

Few programmers write new applications by using DAO. However, many companies upsized old Access-based applications to SQL Server and retained the old DAO-oriented applications. The performance of those applications can be greatly improved simply by migrating the older DAO code to ADO. Microsoft has a number of white papers about migration techniques that you can access at http://www.Microsoft.com/data.

> ✎ OLE DB = ODBC? A little-known fact is that OLE DB, the most modern way to access SQL Server, is actually built on top of ODBC. OLE DB eliminates many of the middleman layers required to access SQL Server, and provides the fastest and easiest-to-use means of doing so. C++ programmers can work directly with OLE DB; ADO's purpose is to provide an object model that makes OLE DB accessible to Visual Basic, VBScript, and so forth. Similarly, ADO.NET provides a set of classes that make OLE DB accessible from within the managed environment of the .NET Framework.

Still confused about where DAO, RDO, ADO, and ODBC fit in to your world? Check out the helpful API comparison chart Microsoft provides at http://msdn.Microsoft.com/library/default.asp?url=/library/en-us/architec/8_ar_ad_3hih.asp. You'll see that DAO is considered a *legacy* technology, meaning it isn't undergoing further development and provides a low degree of developer control. It also provides limited support for SQL Server 2000 features and no support for advanced features such as OLAP and XML. RDO is in a similar state: Although considered an existing technology, it provides poor support for SQL Server 2000. ADO, however, provides the best support for SQL Server 2000 features and is an ideal way to interact with SQL Server.

Improving ADO Performance with SQL Server

Applications written in Visual Basic, C++, and other COM-oriented programming languages should use ADO to access SQL Server. ADO is a part of the Microsoft Data Access Components (MDAC) toolkit, which also includes OLE DB providers for SQL Server 2000 and a number of other data sources. OLE DB is a newer data access technology, and it replaces the older ODBC used in the past.

> ☞ Get the latest MDAC. Whenever possible, work with the latest version of MDAC for the best performance and stability. You can download the latest version (currently 2.7) from http://www.microsoft.com/data. Windows computers can have multiple versions of MDAC installed simultaneously, which allows applications written to use an older version of MDAC to coexist with applications written to use a newer version.
>
> Speaking of versions, there are a number of production versions of MDAC available: 2.7, two versions of 2.6, three versions of 2.5, and two versions of 2.1. Applications are generally written to a specific version, so it's not uncommon for client computers to have all of them installed!

Because ADO is responsible for retrieving and updating data on behalf of your application, it pays to know a little bit about how ADO works internally. Knowing where ADO's internal performance faults lie will allow you to program around them, thus increasing the overall performance of your applications.

Cursor Types

ADO supports a number of cursor types. The default and most popular is the forward-only snapshot, which allows you to move from one row of data to the next, but not backward. The data in the cursor is static, and doesn't change even if the underlying data on the server is updated while you're working with the cursor (hence the name *snapshot*). The forward-only cursor type is by far the fastest for ADO to create and process because it involves a one-time hit to the server to retrieve the data and create the cursor. Forward-only cursors reside entirely on the client, eliminating any ongoing server overhead. However, the forward-only cursor might not always be useful within your application, and so you might find yourself using another ADO cursor type.

Of the remaining cursor types, dynamic and static are perhaps the most used. A static cursor is a lot like a forward-only cursor in that a static cursor creates a snapshot of data from the server. However, a static cursor also allows you to move backward through the rows of data within the cursor. Static cursors create a higher performance hit than forward-only cursors, and that performance hit is largely client-side, requiring extra memory and processing time to maintain and operate the cursor.

Dynamic cursors also provide backward-scrolling capabilities and offer the ability to automatically update themselves to reflect changes in the underlying data. Thus, the cursor is constantly changing to reflect changes made to the data by other users. As you might expect, dynamic cursors can create a significant performance impact on client applications as dynamic cursors require constant communications with the data source.

☞ Generally, I recommend using forward-only cursors whenever possible. If you need to be able to scroll back and forth through the rows in a cursor, use a static cursor. Avoid using dynamic cursors, as they impose a major performance hit compared with the other cursor types. Keep in mind that static cursors usually require the client to download each and every row all at once, which is pretty inefficient for large cursors. If you do need to work with static cursors, keep the number of rows small.

Cursor Location

One often-ignored feature of ADO is its ability to remote cursors. By default, ADO creates client-side cursors, meaning the data rows within the cursor are maintained on the computer that created the cursor. Normally, that would be the computer running your client application. Client-side cursors are a form of distributed computing; each client computer creates and maintains its own cursors, spreading out the processing load of your application across those computers.

ADO can, however, create server-side cursors. These cursors are stored and manipulated on the SQL Server that contains the data. The client application tells ADO when to retrieve the next or previous row of data from the cursor, and ADO communicates with SQL Server to make it happen. Some programmers believe that server-side cursors provide a performance benefit. From a purely client-side perspective, they do: The client computer doesn't have to dedicate much in the way of memory or processing power to deal with a server-side cursor. Figure 7.1 illustrates this distribution of cursor processing.

VERITAS™
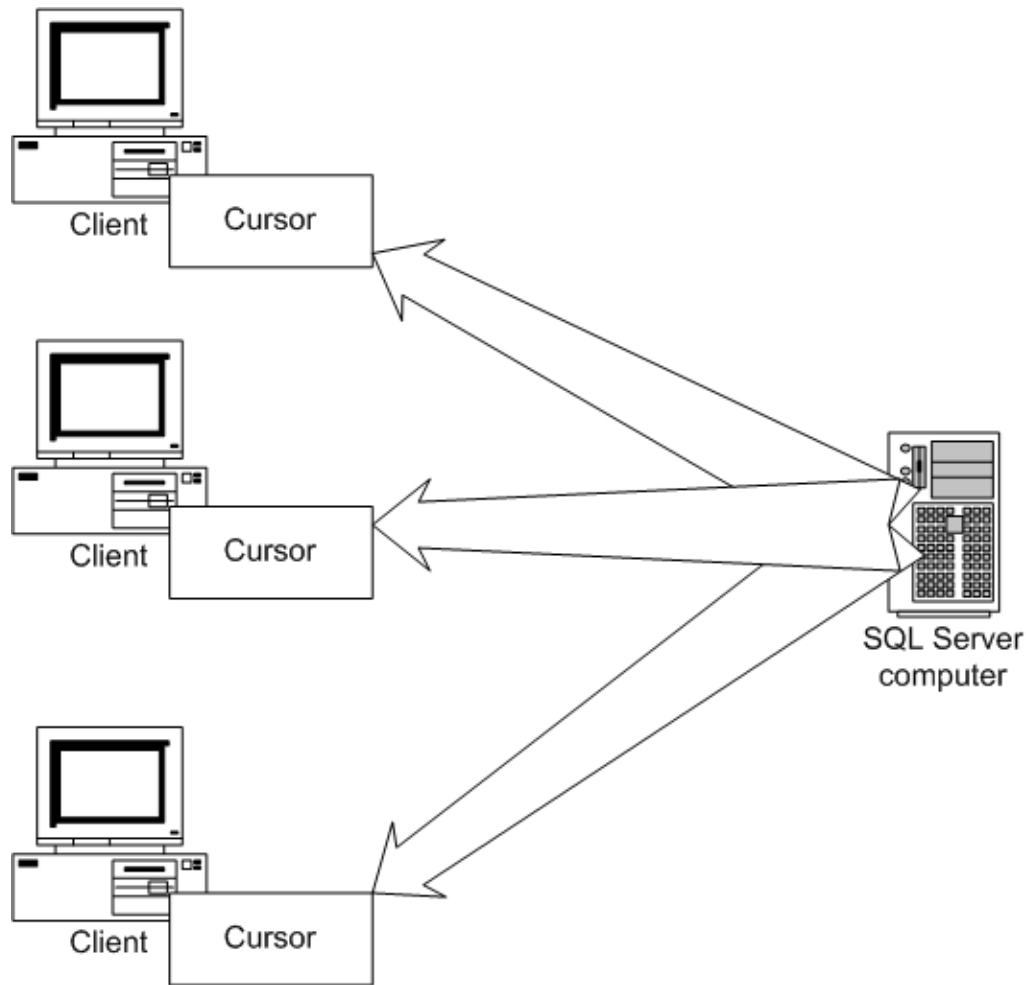
Figure 7.1: Client-side cursors require the computer running your client application to manage the application's cursors.

From a larger application perspective, however, it doesn't make much sense to have your SQL Server maintain and operate the cursors for every single instance of your client application. If you have a hundred application users, your SQL Server computer will be required to deal with hundreds of cursors. That's a significant load of processing to consider! Figure 7.2 illustrates how server-side cursors centralize cursor processing.

VERITAS™
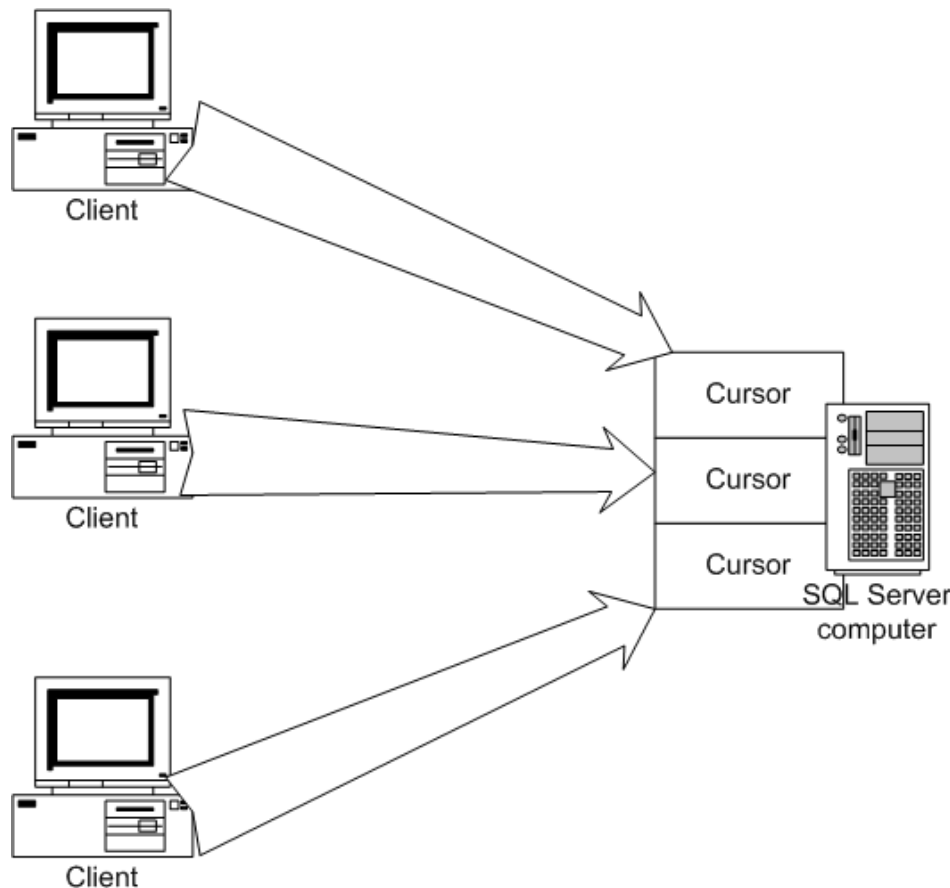
Figure 7.2: Server-side cursors concentrate cursor processing on a single back-end computer.

Server-side cursors are stored in SQL Server's TempDB database. Using TempDB involves quite a bit of overhead processing, which helps make server-side cursors even less desirable from a performance perspective.

I prefer to use client-side cursors. You can ensure that ADO does so by specifying the CursorLocation property of Recordset objects. For example, in Visual Basic, you might use the following:

```
Dim oRS As New ADODB.Recordset
oRS.CursorLocation = adUseClient
```

🖉 You can't set advanced options such as cursor locations and cursor types when you're using Visual Basic 6.0's Data Environment features. Although the Data Environment provides an easy way to connect to and utilize data sources, it's fairly inefficient when compared with applications that simply create ADO objects in code, as I've done in the previous example. I generally recommend avoiding the Data Environment at all costs for a number of reasons, including its poor performance.

It's a good idea to specify a client-side cursor if that's what you want. The reason is that some cursor types—such as dynamic cursors—are server-side by default. Of course, client-side cursors are always static. That's because ADO can't maintain a constant connection between the client and the server to support a dynamic cursor. If you are in a situation in which a dynamic cursor is a must, then you're forced to use server-side cursors.

VERITAS™

Executing Stored Procedures

Many programmers simply use the Connection object's Execute method to execute stored procedures, saving the results to a recordset object if necessary:

```
Dim oRS As ADODB.Recordset, oCN As New ADODB.Connection

oCN.Open "MyDataSource"

Set oRS = oCN.Execute("EXEC sp_MyProcedure('Customers')")
```

Although this method works perfectly well, it doesn't take full advantage of ADO's ability to integrate tightly with SQL Server. ADO provides a Command object that you can use to execute stored procedures on SQL Server. This object provides the best performance for executing stored procedures (although it's definitely not the only way to do so) and takes full advantage of SQL Server's capabilities:

```
Dim oRS As ADODB.Recordset, oCN As New ADODB.Connection

Dim oCmd As New ADODB.Command

oCN.Open "MyDataSource"

oCmd.ActiveConnection = oCN

oCmd.CommandText="sp_MyProcedure"

oCmd.CommandType=adCmdStoredProc

oCmd.Parameters.Append "Customers"

Set oRS = oCmd.Execute
```

This method feeds the command and its parameters to SQL Server in a way that allows SQL Server to optimize execution. The Command object can also accept return parameters from a stored procedure, which the Execute method of the Connection object can't do.

Explicitly SELECT Columns

One of the worst things you can do to SQL Server is to execute wildcard queries such as this:

```
SELECT * FROM Customers
```

Doing so requires ADO and SQL Server to first figure out the names of each column that you'll be selecting, then to rebuild your query to explicitly list each column. For better performance (and more maintainable code), explicitly list the columns you want to query, even if you want to query all of the columns from a table:

```
SELECT Name,Address,City,State,ZIP FROM Customers
```

Of course, you can increase performance even more by only selecting the columns that you actually plan to work with. Reducing the columns you select will, at the very least, reduce the amount of data that SQL Server has to transmit to the client application.

☞ This trick works within stored procedures, too. In fact, *anytime* you're writing a SELECT query, you should explicitly list columns rather than using the SELECT * shortcut.

VERITAS™

Working with Columns

Under ADO, programmers usually work with columns by referring to the column names. For example, suppose you've created a recordset object named oRS that contains two columns, Name and Address. You could access these columns' data using the following code:

```
oRS("Name")

oRS("Address")
```

However, ADO itself can't access columns by name. Instead, it has to access columns by their ordinal number within the recordset's collection of columns. You can save ADO a step by referring to those ordinal numbers yourself:

```
oRS(0)

oRS(1)
```

Naturally, coding in that fashion makes your code less maintainable because you have to memorize ordinal numbers. Using constants, however, would allow you to directly refer to column ordinal numbers while still keeping your code easy to read:

```
'Declarations
Const COL_NAME As Integer = 0
Const COL_ADDRESS As Integer = 1


'Accessing Columns
oRS(COL_NAME)
oRS(COL_ADDRESS)
```

Is the extra work worth it? It's debatable. Purists in search of the best possible performance believe that accessing columns by their ordinal numbers makes sense, but there are a number of potential issues with the method. First, you should never use this method if you're generating recordsets based on a SELECT * query because the order of the columns might change over time as your database is restructured. If you're doing the right thing, however, and explicitly listing columns in your SELECT queries, then you have control over the columns' ordinal numbers and can be sure they won't change, even if the database is redesigned in the future. Using constants to access columns by their ordinal numbers isn't much extra work. However, doing so doesn't usually provide a measurable performance benefit so you might not think it's worth your while.

Minimize Round-Trips

From a purely performance standpoint, you'll want to minimize the number of queries you execute against SQL Server. Each time you submit a query through ADO, a number of steps occur:

1. ADO sends a query to SQL Server through an ADO connection object.
2. Your computer's networking stack breaks down the request into network packets.
3. The packets are sent across the network.
4. The SQL Server computer must reassemble the packets into the original request.

Chapter 7

5. SQL Server must process the query. If the query isn't utilizing a stored procedure, it has to be optimized prior to execution.

> 📖 Chapter 4 explains how the Query Optimizer works.

6. The query results are broken down into network packets.

7. The packets are transmitted across the network.

8. The client computer must reassemble the packets into their original data stream.

9. ADO receives the data stream, which is in the form of a Tabular Data Stream (TDS), and converts it into a recordset object.

That's a lot of steps, and although most of them aren't terribly intensive, you don't want to repeat them any more than necessary. You'll need to strike a balance between querying just the data you need to reduce SQL Server's workload, and getting all the data that you'll need to use, to avoid repeated round-trips to SQL Server for more data.

Be aware that querying all of your data up front may result in users having a poor perception of your application's performance (see the sidebar "Perception = Performance" earlier in this chapter). You'll need to decide which is more important—user perception or actual measured performance—in your environment.

Avoid Data Source Names

Data Source Names (DSNs) are a handy way to create database connections. You can use your computer's ODBC32 control panel, which Figure 7.3 shows, to create a named DSN, and then easily open it in your application code.
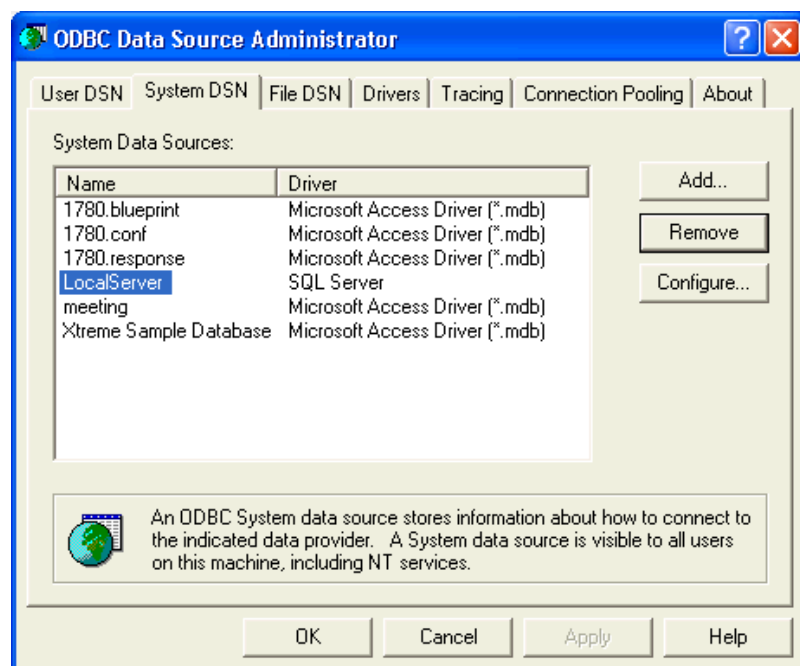
Figure 7.3: Managing system DSNs through the ODBC32 control panel.

realtimepublishers.com™

140

VERITAS™

The problem with DSNs is that they are inherently ODBC-related and cannot use the faster, more efficient OLE DB that is designed to work best with ADO. Rather than using DSNs, write complete connection strings:

```
oConnection.ConnectionString = _
 "driver={SQL Server};" & _
 "server=srv;" & _
 "uid=user;" & _
 "pwd=password;" & _
 "database=database"
```

Coding connection strings also makes your application more self-contained and portable because you don't have to worry about first setting up DSNs on client computers in order for your application to function properly.

Improving ADO.NET Performance with SQL Server

You should realize that the .NET Framework, which includes ADO.NET, is very new. Developers are only starting to use .NET for new projects. That means tips and tricks about improving ADO.NET performance are still few and far between because there isn't a huge body of experience to tell us which techniques are good and which are bad. Fortunately, Microsoft offers some excellent ADO.NET pointers, and enough companies are experimenting (or starting their first big projects) with ADO.NET to provide some good performance tips.

Use the SQL Server .NET Data Provider

ADO.NET includes a number of data providers that can access SQL Server data: OLE DB.NET, ODBC.NET, SQLXML, and so forth. Of the available options, the native SQL Server .NET data provider is fastest. It communicates with SQL Server by using SQL Server's native TDS, and is about 30 percent faster than the other providers. The provider is also compatible with SQL Server 7.0, making it ideal for use in older environments. However, the provider won't work with SQL Server 6.5; use the OLE DB.NET provider to talk to SQL Server 6.5 servers.

☞ You'll find the SQL Server .NET data provider in the System.Data.SqlClient namespace.

VERITAS™

Choosing Between the DataSet and DataReader

ADO.NET provides two objects, DataSet and DataReader, that you can use to retrieve and work with data. The DataReader is designed to move through data (forward only) without caching the data in the client's memory. Effectively, you're working with a stream of data that you just need to spin through quickly. DataReaders aren't ideal when you need to do a lot of processing on each data row or when you need to wait a long time between retrieving rows (such as waiting for user input). The reason is that DataReaders keep their connection open, which consumes both client and serve resources. Also, DataReaders are read-only, so you can't perform any updates to them. DataSets, however, are much like ADO recordsets, and represent an in-memory data structure that contains your query results. DataSets are actually populated by using a DataReader to pull over the desired rows. Unlike a DataReader, DataSets cache their contents in-memory, allowing you to work with data for an extended period of time without consuming unnecessary server resources.

Avoid Convenience Features

.NET's DataAdapter object has a Fill method, which allows you to fill a DataSet with rows. One of the Fill method's overloads accepts parameters indicating a starting record and a maximum number of records, allowing you to fill a DataSet with, say, records 50 to 100. This technique is extremely useful when generating pages of information, as it allows you to easily pick up pages of data for display.

Unfortunately, the Fill method actually returns your *entire query*, then filters for the specified records. Imagine, then, that you're querying a thousand rows from SQL Server and displaying ten rows at a time. If your user pages through all one hundred rows, you will have queried all one thousand rows, one hundred times—a huge performance hit. Although the overloaded Fill method is convenient, it isn't smart from a performance perspective.

To achieve the same results in a more performance-friendly fashion, you could query data in sets of ten rows by using a WHERE clause in the query to restrict your results to just the rows you want to display. You might, for example, filter by ID number with a WHERE clause such as

```
WHERE CustomerID BETWEEN 1 and 10
```

Or, if the total amount of data you're querying isn't huge, simply pull the entire query into a DataSet, then display the data ten rows at a time without requerying SQL Server for each new page of data.

What if your data doesn't include some unique ID that you can use in a WHERE clause? You can combine the SELECT statement's TOP predicate with an embedded SELECT statement to create pages of results. For example, suppose you have a thousand rows of data and you want to display the third page of rows, assuming ten rows per page. Just use a query like this:

```
SELECT TOP 10 CustomerName,Address FROM

  (SELECT TOP 30 CustomerName,Address FROM Customers

   ORDER BY CustomerName ASC) AS TableA

ORDER BY CustomerName DESC
```

Essentially, the subquery is selecting the top 30 rows of data, then the outer query is just selecting the top ten of those—providing you with the third page of data.

Managing Connections

.NET is a managed environment, meaning it takes care of resources such as memory allocation for you. That means you don't need to explicitly worry about releasing unused resources, because eventually .NET will perform its *garbage collection* and automatically release out-of-scope and other unused resources, all automatically. However, it's still a good idea to explicitly close and release resources, especially ADO.NET objects, once you no longer need them. That way, you'll prevent the need for garbage collection to clean up after you, and you'll retain more control over your application's performance. Specifically, be sure to close Connection, Recordset, and Command objects once you're finished with them and before they go out of scope. Also, bear in mind that these objects do not automatically close when they go out of scope. Therefore, you need to be very careful to explicitly close them. Finally, Microsoft claims that the garbage collection process won't necessarily close Connection objects for you, which is yet another reason to explicitly close these objects when you're finished using them.

Another benefit of cleaning up after yourself is that you'll help SQL Server better manage connection pooling. For example, suppose you create three Connection objects, then allow them to fall out of scope. Because .NET doesn't immediately perform garbage collection, the connections used by those Connection objects can remain open. If you later create three more Connection objects, your client computer will have a total of six connections open, and each connection will consume SQL Server resources. By explicitly closing the Connection objects when you're finished with them, you'll release the connections to SQL Server and allow it to better manage pooled connections and its own server-side resources.

The C# language provides a shortcut (via the using statement) that allows you to clean up after yourself without a lot of extra work. Listing 7.1 provides an example.

```
string sConnection = "Data Source=localhost;";

using (SqlConnection conn = new SqlConnection(sConnection))
{
 SqlCommand oCmd = conn.CreateCommand();
 oCmd.CommandText = "SELECT CustomerName FROM Customers";
 conn.Open();

 using (SqlDataReader dr = oCmd.ExecuteReader())
 {
  while (dr.Read())
    do something with each row of data here
 }
}
```

Listing 7.1: An example of the using construct.

The using construct will automatically call the objects' Dispose method when the construct ends, effectively closing the object and releasing its resources. Unfortunately, the using construct isn't available in VB.NET.

It's also important—both in ADO.NET and ADO—to use the *exact same* connection string for all Connection objects. Doing so will allow connection pooling to work properly; using different connection strings will result in new connections to SQL Server, which can use unnecessary server resources. (For more information about connection pooling, see the sidebar "Connection Pooling.") A common way for connection strings to be different is the server name. The original programmer of an application might use the server's name, and a later programmer making modifications might use the server's IP address (which is actually a bit faster, as it doesn't require name resolution to occur, but will require reprogramming if the address ever changes). That minor difference will result in a new connection to SQL Server, defeating the performance advantages of connection pooling.

Connection Pooling

Connection pooling is a way for client applications to minimize the number of connections they make to SQL Server, thus reducing performance overhead on the server. Generally, every new Connection object you create—whether in ADO or ADO.NET—results in a new connection to SQL Server, and the corresponding utilization of server resources to maintain the connection. With connection pooling, however, ADO (and ADO.NET) only creates a new physical connection for every new connection string. When a new Connection object is created with the same string as an existing object, ADO simply "hooks" both objects to the same physical connection. The result is multiple Connection objects in code, which can be convenient, all using a single physical server connection, which saves resources. Connection pooling will still open new physical connections for busy applications that need multiple simultaneous connections to the server, but connection pooling will generally ensure the fewest possible physical connections, helping conserve server resources.

The Fill and Update methods of the .NET DataAdapter object can implicitly open and close connections if necessary. Generally, you should allow the DataAdapter to do so if you only have one Fill or Update to execute. For multiple executions, however, allowing the DataAdapter to implicitly control the connection status will reduce client application performance and consume unnecessary resources on the server. In those instances, you should explicitly open the connection and close it when you're finished. If you're performing transactions, you should *always* explicitly open and close the connection. Listing 7.2 shows a VB.NET example.

```
Public Sub MyTransaction(oDA As SqlDataAdapter, _
 oCN As SqlConnection, _
 oDS As DataSet)

 oCN.Open()
 Dim oTrans As SqlTransaction = oCN.BeginTransaction()
 oCmd.Transaction = oTrans

 Try
  oDA.Update(oDS)
  oTrans.Commit()
 Catch e As Exception
  Try
   oTrans.Rollback()
  Catch
   'couldn't roll back transaction;
   'add appropriate handler code here
  End Try
 End Try

 oCN.Close()
End Sub
```

Listing 7.2: A VB.NET example of explicitly opening and closing a connection.

VERITAS™

Notice the use of the Open and Close methods of the Connection object to open and close the connection, rather than implicitly allowing the Update to open and close the connection. The implicit open wouldn't work because the connection must already be open in order for the BeginTransaction method to work properly.

Stored Procedures

ADO.NET provides a SqlCommand object that you can use to submit stored procedures for execution. In fact, this object is the only object that lets you execute a stored procedure (unlike the older ADO). Normally, ADO.NET has to analyze your command to see whether it is a stored procedure; you can boost performance a bit by setting the object's CommandType parameter to StoredProcedure. Doing so will prevent the need to parse the command.

Also, be sure to optimize the use of SqlCommand objects for your back-end data source. For example, the SqlCommand object includes a Prepare method, which instructs the data source to optimize the command for multiple calls. This functionality is especially useful when you'll be executing the same command over and over, as it lets the data source know that it would be a good idea to cache the command's execution plan for faster optimization after the first call.

The Prepare method is unnecessary with SQL Server 2000, however, which optimizes all commands and caches their execution plans. SQL Server 7.0 doesn't use that behavior, making the Prepare method desirable.

Avoid ADO

Finally, avoid using ADO in .NET applications. It's certainly possible to do so; although ADO is COM-based, the .NET COM Interop Layer will let you easily use the familiar ADO programming model from within .NET applications. However, your data access will be passing through a number of additional software layers, making it slower and less effective than if you simply used ADO.NET. The main reason you might see ADO used in a .NET application is if the application was originally written in a language such as Visual Basic 6.0, then migrated to VB.NET without a great deal of recoding.

Note that I suggest to avoid using ADO in .NET applications rather than tell you simply not to use it at all. The reason is that certain ADO capabilities, most notably ADO Multidimensional (ADOMD), aren't yet available as native ADO.NET classes. If you're writing an application that needs some of ADO's more advanced capabilities, including multidimensional support, you're currently stuck using the necessary COM object libraries through the .NET COM Interop Layer.

Third-Party Data Tools

Much of .NET can be pulled out and replaced with your own components, and it should come as no surprise that ADO.NET is just as extensible. Many third-party companies, such as DataDirect Technologies (http://www.datadirect-technologies.com), are producing native .NET data providers. DataDirect's Connect for .NET is 100 percent .NET-managed code, and provides a native data connection to Oracle and Sybase databases. A number of other companies offer similar tools and providers, making it possible to create high-performance database applications regardless of your back-end database type.

Performance Tips for Client Applications

Technique is sometimes more important than technology when creating high-performance database client applications. In the next few sections, I'll offer some advice and examples of how you can design applications that impose less overhead on SQL Server—thus increasing the number of clients a single SQL Server computer can support. Many of these techniques have been possible for years, although they're made easier by the technologies present in the newest versions of MDAC and the .NET Framework.

Store Lookup Tables Client-Side

One fairly easy way to improve client application performance and reduce SQL Server processing is to store lookup tables client-side. Most database applications include a number of different lookup tables, such as accounting codes, salesperson names, and so forth. Typically, these codes don't change frequently, so there's no sense in requerying them from SQL Server each time you need to populate a drop-down list box with the lookup tables' contents.

Instead, query all of your application's lookup tables when your application launches. You can store the tables in disconnected recordsets so that they exist completely client-side and don't impose any overhead on SQL Server. If users need to refresh the lookup tables (suppose a new accounting code is added mid-day), they can simply close and re-open the application.

You can go a step further, too, by adding some intelligence to your application. Suppose you maintain a master table that lists each lookup table and the last date it was updated. You could then write your client application to do the following each time it started:

1. Query all rows from the master table, detailing when each lookup table was last updated.

2. Retrieve a local copy of the master table, which can be persisted in XML on the client.

3. Compare the queried master table with the local one. For any lookup tables that have changed recently, requery those tables from SQL Server. Persist the tables locally in XML files on the client.

4. For all unchanged lookup tables, reload them into recordset objects from persisted XML files on the client.

Listing 7.3 shows a Visual Basic 6.0 example that uses ADO, and shows how this might work, assuming you have three lookup tables named AccountCodes, CustomerTypes, and OfficeLocations. This example assumes that you've already created an ADODB connection object named oCN that connects to your SQL Server database.

```
'Create Recordset objects
Dim oAccountCodes As New ADODB.Recordset
Dim oCustomerTypes As New ADODB.Recordset
Dim oOfficeLocations As New ADODB.Recordset
Dim oQueriedMaster As New ADODB.Recordset
Dim oLocalMaster As New ADODB.Recordset

'Query the master table
oQueriedMaster.Open "SELECT TableName,LastUpdate FROM LookupMaster", _
 oCN

'Load the local master copy
'Assumes it already exists
```

```
oLocalMaster.Open "c:\application\lookupmaster.xml", ,adOpenStatic

'Run through the master table
Do Until oQueriedMaster.EOF

 'Compare the local copy date
 oLocalMaster.Find "TableName = '" & _
  oQueriedMaster("TableName")
 If oQueriedMaster("LastUpdate") > oLocalMaster("LastUpdate") Then

   'Need to requery the table from SQL Server
   Select Case oQueriedMaster("TableName")
    Case Is = "OfficeLocations"

      'query the table
      oOfficeLocations.Open "SELECT LocationID,LocationName " _
       & "FROM OfficeLocations", oCN

      'persist the table for future use
      oOfficeLocations.Save "c:\application\offices.xml", adPersistXML

    Case Is = "CustomerTypes"

      'query the table
      oCustomerTypes.Open "SELECT TypeID,TypeName " _
       & "FROM CustomerTypes", oCN

      'persist the table for future use
      oCustomerTypes.Save "c:\application\types.xml", adPersistXML

    Case Is = "AccountCodes"

      'query the table
      oAccountCodes.Open "SELECT CodeID,CodeName " _
       & "FROM AccountCodes", oCN

      'persist the table for future use
      oAccountCodes.Save "c:\application\codes.xml", adPersistXML

   End Select

 Else

   'Use the table's local copy
   'Assumes it already exists
   Select Case oQueriedMaster("TableName")
    Case Is = "AccountCodes"
      oAccountCodes.Open "c:\application\codes.xml", ,adOpenStatic

    Case Is = "OfficeLocations"
      oOfficeLocations.Open "c:\application\offices.xml", ,adOpenStatic

    Case Is = "CustomerTypes"
      oCustomerTypes.Open "c:\application\types.xml", ,adOpenStatic

   End Select

 End If
```

```
 oQueriedMaster.MoveNext

Loop

'Save the latest version of the master locally
oLocalMaster.Close
oQueriedMaster.Save "c:\application\lookupmaster.xml", adPersistXML
oQueriedMaster.Close
```

Listing 7.3: A Visual Basic 6.0 example that uses ADO.

This technique will help minimize traffic to and from SQL Server, and will improve both SQL Server performance and client application performance. Users will still have the latest copies of the lookup tables each time the application launches, but they won't have to query SQL Server unless the data has changed. You can ensure that your lookup master table gets updated properly by applying triggers to your lookup tables: whenever a change to a lookup table is made, the trigger updates the lookup master table with the current date and time.

Avoid Blocking

Blocking occurs when one application's connection is holding a lock, and thus preventing another application connection from obtaining its own lock. SQL Server itself has no control over this situation and relies on well-behaved client applications to avoid blocking. A common cause of blocking is long-running transactions in which a number of rows, or even entire tables, are locked by one connection, preventing other connections from obtaining the locks they need. Users experience blocking as slow performance because their client applications essentially stop responding while waiting for the block to clear. Figure 7.4 illustrates how blocking can make client applications appear unresponsive.
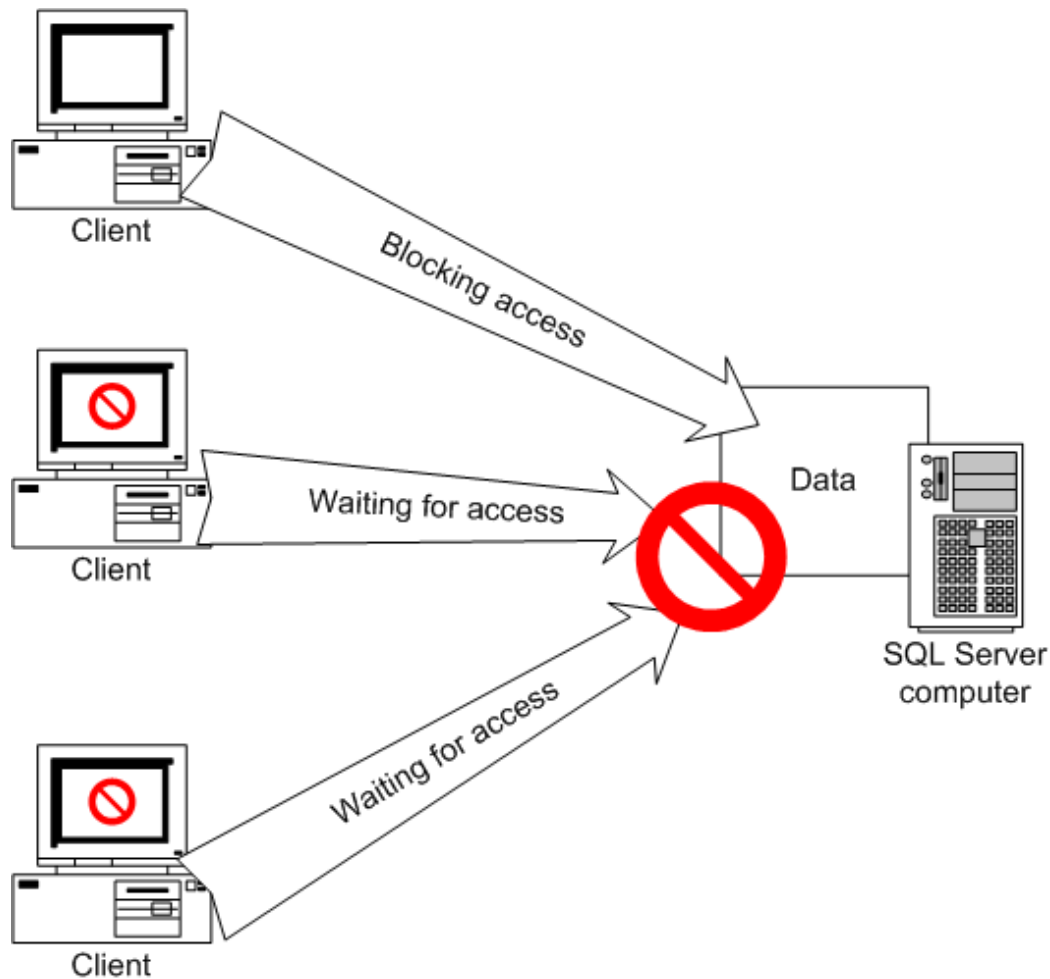
Figure 7.4: One client can block others, making the overall application appear to have poor performance.

Good application design is the only way to avoid blocking. Some easy-to-implement tips include:

- Don't create applications that require user input in the middle of a transaction. You shouldn't start a transaction with SQL Server until the client application has enough data to finish the transaction. If something occurs in the middle of the transaction and user input is required, cancel and roll back the transaction, gather the necessary information from the user, and start again.

- Provide a way for users to gracefully cancel long-running queries, rather than allowing them to simply quit your application. Quitting the application may not release SQL Server connections, and an administrator will have to manually cancel the connection and release the block.

- Design applications that encourage shorter queries. For example, don't create a search form that allows users to leave a large number of fields blank as wildcards. Doing so will result in longer-running queries, which can result in blocking.

Stress testing is the only way to ensure that your application is well-behaved. Use stress-test tools to simulate the expected production workload in your environment with typical queries from your application and see whether blocking becomes a problem. If necessary, redesign your applications so that blocking is less likely to occur.

Summary

Client applications play an important role in SQL Server performance. Poorly written applications will place a heavier load on SQL Server, resulting in a SQL Server computer that can support fewer clients. Well-written applications, in contrast, minimize their impact on SQL Server, allowing a single SQL Server computer to support many more clients. Whether you're using COM-based development in Visual Basic or managed development in the .NET Framework, the information in this chapter should help you design and develop better-behaved applications that help SQL Server realize its full performance potential.

Chapter 8: Getting the Best Performance Bang for Your Buck

In the previous seven chapters, I've covered a number of tips and techniques for improving SQL Server performance. Chapter 1 explains how SQL Server's transaction-based database engine works, giving you some insight into how performance comes into play. Chapter 2 explores the various ways you have to measure SQL Server performance and introduces some third-party performance tools that you can use. Chapter 3 shows you how to identify performance bottlenecks and introduces a basic performance-tuning methodology that you can use to achieve repeatable, predictable results. Chapters 4 and 5 focus on query tuning and explain how SQL Server's query optimizer works. Chapter 6 shows you how to improve performance by making SQL Server bigger and better, and Chapter 7 covers performance from the client side.

What I haven't covered so far are some of the myriad unrelated tips and techniques that can improve SQL Server performance. All of these techniques take advantage of information I've already covered, such as query optimization and client application design, but these techniques don't all fit together into a "story." Instead, they're just things that you need to keep in mind as you're working with SQL Server and SQL-based client applications.

☞ Want to get more tips and techniques? One of the best resources on the Web for SQL Server performance tips is http://www.sql-server-performance.com. The site is organized into categories, such as query tuning and T-SQL performance, and contains tips and strategies submitted by site visitors.

Database Design Tips

Obviously, the design of your database has a tremendous impact on SQL Server's performance. A poorly designed database will require more complex and longer-running queries, which will take the query optimizer additional time to figure out. The following sections offer normalization and denormalization, physical, and miscellaneous database design tips.

Normalization and Denormalization

All database designers strive for *normalization*, a database design technique that attempts to eliminate repeated information as much as possible. A database's *normal form* represents the degree to which the database has been normalized. There are several normal forms, but third normal form is generally considered sufficient for production databases.

However, normalization can be taken too far. In a previous chapter, I mentioned a customer of mine that had normalized their database so much that a simple street address lookup required joining more than nine tables. Here's a hint that something's wrong: If you're joining more than four tables, you've probably overnormalized and you should try to back off a bit. Table 8.1 illustrates the database design of my customer, who is in the real estate industry.

Table Name	Purpose	Example
PreDirection	The direction that precedes the street name	N, S, E, W, and so forth.
StreetType	The type of street	Rd, St, Ave, Lp, Ct, and so on.
PostDirection	The direction that follows the street name	NW, S, E, and so on.
UnitName	If there's a unit, the name of it	Apt, Unit, and so on.

Table 8.1: An example design database.

Table 8.1 doesn't shot the exact lookup tables that the customer used, of course, but you get the idea. A single piece of property would be represented in a giant resolver table that has five columns, as Table 8.2 shows.

Column	Purpose	Example
StreetNumber	The street number of the property	5527
PreDirection	The ID number of a row in the PreDirection table	5
StreetName	The name of the street	Main, Maple, 1st
StreetType	The ID number of a row in the StreetType table	12
PostDirection	The ID number of a row in the PostDirection table	35
UnitName	The ID number of a row in the UnitName table	2
UnitNumber	The number of the unit	A, Back, 2nd

Table 8.2: The columns that represent each property.

In my simplified example, looking up a single street address requires you to join five tables: The resolver table and all four lookup tables. That's a lot of work for a simple street address, especially because not every address will use every lookup table. The address 5527 N Main St, for example, won't use the PostDirection or UnitName tables.

There's nothing wrong with using lookup tables to store a list of acceptable values. When the list is small, however, there's no reason to create a foreign key constraint as in this example. After all, the StreetType table, for example, might have a couple of dozen entries. Instead, you can denormalize most of that information into the resolver table. The layout of that table doesn't change, just its contents, as Table 8.3 shows.

| Column	Purpose	Example
StreetNumber	The street number of the property	5527
PreDirection	The ID number of a row in the PreDirection table	N
StreetName	The name of the street	Main
StreetType	The ID number of a row in the StreetType table	St
PostDirection	The ID number of a row in the PostDirection table	\<Null\>
UnitName	The ID number of a row in the UnitName table	\<Null\>
UnitNumber	The number of the unit	\<Null\>

Table 8.3: The table with new contents.

The application user interface would still use the lookup tables to populate drop-down list boxes, forcing users to select from one of the acceptable values for items such as StreetType. Once selected, however, that information is copied into the main table rather than simply a reference of the information using an ID number. Now, addresses can be looked up using a single table, which is an incredible performance savings. Client application languages such as VB.NET let you fairly simply re-select the appropriate drop-down list values when a user edits an address, so there's no need to maintain complex foreign key constraints to the lookup tables and no need to join all five tables to look up a street address. Yet, the business requirement—ensuring that users only enter acceptable values in certain columns—is still fulfilled. Sure, there's duplicate data because multiple records will contain the value "St," but the redundant data is small, and so the impact on the database size will be absolutely minimal. When you're designing your own tables or looking at a redesign to improve performance, consider selective denormalization as a way to reduce query complexity.

💣 This is not carte blanche permission to denormalize! I generally recommend that you normalize your databases, then test their performance. Denormalize only when performance warrants doing so. Try to save denormalization for situations similar to my street address example. In that example, denormalizing small pieces of data has much less impact than if you were denormalizing customer names or some other large, critical piece of data.

Physical Design

By default, SQL Server databases start out with a single database file. Don't forget that you can add multiple database files and split database objects between files. To add files, simply modify a database's properties, as Figure 8.1 shows.

VERITAS™
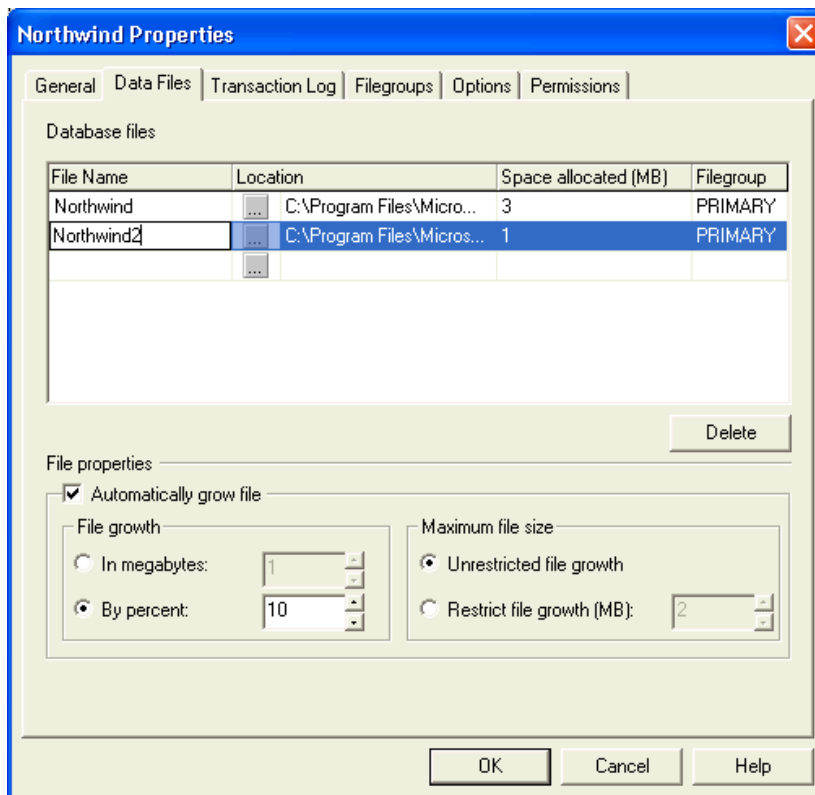
Figure 8.1: Adding a second database file to the Northwind database.

Notice that is Figure 8.1 the secondary file is being created as part of the same filegroup, PRIMARY. Thus, the secondary file will be used only when the other file in PRIMARY fills up. You can, of course, create your own filegroups, and therein lies the key to performance.

Using multiple database filegroups allows you to control where your database objects physically reside, which can be a useful performance trick. For example, suppose you've got a database that includes a heavily queried table, along with two associated indexes. If the SQL Server computer contains multiple hard drives, you can use multiple database files and filegroups to place the table and its indexes on different drives, speeding throughput. Simply create secondary database files (in new filegroups) on the other hard drives, then recreate the indexes so that they're stored on those filegroups:

```
CREATE INDEX MyIndex ON MyTable(MyColumn ASC)
ON SecondaryFileGroup
```

Now, SQL Server will be able to pull data from the table and the index at the same time without creating contention for the same hard drive. Of course, if you're using some types of RAID arrays, such as RAID 5, to store your data, then you're already avoiding drive contention. Instead of moving data to different physical drives, you might want to move the data and database objects to entirely different arrays to improve performance by reducing drive contention within the array itself.

VERITAS™

Miscellaneous Database Design Tips

As I said earlier, there are some SQL Server performance tuning tips and tricks that don't fit in a "story." The following tips are miscellaneous tips that can help you squeeze a little extra performance from SQL Server:

- The more space your database takes up, the more time SQL Server will wait while the OS reads from and writes to the hard drive. Try to minimize database size. One thing to pay attention to is columns with a large number of NULL values. If you have a fixed-size column, such as a 50-character column, storing a NULL value still requires 50 characters' worth of disk space. Consider using varchar columns instead when you anticipate a large number of NULLs, because nulling a varchar column requires only a couple of bytes.

- SQL Server 2000 stores data in 8KB pages, a concept I discussed in Chapter 1. Try to design your database tables so that as many rows as possible fit into 8KB (technically, 8060 bytes). SQL Server reads and writes pages as a unit, so if you can fit two rows into 8KB, SQL Server will read two rows of data. If you only fit one row into 8KB, SQL Server will only read one row for the same amount of work.

- Avoid using Windows' built-in RAID capabilities on the hard drives that store your database files. RAID is a great thing for data protection, but Windows' software RAID is slow. If you need RAID (and who doesn't?), invest in a SCSI or IDE RAID controller card. The hardware-based card will perform RAID without impacting the OS' throughput. For maximum throughput, of course, rely on fast SCSI hard drives rather than their cheaper, slower IDE cousins.

- Don't rely on auto-growing databases or transaction logs. SQL Server incurs a minor performance hit every time it has to expand a database. Leave the auto-growth feature turned on for emergencies, but monitor your databases carefully and resize them on your own terms, when you can plan for the performance hit.

- If you can, consider defragmenting your SQL Server computer's hard drives every so often. Doing so is especially important if you've resized your database files a number of times. To use Windows' built-in defragmenter or a third-party defragmenter, stop the SQL Server services. Doing so ensures that no database files are open and lets the defragmenter do its job. The defragmenter should get your database files into a single, contiguous area of disk space, which will provide maximum performance. Of course, if you're using RAID-5, which spreads your files across physical disks, defragmenters won't likely help much.

Application Design Tips

Application design—how your applications use SQL Server—can have an incredible impact on performance. Some application design problems can be mitigated through especially clever use of indexes and other SQL Server performance techniques. Other application design problems will continue to be a problem until you redesign the application.

Minimize Data and Play by the Rules

Client applications should be designed to query just the data they need from SQL Server. Having a client application query 100 rows when it only needs two is a waste of time and server resources. However, don't create client applications that query one row at a time as the user pages through data. Querying one row at a time results in unnecessary SQL Server connections and queries that can impede performance.

Ideally, have your client applications query an entire page of data, or a set of pages, at once. For example, if users frequently need to look at product information, and each product represents one screen of data in the client application, design the application to query a dozen products or so at once. As users move beyond the queried set of products, the application can query a few more products. This technique helps strike a fine line between querying too much data at once and not querying enough.

On the same note, make sure that client applications are behaving as good database citizens. Applications should be designed to minimize database locks and to keep transactions as short as possible to help minimize locks. Applications should never assume that things will work perfectly, and they should be prepared to handle a failed query, deadlock, or other situation gracefully.

Remember that SQL Server is, in effect, a very stupid robot that happens to be very good at executing queries. It can't tell if a query is especially good, if it's poorly written, or if the query will cause a major server problem. SQL Server will dutifully accept and execute every query it's given, so it's up to the application's designer to make sure that those queries will run properly and as quickly as possible.

☞ Avoid giving users ad-hoc query capabilities. The best way to bring a SQL Server computer to its knees, performance-wise, is to allow users to write their own queries. These queries will nearly always be the worst-running ones on your system because they're not optimized, aren't backed by specific indexes, and aren't running from a stored procedure. Instead, provide your users with some means of requesting new reports or queries, and allow trained developers and DBAs to work together to implement those queries in the most efficient way possible.

Avoid Triggers and Use Stored Procedures

Triggers are neat database objects that can serve a useful purpose. Triggers can be used to intercept data and make sure it's clean, cascade referential integrity changes throughout a hierarchy of table relationships, and so forth. That said, I'm not a big fan of triggers from a performance point of view.

SQL Server doesn't seem to execute triggers especially efficiently. For example, if you write a trigger that deals with deleted rows, then delete a thousand rows, the trigger takes a long time to execute and plow its way through the thousand deletions.

I realize that triggers represent a way to centralize business logic in the data tier, but I'm not convinced that they're the best way to do so. After all, the theory is that triggers are in place to automatically react to any database change, even ones made by ad-hoc queries. I think ad-hoc queries are a bad thing, too, and if you eliminate them from your environment, then you don't need triggers.

I'm a firm believer that all database changes should be made by way of stored procedures. They're more efficient, they centralize critical operations into the application's data tier, and SQL Server retains their execution plan for future use. If you use stored procedures as your "single point of entry" into SQL Server, there's no need for triggers—you can just put whatever trigger code you might need right into the original stored procedure.

SQL Server makes it quite easy to set up stored procedures as a single point of entry. Simply make sure that the same database user (typically, dbo) owns the stored procedure objects and all objects (such as tables) the stored procedures reference. Then grant users permission to execute the stored procedures, and *remove* permission to actually work with the underlying tables and other objects. Any user attempting to submit an ad-hoc query will be denied, forcing the user to employ the stored procedures and enforcing any business logic you've programmed into those stored procedures.

✎ Better stored procedures are coming soon! In SQL Server 7.0 and SQL Server 2000 (and in every other version, for that matter), stored procedures can be written only in SQL Server's native T-SQL language. Supposedly, the next version of SQL Server, currently code-named Yukon and slated for a 2004 release, will embed the .NET Framework's Common Language Runtime (CLR) within the SQL Server engine. The practical upshot of this inclusion is that you'll be able to write stored procedures in any .NET language, such as VB.NET or C#.

With the additional power and flexibility offered by .NET, there will be even less reason not to use stored procedures for absolutely every query that SQL Server executes. Stored procedures will become an even better single point of entry into SQL Server, eliminating the need for triggers.

Use a Middle Tier

Large applications need more than just a client application that executes stored procedures. That sort of two-tier design isn't very scalable because it places a lot of workload on the least-scalable component of the application—SQL Server itself.

Using a middle tier in your application design allows you to take some of the load off of SQL Server. Clients no longer connect to SQL Server itself; they request data through middle-tier components, and those components access SQL Server directly. The components should, ideally, run on their own separate tier of servers. In effect, the middle-tier components become clients of SQL Server, while your regular clients become clients of the middle tier. It's all laid out in Figure 8.2.

VERITAS™
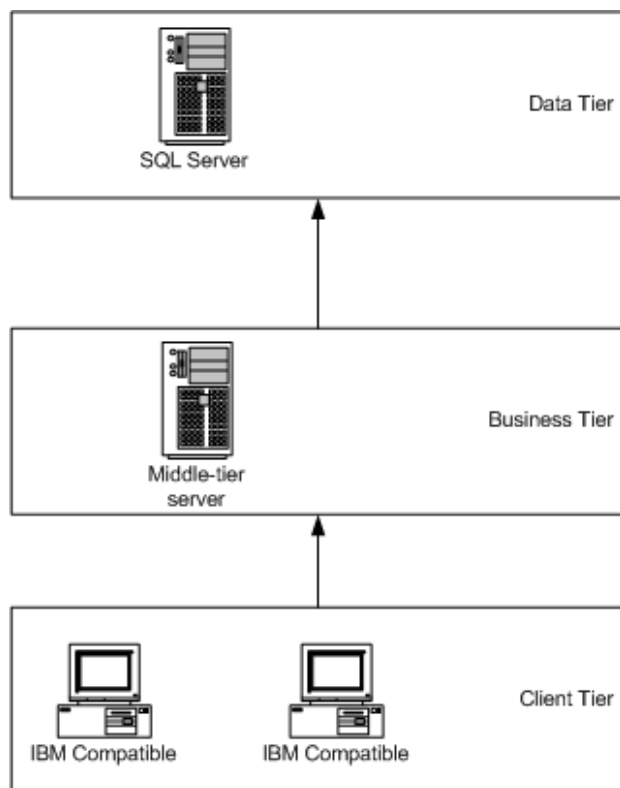
Figure 8.2: A typical three-tier application architecture.

The middle tier is often referred to as the *business tier* because the middle tier is where you typically implement business logic. Rather than writing stored procedures to validate data or maintain referential integrity, you write that code into the components running on the business tier. Your business logic can thus be fairly centralized but won't be a burden to SQL Server any longer.

The business tier should still access SQL Server exclusively through stored procedures, but these stored procedures can now be simplified because they don't need to incorporate business logic. They'll execute more quickly, allowing SQL Server to support more clients. As your middle-tier servers become overloaded, you simply add more, meeting the client demand.

☞ Working with XML? Consider a business tier. SQL Server 2000 and its various Web-based feature releases support several native XML features that can be very useful. For example, if you're receiving XML-formatted data from a business partner, you can have SQL Server translate, or *shred,* the XML into a relational data format and use it to perform table updates.

Unfortunately, doing so is about the least efficient thing you can do with SQL Server, at least in the current version. If you will be working with a great deal of XML-based data, I highly recommend that you build a middle tier to do so. The middle tier can do all the hard work of shredding XML data, and can submit normal T-SQL queries to SQL Server. Execution will be more efficient and you'll have a more scalable middle tier to handle any growth in data traffic.

VERITAS™

Use Microsoft Message Queuing Services for Long-Running Queries

Microsoft Message Queuing (MSMQ) services can be a great way to handle long-running queries. Rather than allowing clients (or even business-tier components) to submit long-running queries directly to SQL Server, clients submit their queries to a queue. A component running on SQL Server pulls these requests one at a time and executes them. When a query finishes, SQL Server places the results on a queue for the requestor. The requestor can check back periodically (or receive notification from MSMQ) and retrieve the results.

This technique allows you to strictly control the time and quantity of long-running queries, and allows requestors to continue working on other projects while they wait for their query results to become available. The requestor doesn't even have to be available when the query completes; MSMQ will store the results until the requestor is ready to retrieve them.

SQL Server's Data Transformation Services (DTS) even includes an MSMQ task that allows DTS to place query results and other information onto an MSMQ message queue. MSMQ is also accessible from COM-based languages, such as Visual Basic, and to .NET Framework applications.

Plan for Archival

When you initially design your application, plan for some way to move old data out of the application. Typically, databases contain date-sensitive data. So your database might need to contain 3 years' worth of data, after which you can move the oldest year into some archive. Your users might still have occasion to use that archived data, but not on a daily basis.

A good strategy is to horizontally partition your tables. Basically, you'll have a second copy of your database (perhaps on a different server) that contains archived data. Client applications will need to be written to query against this old data, if necessary, and you'll need to write stored procedures or DTS packages to move data into the archive every year.

> ☞ Use distributed views! You can create distributed views that combine the current and archived databases into a single set of virtual tables. Client applications can be written to query the current tables most of the time and to query the views when users need to access archived data. It's the easiest way to implement a distributed-archive architecture.

Why bother with archiving? The answer is that you archive primarily for performance reasons. The larger your databases, the larger your indexes, and the less efficient your queries will run. If you can minimize database size while still meeting your business needs, you'll give SQL Server a better chance at maintaining a high level of performance over the long haul.

Make sure that your users are aware of the consequences of querying archived data. For example, your client application might pop up a message that warns "Querying archived data will result in a longer-running query. Your results might take several minutes to retrieve. Are you sure you want to continue?" If you're running queries synchronously, make sure that you give users a way to cancel their query if it takes longer than they're willing to wait. For queries that you know will take several minutes to complete, consider running the queries asynchronously, perhaps using the MSMQ method I described earlier.

☞ Don't forget to update statistics! After you've made a major change, such as removing archived data, be sure to update the statistics on your tables (or ensure that the automatic update database option is enabled) so that SQL Server's query optimizer realizes your database has shrunk.

Indexing Tips

More often than not, indexes are the key to database performance. Thus, you should expect to spend a lot of time fussing with indexes to get them just right.

Fillfactor

When you create a new index, you specify a fillfactor. You do the same when you rebuild an index. Like everything else, SQL Server stores indexes in 8KB pages; the fillfactor specifies how full each 8KB page is when the index is created or rebuilt.

A low fillfactor leaves plenty of room for growth but means that SQL Server has to read more pages in order to access the index. A high fillfactor allows SQL Server to read the invoice as quickly as possible, but as rows are added, the index will be more likely to encounter split pages as the old pages fill up. Split pages are inefficient for SQL Server to read and will require you to rebuild the indexes more frequently.

The bottom line when determining a fillfactor is to understand how your data will grow. Specify the highest fillfactor that you can while still allowing enough room for the database to grow between indexes rebuilds.

Smart Indexing

Indexing, of course, is the best and worst things you can do to your tables. Many new DBAs throw indexes on every column in a table, hoping that one or two will be useful. Don't make that mistake! Indexes can make querying a database faster, but they slow changes to the database. The more write-heavy a table is, the more careful you need to be when you add your indexes.

In Chapters 4 and 5, I showed you various ways that you can use SQL Server's Index Tuning Wizard to get the right indexes on your tables to handle your workload. Used in conjunction with SQL Profiler and a representative query workload, the Index Tuning Wizard is your best first weapon in the battle to properly index your tables.

Indexing isn't a one-and-done deal, though. As your database grows, you'll need to reevaluate your indexing strategy. Indexes will need to be periodically rebuilt to ensure best performance. Changes to client applications, database design, or even your server's hardware will change your indexing strategy.

Learn to practice what I call *smart indexing.* Constantly review your indexes for appropriateness. Experiment, when possible, with different index configurations. One way to safely experiment with indexes is to create a testing server that's as close as possible in its configuration to your production server. Use SQL Profiler to capture a day's worth of traffic from your production server, then replay that traffic against your test server. You can change index configurations and replay the day's workload as often as necessary, monitoring performance all the while. When you find the index configuration that works best, you can implement it on your production server and check its performance for improvements.

Always Have a Clustered Index

Remember that a clustered index controls the physical order of the rows in a table, meaning you can have only one clustered index. That said, make sure that you *always* have a clustered index. There's rarely a reason not to have a clustered index. In fact, if you create a nonclustered index and don't already have a clustered index, SQL Server has to create a "phantom" clustered index anyway because nonclustered indexes always point to clustered index keys. You might as well create your own clustered index, ensuring that it will be of some use to the queries that you run against the table. If nothing else, create a clustered index on the table's identity column or some other unique column.

The best column for a clustered index is a column with unique values that is used in a number of different queries. For most queries, especially those that return multiple rows, a clustered index is faster than a nonclustered index. Because you only get one clustered index, try to use it where it will have the best impact on performance.

Don't forget that SQL Server 2000 offers a great GUI, in the form of Enterprise Manager, for managing indexes. You don't have to deal with complex stored procedures and T-SQL commands to get the right combination of indexes built on your tables. Simply open a table in design mode, right-click the workspace, and select Indexes from the pop-up menu. You'll see the dialog box that Figure 8.3 shows, which lets you manage your indexes, including creating a clustered index as shown.
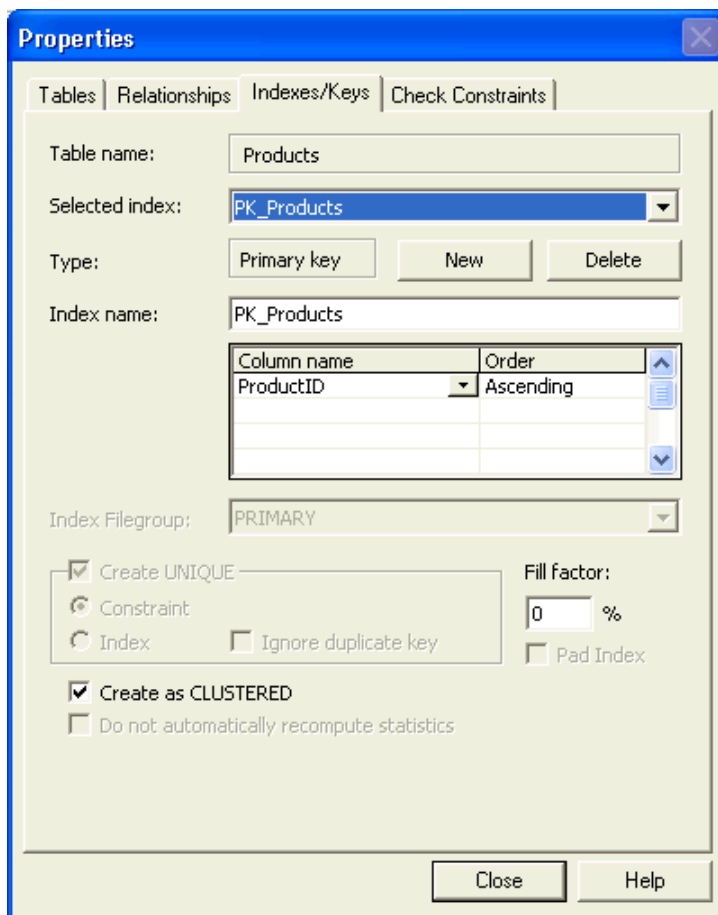

Figure 8.3: Use the check box at the bottom of the dialog box to create a clustered index on a table.

Composite Indexes

SQL Server indexes work best when they have a high degree of uniqueness. You can, of course, create indexes that group several columns together. For example, neither a first name nor last name column will usually be very unique in a customer table. However, the combination of first name and last name will be much more unique. These indexes are referred to as *composite* indexes.

You create composite indexes just like any other index. As Figure 8.4 illustrates, you can use Enterprise Manager to create them. Simply use the drop-down list boxes to specify the columns that will belong to the index. Always specify the most unique column first, when possible, because doing so will help SQL Server locate matching rows more quickly during a query.
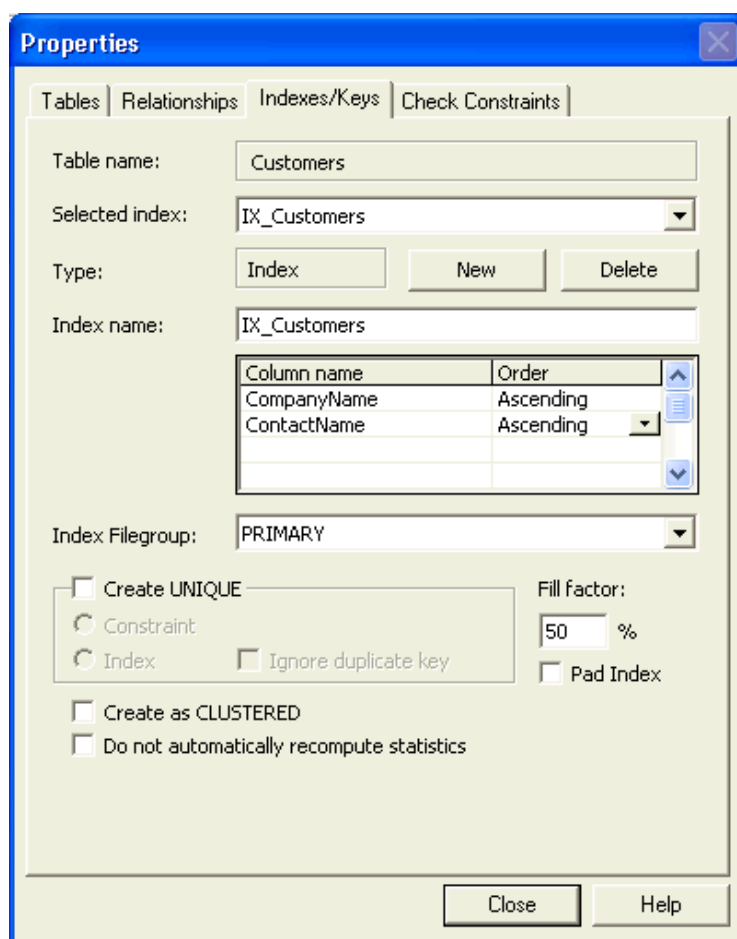
Figure 8.4: Creating composite indexes in SQL Server Enterprise Manager.

There are some unique best practices for composite indexes that you should keep in mind (in addition, you should be aware of a SQL Server composite index bug; see the sidebar "The Composite Index Bug" for more information):

- Keep indexes as narrow as possible. In other words, use the absolute minimum number of columns that you have to in order to get the effect you want. The larger the composite index, the harder SQL Server will have to work to keep it updated and to use it in queries.

VERITAS™

- The first column you specify should be as unique as possible, and ideally should be the one used by most queries' WHERE clauses. For example, if you frequently issue queries such as SELECT * FROM Customers WHERE ContactName='Don Jones', the composite index that Figure 8.4 shows wouldn't be very useful because ContactName is the second column in the index.

- Composite indexes that are also covering indexes are always useful. For example, using the index that Figure 8.4 shows, the query SELECT ContactName FROM Customers WHERE ContactName='Don Jones' would use the index, because the index contains 100 percent of the information required by the query—the ContactName column.

- Avoid using composite indexes as a table's clustered index. Clustered indexes don't do as well when they're based on multiple columns. Clustered indexes physically order the table's data rows and work best when they're based on a single column. If you don't have a single useful column, consider creating an identity column and using that as the basis for the clustered index.

The Composite Index Bug

Sadly, SQL Server isn't a perfect product, and bugs do exist. One fairly well-known bug relates to how the query optimizer uses composite indexes in large queries. The bug exists in SQL Server 7.0 and SQL Server 2000.

Here's what happens: When you issue a query that includes a WHERE clause with multiple OR operators, and some of the WHERE clauses rely on a composite index, the query optimizer might just give up and do a table scan instead of using the index. The bug occurs only when the query is executed from an ODBC application or from a stored procedure.

Microsoft has documented the bug and provides suggested workarounds in the article "BUG: Optimizer Uses Scan with Multiple OR Clauses on Composite Index" at
http://support.microsoft.com/default.aspx?scid=KB;en-us;q223423. Workarounds include using index hints to take the choice away from the optimizer and force it to use the index.

How can you tell if this bug is affecting you? You'll need to pay close attention to your production query execution plans, which you can view in SQL Profiler.

T-SQL and Query Tips

What hidden secrets lie within the T-SQL language that can help you improve performance? Quite a few, actually. As you're writing your queries, keep these tips in mind and you'll get better performance.

Always Use a WHERE Clause

The advice to always use a WHER clause might seem pretty basic, but you'd be surprised at the number of queries that folks write that return all rows. They'll then use some client-side trick to filter for the data they wanted, after making poor SQL Server work so hard to return all of the rows to begin with.

In addition, always be as specific in your WHERE clauses as possible so that SQL Server is returning as little data as possible. And, remember, if you're not using a WHERE clause, then SQL Server is just going to perform a table scan. Indexes are most useful when you've got a WHERE clause to limit rows.

While I'm on the subject of WHERE clauses, try to avoid WHERE clauses that aren't *sargeable*. Sargeable is DBA slang for queries that contain a constant value. The term itself comes from SARG, which stands for *search argument.* Hey, nobody ever said DBAs spent a lot of time making up cool words.

Basically, the deal is this: A WHERE clause such as WHERE CustomerID < 10 is sargeable because it contains a constant value to compare to. The query optimizer can take that WHERE clause and probably apply an index to make the query perform as quickly as possible. Non-constants in a WHERE clause include

- The <>, !=, !>, !< operators

- The IS NULL comparison

- The NOT, NOT EXISTS, NOT IN, NOT LIKE, and LIKE operators. Actually, the LIKE operator is sargeable when you're not using a wildcard as the first character (for example,

  ```
  LIKE 'A%'
  ```

  is sargeable, but

  ```
  LIKE '%A'
  ```

  isn't.

- Any function that includes a column, such as SUM(OrderTotal)

- Expressions with the same column on both sides of the operator, such as CustomerTotal = CustomerTotal + 5

These won't always prevent the optimizer from using an index, but there are better odds that they will. However, the optimizer is smart enough to use an index, where possible, for any sargeable expressions and to at least look at non-sargeable expressions to see if it can do anything with them.

If you've been careful to specify the columns you want in your SELECT statement, and those columns exist in a covering index, the optimizer will use the covering index rather than performing a slower table scan. Remember, a covering index is one that contains all of the columns specified in the SELECT statement.

Consider rewriting queries that aren't sargeable into ones that are. For example:

```
WHERE SUBSTRING(CustomerName,1,2) = 'do'
```

will do a fine job of returning all customers whose names begin with "do." It'll just take a table scan to do it because functions aren't sargeable. You could rewrite this query to

```
WHERE CustomerName LIKE 'do%'
```

That's a sargeable query, allowing the query optimizer to use an index on CustomerName (if one exists) to instantly narrow the query to everything that starts with "do."

Avoid Cursors!

Cursors are just a horrible thing, from a performance perspective. Granted, they can sometimes seem like the only way to complete a task, but that's rarely true. Consider the code sample that Listing 8.1 shows, which is adapted from a sample on http://www.sql-server-performance.com.

```
DECLARE @LineTotal money
DECLARE @InvoiceTotal money
SET @LineTotal = 0
SET @InvoiceTotal = 0

DECLARE Line_Item_Cursor CURSOR FOR
SELECT UnitPrice*Quantity
FROM [order details]
WHERE orderid = 10248

OPEN Line_Item_Cursor
FETCH NEXT FROM Line_Item_Cursor INTO @LineTotal
WHILE @@FETCH_STATUS = 0

BEGIN
SET @InvoiceTotal = @InvoiceTotal + @LineTotal
FETCH NEXT FROM Line_Item_Cursor INTO @LineTotal
END

CLOSE Line_Item_Cursor
DEALLOCATE Line_Item_Cursor
SELECT @InvoiceTotal InvoiceTotal
```

Listing 8.1: Sample code that uses a cursor.

This code simply locates an invoice (10248), adds up all the items on that invoice, and presents a total for the invoice. The cursor is used to step through each line item on the invoice and add its price into the @LineTotal variable. Listing 8.2 shows an easier way that doesn't involve a cursor.

```
DECLARE @InvoiceTotal money
SELECT @InvoiceTotal = sum(UnitPrice*Quantity)
FROM [order details]
WHERE orderid = 10248
SELECT @InvoiceTotal InvoiceTotal
```

Listing 8.2: The sample code modified so that it doesn't involve a cursor.

The new code uses SQL Server's aggregate functions to sum up all of the same information in fewer lines of code and without using a resource-hogging cursor. These aggregate functions can be a big timesaver and return the same results as more complex cursor operations.

VERITAS™

Miscellaneous T-SQL Tips

Again, there are a few miscellaneous tips that don't fit under one category heading. The following tips are some additional items to do and to avoid when using T-SQL:

- The DISTINCT clause is used to return a rowset that contains no duplicates. Before using the clause, however, ask yourself whether duplicates would be such a bad thing. The DISTINCT clause takes a considerable amount of extra time to run, in SQL Server terms, and if you can live without it you'll increase performance.

- I've mentioned before the need to limit the rows you query to just the ones you actually need; do the same with columns. Never type SELECT *, even if you want all columns returned. Instead, specifically list each column that you want, and only specify the columns you really need. You'll reduce the amount of data that SQL Server has to transmit. Additionally, you'll give SQL Server a better chance to use covering indexes if you limit the information you're querying.

- One good way to limit the amount of rows a query returns is to use the TOP keyword along with SELECT. TOP allows you to specify a maximum number of rows. SQL Server still has to execute any WHERE clause you specify, but will stop processing once it has found the specified number of matching rows, saving processing time. You can SELECT TOP 100 to return just a hundred rows or SELECT TOP 10 PERCENT to select a percentage of the actual result set. Alternatively, you can use the SET ROWCOUNT command to limit the number of rows returned for all queries executed by the current connection.

Hardware Tips

Hardware performance is obviously a big deal in the SQL Server world. I covered hardware in some detail in Chapter 6, in which I showed you how to scale out SQL Server to include larger servers and multiple servers. Here are some additional hardware-related performance tips:

- The more hard drives, the merrier. You can use them to house different filegroups or combine hard drives into read-efficient RAID arrays. Keep in mind that RAID arrays require additional write activity; implementing RAID arrays in a write-intensive database requires a high-end RAID card with lots of on-board memory to reduce hard drive latency.

- SQL Server will scale pretty well across any number of processors you can give it. Windows editions support between 4 and 32 processors, depending on the specific edition and the hardware on which the OS is running. For true Big Iron, look at Windows 2000 Datacenter (or Windows Server 2003 , Datacenter Edition) hardware.

- Use SCSI drives and Ultra160 or better SCSI when you can. SCSI provides the fastest throughput for most applications. If you're using a Storage Area Network (SAN) rather than dedicated drives, thoroughly test the SAN's throughput and compare it with a dedicated drive solution—don't trust the SAN manufacturer's performance numbers.

VERITAS™

- It's impossible to add too much RAM to a SQL Server computer, and with today's memory prices, there's no reason not to max out your server motherboards. Pay close attention to memory specifications, and purchase the highest-end memory your motherboard supports. Error correction and other features can improve stability and prevent downtime, which is the ultimate performance drag.

- Select CPUs with a large Level-2 (L2) cache. Communicating with main memory can introduce heavy latency in many servers, and the L2 cache helps reduce this latency by keeping much-needed data right on the processor, preventing a trip to main memory.

- When possible, look for servers that implement a look-aside, rather than look-through, memory cache architecture. With look-through caching, the server has to look at the cache for data. If the cache doesn't contain the data, the cache retrieves it from main memory, causing the server to wait longer while the data comes into the cache. With a look-aside architecture, the server sends memory requests to main memory and the cache at the same time. If the cache has the data requested, it'll respond quicker; if it doesn't, the request to main memory is already underway, so the processor won't have to wait as long for a result. Ask your server manufacturer which architecture they use.

- More processors or faster processors? Take a look at your average processor queue length in Performance Monitor. With high CPU usage and a low queue length (under 2), you might find that faster processors get you more benefit than more processors of the same speed. A high queue length, however, indicates a need for more processors. If buying more processors means buying a new server, opt for the fastest server-type processors you can get.

☞ 64-bit! Microsoft is already offering Windows 2000 Advanced Server in a 64-bit edition, and the upcoming Windows Server 2003 includes 64-bit versions of some editions. SQL Server will likely be the first of Microsoft's Windows Server 2003 Enterprise Servers to release a 64-bit edition, so keep that in mind for future purchases.

The 64-bit architecture, which will be available in competing versions from both Intel and AMD, will offer faster CPU speeds, better multiprocessor support, more memory support, and much faster peripheral busses (replacing the aged PCI busses we use now). The result will be servers that offer better performance than closely matched 32-bit machines, and you can be sure that SQL Server will take full advantage.

- Back on the subject of RAM: If you simply can't afford to fill your server's memory slots, then leave as many of them open for future expansion as possible by using the largest memory modules you can get for the RAM you do buy. As a general rule, try to get enough RAM to hold your database's largest table completely in RAM, if not the entire database itself. SQL Server Enterprise Edition running on Windows Datacenter can support as many as 64GB of RAM; if you (or your boss) feels that much RAM would be overkill, aim for enough RAM to contain your database.

- Multiprocessor servers configured to allow SQL Server to use parallel query execution need about 15 to 20 percent extra RAM to manage the more advance query execution methodology.

VERITAS™

- Never use Network-Attached Storage (NAS) for SQL Server databases. The file traffic must be converted to TCP/IP and pass over a network connection to the NAS. This setup isn't like a SAN, which generally uses directly attached fiber connections to carry the data. NAS is too slow and prone to TCP/IP hiccups that will dramatically (and negatively) affect server performance.

- Keep in mind that drive controllers have a maximum throughput, too. Plopping a dozen 50GB drives into a single RAID array might not be the most efficient way to use all of that space; consider buying additional controllers and creating several smaller arrays. You'll benefit from increased throughput because you'll have more controllers working for you.

Finally, use the latest drivers for all of your hardware. Manufacturers often correct performance problems in driver releases, especially in SCSI controller drivers and network adapter drivers. Take maximum advantage of your hardware's capabilities by using drivers that really push the envelope.

☞ Use signed drivers! Use only drivers that are digitally signed by Microsoft to ensure that the drivers have passed compatibility testing. Compatibility testing and only allowing signed drivers is a big reason why Datacenter gets 99.999 percent reliability numbers in tests. If you find that your vendor offers newer, unsigned drivers, ask why they aren't getting the newer drivers signed. If there's no good reason (and there usually isn't), consider switching to a more cooperative vendor.

Spread Out Your Databases

Another hardware-related performance trick is to run fewer databases on smaller servers, rather than heaping all of your company's databases onto one giant server. Two processors is not exactly twice the power of one processor; two one-processor servers running separate databases will nearly always outperform a single server running two of the same processors. Servers are complex devices with hundreds of potential bottlenecks. You'll get the best scalability when you spread databases across servers.

I discussed this type of scalability in Chapter 6. You can use federated databases and distributed views to spread a single logical database across several servers; you should always (as much as possible, at least) spread unrelated databases across different physical servers whenever possible.

Summary

In this chapter, I've tried to wrap up this book by bringing together a number of different performance topics and offering some general best practices. Use this chapter as a form of checklist for your environment, making sure that you're implementing as many of these practices as is practical for you. Good luck in all your SQL Server performance tuning efforts!