Wrox Programmer to Programmer™

Professional Microsoft® SQL Server® 2008 Programming

Robert Vieira

WROX

Updates, source code, and Wrox technical support at **www.wrox.com**

20

A Grand Performance: Designing a Database That Performs Well

This, and the chapter that follows, are probably the toughest chapters in the book from my perspective as the author, but not for the normal reasons. Usually, the issue is how to relate complex information in a manner that's easy to understand. As we're getting near the end of the book, I hope that I've succeeded there — even if there is still more to come. At this point, you should, from prior experience and the topics covered in this book, have a solid foundation in everything we're going to discuss in this chapter. That means I'm relatively free to get to the nitty-gritty and not worry quite as much about confusion.

Why then would this be a tough chapter for me to write? Well, because deciding exactly what to put into this and the sibling chapter that follows is difficult. You see, this isn't a book on performance tuning — that can easily be a book unto itself. It is, however, a book about making you successful in your experience developing with SQL Server. Having a well-performing system is critical to that success. The problem lies in a line from Bob Seger: ''What to leave in, what to leave out.'' What can we focus on here that's going to get you the most bang for your buck?

Perhaps the most important thing to understand about performance tuning is that you are never going to know everything there is to know about it. If you're the average SQL developer, you're going to be lucky if you know 20 percent of what there is to know. Fortunately, performance tuning is one of those areas where the old 80-20 rule (80 percent of the benefit comes from the right 20 percent of the work) definitely applies.

For this edition of the book, I've decided to expand this topic a bit, maintaining coverage of the structural decisions, and adding additional content on ''how to figure out where performance opportunities exist.'' This chapter will largely be on topics that have been around for a while including such things as:

- Index choices
- Client vs. server-side processing

- Strategic de-normalization
- Organizing your sprocs
- Uses for temporary tables
- Small gains in repetitive processes vs. big gains in long-running processes

The focus for this chapter is really going to be about things you should be thinking about in the area of design, those that are somewhat structural in nature. In many cases, it will be a subject we've already covered, but with a particular eye on performance. In our next chapter, we'll take a look at what to do once the system is already in place (maintenance, locating problems, and planning future changes).

There is, however, a common theme that one should get out of both chapters: This is only the beginning. The biggest thing in performance is really just to stop and think about it. There is, for some strange reason, a tendency when working with SQL to use the first thing that comes to mind that will work. You need to give the same kind of thought to your queries, sprocs, database designs — whatever — that you would give to any other development work that you're doing. Also, keep in mind that your T-SQL code is only one part of the picture — hardware, client code, SQL Server configuration, and network issues are examples of things that are ''outside the code'' that can have a dramatic impact on your system.

Performance means a lot of different things to a lot of different people. For example, many will think in terms of simple response time (how fast does my query finish). There is also the notion of *perceived* performance (many users will think in terms of how fast they receive enough to start working on, rather than how fast it actually finishes). Yet another perspective might focus on scalability (for example, how much load can I put on the system before my response time suffers or until users start colliding with each other?).

Many of the examples and suggestions in the two performance chapters are about raw speed — how fast do I return results — we do, however, touch on perceived performance and scalability issues where appropriate. Make sure that all facets of performance are considered in your designs — not just time to completion.

When to Tune

Okay, so this is probably going to seem a little obvious, but performance starts much earlier in the process than when you are writing your code. Indeed, it really should start in the requirements-gathering process and then never end.

What's the big deal about performance tuning in the requirements-gathering stage? Well, while you obviously can't do anything yet to *physically* tune your system, you can do a lot to *logically* tune your system. For example, is the concern of the customer more toward the side of *perceived* performance or actual completion of the job? For interactive processes, users will generally be more satisfied and *think*

the system is faster if you do something to show them that something is happening (even if it's just a progress bar). In addition, sometimes it's worth having a process that completes a little more slowly as long as the ''first response'' — that is, when it starts outputting something — is faster. Which of these is preferable is something you should know in the requirements-gathering stage. Finally, you should, in the requirements-gathering process, determine what your performance requirements are for the system.

> *Many is the time that I have seen the system that the developer thought was ''fast enough'' only to find out that the performance was unacceptable to the user. This can happen for a lot of reasons, though the most common is certainly the developer having his or her head buried in the sand.*
>
> *Find out what's expected! Also, remember to test whether you've met expectations under a realistic load on something resembling the real live hardware — not a load based on one or two developers sitting at their development system.*

Performance obviously also continues into design. If you design for performance, then you will generally greatly reduce the effort required to tune at completion. What's more, you'll find that you've greatly enhanced what are the ''best'' numbers you can achieve.

I'm starting to drone on here, but performance never stops — when you're actually coding, get it working, but then STOP! Stop and take a look at your code. Once an entire system is together, the actual code will almost never be looked at again unless:

- Something breaks (there's a bug).
- You need to upgrade that part of the system.
- There is an overt performance problem (usually, a *very* bad one).

In the first two of these instances, you probably won't be looking at the performance issues, just how to get things fixed or the additional functionality added. The point here is that an extra few minutes of looking at your code and asking yourself ''Could I have done it better?'' or ''Hey, have I done anything stupid here?'' can shave a little bit here and a little bit there and, occasionally, a whole lot in some other place.

> *Simply put: I make stupid mistakes, and so will you. It is, however, amazing how often you can step back from your code for a minute or two, then look at it again with a critical eye and say, ''Geez, I can't believe I did that!'' Hopefully, those moments will be rare, but, if you take the time to be critical of your own code, you'll find most of those critical gaffes that could really bog your system down. As for the ones you don't find, well, that's what the next chapter is for!*

The next big testing milestone time is in the quality assurance process. At this juncture you should be establishing general system benchmarks and comparing those against the performance requirements established during the requirements phase.

Last, but not least — never stop. Ask end users where their pain is from a performance perspective. Is there something they say is slow? Don't wait for them to tell you (often, they think ''that's just the way it is'' and say nothing — except to your boss, of course); go ask.

Index Choices

Again, this is something that was covered in extreme depth previously, but the topic still deserves something more than a mention here because of its sheer importance to query performance.

People tend to go to extremes with indexes — I'm encouraging you not to follow any one rule but to instead think about the full range of items that your index choices impact.

Any table that has a primary key (and with very rare exception, all tables should have a primary key) has at least one index. This doesn't mean, however, that it is a very useful index from a performance perspective. Indexes should be considered for any column that you're going to be frequently using as a target in a `WHERE` or `JOIN`, and, to a lesser extent, an `ORDER BY` clause.

Remember though, that the more indexes you have, the slower your inserts, updates, and deletes are going to be. When you modify a record, one or more entries may (depending on what's going on in the non-leaf levels of the B-Tree) have to be modified for that index (certainly true in the case of an insert or delete, and true for updates on any column participating in the index). That means more indexes and also more for SQL Server to do on modification statements. In an Online Transaction Processing (OLTP) environment (where you tend to have a lot of inserts, updates, and deletes), this can be a killer. In an Online Analytical Processing (OLAP) environment, this is probably no big deal since your OLAP data is usually relatively stable (few inserts), and what inserts are made are usually done through a highly repetitive batch process (doesn't have quite the lack of predictability that users have).

Technically speaking, the problem is smaller on updates and deletes. For updates, your indexes need to be updated only if the column that was changed is part of the key for that index. If you do indeed need to update the index though, think about it as a delete and an insert — that means that you're exposed to page splits again.

So, what, then, about deletes? Well, again, when you delete a record you're going to need to delete all the entries from your indexes too, so you do add some additional overhead, but you don't have to worry about page splits and having to physically move data around.

The bottom line here is that if you're doing a lot more querying than modifying, then more indexes are okay. However, if you're doing lots of modifications to your data, keep your indexes limited to high use columns.

If you're treating this book as more of a reference than a full ''learn how'' book and haven't taken the time to read the index chapters (Chapters 6 and 7) yet — do it!

Check the Index Tuning Tool in the Database Engine Tuning Advisor

The *Database Engine Tuning Advisor* is a descendant of the Index Tuning Wizard that made its first appearance back in version 7.0. While the Database Tuning Advisor has grown to include much more than just index tuning, it still has this key feature.

Be very careful when using automated tuning tools with indexes. In particular, watch out about what indexes you let it delete. It makes its recommendations based on the workload it has been exposed to — that workload may not include all of the queries that make up your system. Take a look at the recommendations and ask yourself why those recommendations might help. Particularly with deletions, ask yourself what that index might be used for — does deleting it make sense? Is there some long-running report that didn't run when you were capturing the workload file that might make use of that index?

Client vs. Server-Side Processing

Where you decide to ''do the work'' can have a very serious impact — for better or worse — on overall system performance.

When client/server computing first came along, the assumption was that you would get more/faster/cheaper by ''distributing'' the computing. For some tasks, this is true. For others though, you lose more than you gain.

Here's a quick review of some preferences and how they perform on client-side versus server side:

Static cursors	Usually much better on the client. Since the data isn't going to change, you want to package it up and send it all to the client in one pass — thus limiting roundtrips and network impact. The obvious exception is if the cursor is generated for the sole purpose of modifying other records. In such a case, you should try and do the entire process at the server-side (most likely in the form of a stored procedure) — again eliminating round-trips.
Forward-only, read-only cursors	Client-side again. ODBC and other libraries can take special advantage of the `FAST_FORWARD` cursor type to gain maximum performance. Just let the server spew the records into the client cursor, and then move on with life.
`HOLDLOCK` situations	Most transactioning works much better on the server than on the client.
Processes that require working tables	This is another of those situations where you want to try to have the finished product created before you attempt to move records to the client. If you keep all of the data server-side until it is really ready to be used, you minimize round-trips to the server and speed up performance.
Minimizing client installations	Okay, so this isn't ''performance'' as such, but it can be a significant cost factor. If you want to minimize the number of client installations you have to do, then keep as much of the business logic out of the client as possible. Either perform that logic in sprocs, or look at using component-based development with .NET. In an ideal world, you'll have what I like to call ''data logic'' (logic that exists only for the purpose of figuring out how to get the final data) in sprocs and ''business logic'' in components.

Continued

Significant filtering and/or resorting	Use ADO.NET or LINQ. They have a great set of tools for receiving the data from the server just once (fewer round-trips!), then applying filters and sorts locally. If you wanted the data filtered or sorted differently by SQL Server, it would run an entirely new query using the new criteria. It doesn't take a rocket scientist to figure out that the overhead on that can get rather expensive. Both ADO.NET and LINQ also have some cool things built-in to allow you to join different data sets (including homogeneous data sets) right at the client. Note, however, that with very large result sets, your client computer may not have the wherewithal to deal with the filters and sorts effectively — you may be forced to go back to the server.

These really just scratch the surface. The big thing to remember is that round-trips are a killer even in this age of gigabit Ethernet (keep in mind that connection overhead is often more of the issue than raw bandwidth). What you need to do is move the smallest amount of data back and forth — and only move it once. Usually, this means that you'll preprocess the data as much as possible on the server side, and then move the entire result to the client if possible.

Keep in mind, though, that you need to be sure that your client is going to be able to handle what you give it. Servers are usually much better equipped to handle the resource demands of larger queries. By the same token, you also have to remember that the server is going to be doing this for multiple users — that means the server needs to have adequate resources to store all of the server-side activity for that number of users. If you take a process that was too big for the client to handle and move it server-side for resource reasons, just remember that you may also run out of resources on the server, if more than one client uses that process at one time. The best thing is to try to keep result sets and processes in the smallest size possible.

Realize that the term ''client'' has more than one possible meaning. The client, from a data connection perspective, may not be where the end user sits. If it is a browser-based application, then the client that is truly handling the data is more likely the Web server. While a Web server is likely on some very solid hardware, it may be dealing with multiple such queries at the same time (multiple large data sets), so plan accordingly.

Strategic De-Normalization

This could also be called, ''When following the rules can kill you.'' Normalized data tends to work for both data integrity and performance in an OLTP environment. The problem is that not everything that goes on in an OLTP database is necessarily transaction-processing related. Even OLTP systems have to do a little bit of reporting (a summary of transactions entered that day, for example).

Often, adding just one extra column to a table can prevent a large join, or worse, a join involving several tables. I've seen situations where adding one column made the difference between a two-table join and a nine-table join. We're talking the difference between 100,000 records being involved and several million.

This one change made the difference in a query dropping from a runtime of several minutes down to just seconds.

Like most things, however, this isn't something with which you should get carried away. Normalization is the way that most things are implemented for a reason. It adds a lot to data integrity and can make a big positive difference performance-wise in many situations. Don't de-normalize just for the sake of it. Know exactly what you're trying to accomplish, and test to make sure that it had the expected impact. If it didn't, then look at going back to the original way of doing things.

Organizing Your Sprocs Well

I'm not talking from the outside (naming conventions and such are important, but that's not what I'm getting at here) but rather from a ''how they operate'' standpoint. The next few sections discuss this.

Keeping Transactions Short

Long transactions cannot only cause deadlock situations but also basic blocking (where someone else's process has to wait for yours because you haven't finished with the locks yet). Anytime you have a process that is blocked — even if it will eventually be able to continue after the blocking transaction is complete — you are delaying, and therefore hurting the performance of, that blocked procedure. There is nothing that has a more immediate effect on performance than that a process has to simply stop and wait.

Using the Least Restrictive Transaction Isolation Level Possible

The tighter you hold those locks, the more likely that you're going to wind up blocking another process. You need to be sure that you take the number of locks that you really need to ensure data integrity — but try not to take any more than that.

If you need more information on isolation levels, check out transactions and locks in Chapter 11.

Implementing Multiple Solutions if Necessary

An example here is a search query that accepts multiple parameters but doesn't require all of them. It's quite possible to write your sproc so that it just uses one query, regardless of how many parameters were actually supplied — a ''one-size-fits-all'' kind of approach. This can be a real timesaver from a development perspective, but it is really deadly from a performance point of view. More than likely, it means that you are joining several unnecessary tables for every run of the sproc!

The thing to do here is to add a few `IF...ELSE` statements to check things out. This is more of a ''look before you leap'' kind of approach. It means that you will have to write multiple queries to deal with each possible mix of supplied parameters, but once you have the first one written, the others can often be cloned and then altered from the first one.

This is a real problem area in lots of code out there. Developers are a fickle bunch. We generally only like doing things as long as they are interesting. If you take the preceding example, you can probably see that

it would get very boring very quickly to be writing what amounts to a very similar query over and over to deal with the nuances of what parameters were supplied.

All I can say about this is — well, not everything can be fun, or everyone would want to be a software developer! Sometimes you just have to grin and bear it for the sake of the finished product.

Avoiding Cursors if Possible

If you're a programmer who has come from an ISAM or VSAM environment (these were older database storage methods), doing things by cursor is probably going to be something toward which you'll naturally gravitate. After all, the cursor process works an awful lot more like what you're used to in those environments (such looping structures are also common in many non-database data handling constructs).

Don't go there!

Almost all things that are first thought of as something you can do by cursors can actually be done as a set operation. Sometimes it takes some pretty careful thought, but it usually can be done.

By way of illustration, I was asked several years ago for a way to take a multiline cursor-based operation and make it into a single statement if possible. The existing process ran something like 20 minutes. The runtime was definitely problematic, but the customer wasn't really looking to do this for performance reasons (they had accepted that the process was going to take that long). Instead, they were just trying to simplify the code.

They had a large product database, and they were trying to set things up to automatically price their available products based on cost. If the markup had been a flat percentage (say 10 percent), then the `UPDATE` statement would have been easy — say something like:

```
UPDATE Products
SET UnitPrice = UnitCost * 1.1
```

The problem was that it wasn't a straight markup — there was a logic pattern to it. The logic went something like this:

- If the pennies on the product after the markup are greater than or equal to .50, then price it at .95.
- If the pennies are below .50, then mark it at .49.

The pseudocode to do this by cursor would look something like:

```
Declare and open the cursor
Fetch the first record
Begin Loop Until the end of the result set
Multiply cost * 1.1
If result has cents of < .50
   Change cents to .49
Else
   Change cents to .95
Loop
```

This is, of course, an extremely simplified version of things. There would actually be about 30–40 lines of code to get this done. Instead, we changed it around to work with one single correlated subquery (which had a `CASE` statement embedded in it). The runtime dropped down to something like 12 seconds.

The point here, of course, is that, by eliminating cursors wherever reasonably possible, we can really give a boost to not only reduce complexity (as was the original goal here) but also performance.

Uses for Temporary Tables

The use of temporary tables can sometimes help performance — usually by allowing the elimination of cursors or by allowing working data to be indexed while it is needed.

Using Temp Tables to Break Apart Complex Problems

As we've seen before, cursors can be the very bane of our existence. Using temporary tables, we can sometimes eliminate the cursor by processing the operation as a series of two or more set operations. An initial query creates a working data set. Then another process comes along and operates on that working data.

We can actually make use of the pricing example we laid out in the last section to illustrate the temporary table concept, too. This solution wouldn't be quite as good as the correlated subquery, but it is still quite workable and much faster than the cursor option. The steps would look something like:

```
SELECT ProductID, FLOOR(UnitCost * 1.1) + .49 AS TempUnitPrice
     INTO #WorkingData
     FROM Products
     WHERE (UnitCost * 1.1) - FLOOR(UnitCost * 1.1) < .50
INSERT INTO #WorkingData
SELECT ProductID, FLOOR(UnitCost * 1.1) + .95 AS TempUnitPrice
     FROM Products
     WHERE (UnitCost * 1.1) - FLOOR(UnitCost * 1.1) >= .50
UPDATE p
     SET p.UnitPrice = t.TempUnitPrice
     FROM Product p
     JOIN #WorkingData t
       ON p.ProductID = t.ProductID
```

With this, we wind up with three steps instead of thirty or forty. This won't operate quite as fast as the correlated subquery would, but it still positively screams in comparison to the cursor solution.

Keep this little interim step using temporary tables in mind when you run into complex problems that you think are going to require cursors. Try to avoid the temptation of just automatically taking this route — look for the single statement query before choosing this option — but if all else fails, this can really save you a lot of time versus using a cursor option.

Using Temp Tables to Allow Indexing on Working Data

Often we will run into a process in which we are performing many different operations on what is fundamentally the same data. This is characterized by a situation in which you are running different

kinds of updates (perhaps to totally different tables), but utilizing the same source data to figure out what to change or what values to change things to. I've seen many scenarios where the same fundamental data is reused — in the same procedure — hundreds or even thousands of times.

Under such ''reuse'' situations, consider querying the data once and placing it into a temp table. Also consider applying indexes to this data as warranted by the queries you're going to be performing against it.

Even for data you're only going to be hitting twice, I've seen a temp table solution make a huge difference if the original query for the source data was, for whatever reason, inefficient. Sometimes this is due to a lack of suitable indexing on the source data, but, more often, it is a scenario with a multi-table join against a large data set. Sucking it into a temp table often allows you to explicitly filter down a large data set early in the overall process. Again, try and avoid the temptation of automatically taking this approach, but keep it in mind as an option.

Update Your Code In a Timely Fashion

Are you still supporting SQL Server 2000? How about 7.0? OK, so you most definitely shouldn't be supporting 7.0 by now, and even 2000 support should be gone (or at least in the late stages of sunsetting it). So, if you're no longer supporting those older editions, why does your system code and design look like you still are?

OK, OK, I understand it isn't as simple as all that, but with each release of your application, make sure that you have time set aside (I recommend 10%–25%) that is oriented around improving existing performance and features. If you only need to support SQL Server 2008, look for special code you may have to address situations now addressed natively by SQL Server 2008, such as:

- Procedures or code streams that handle `INSERT`, `UPDATE`, and `DELETE` scenarios into a specific table; these can use the new `MERGE` command to make all three modifications, as indicated, in a single pass over the data. It also has the advantage of being a single statement, which means you may be able to avoid explicitly defining transactions around the three separate statements.
- Special hierarchy handling: SQL Server now has native constructs for something that is actually very common. The functionality includes not only hierarchy-specific functions (such as pruning or grafting), but both vertical and horizontal index functionality (very cool stuff!).
- Date and Time data type handling.

Sometimes, It's the Little Things

A common mistake in all programming for performance efforts is to ignore the small things. Whenever you're trying to squeeze performance, the natural line of thinking is that you want to work on the long-running stuff.

It's true that the long-running processes are the ones for which you stand the biggest chance of getting big one-time performance gains. It's too bad that this often leads people to forget that it's the total time saved that they're interested in — that is, how much time when the process is really live.

While it's definitely true that a single change in a query can often turn a several-minute query into seconds (I've actually seen a few that took literally days trimmed to just seconds by index and query tuning), the biggest gains for your application often lie in getting just a little bit more out of what already seems like a fast query. These are usually tied to often-repeated functions or items that are often executed within a loop.

Think about this for a bit. Say you have a query that currently takes three seconds to run, and this query is used every time an order taker looks up a part for possible sale — say 5,000 items looked up a day. Now imagine that you are able to squeeze one second off the query time. That's 5,000 seconds, or over an hour and 20 minutes!

Hardware Considerations

Forgive me if I get too bland here — I'll try to keep it interesting, but if you're like the average developer, you'll probably already know enough about this to make it very boring, yet not enough about it to save yourself a degree of grief.

Hardware prices have been falling like a rock over the years — unfortunately, so has what your manager or customer is probably budgeting for your hardware purchases. When deciding on a budget for your hardware, remember:

- Once you've deployed, the hardware is what's keeping your data safe — just how much is that data worth?
- Once you've deployed, you're likely to have many users — if you're creating a public website, it's possible that you'll have tens of thousands of users active on your system 24 hours per day. What is it going to cost you in terms of productivity loss, lost sales, loss of face, and just general credibility loss if that server is unavailable or — worse — you lose some of your data?
- Maintaining your system will quickly cost more than the system itself. Dollars spent early on a mainstream system that is going to have fewer quirks may save you a ton of money in the long run.

There's a lot to think about when deciding from whom to purchase and what specific equipment to buy. Forgetting the budget for a moment, some of the questions to ask yourself include:

- Will the box be used exclusively as a database server?
- Will the activity on the system be processor or I/O intensive? (For databases, it's almost always the latter, but there are exceptions.)
- Am I going to be running more than one production database? If so, is the other database of a different type (OLTP versus OLAP)?
- Will the server be on-site at my location, or do I have to travel to do maintenance on it?
- What are my risks if the system goes down?
- What are my risks if I lose data?
- Is performance ''everything''?
- What kind of long-term driver support can I expect as my O/S and supporting systems are upgraded?

Again, we're just scratching the surface of things — but we've got a good start. Let's look at what these issues mean to us.

Exclusive Use of the Server

I suppose it doesn't take a rocket scientist to figure out that, in most cases, having your SQL Server hardware dedicated to just SQL Server and having other applications reside on totally separate system(s) is the best way to go. Note, however, that this isn't always the case.

If you're running a relatively small and simple application that works with other sub-systems (say IIS as a Web server, for example), then you may actually be better off, performance-wise, to stay with one box. Why? Well, if there are large amounts of data going back and forth between the two sub-systems (your database in SQL Server and your Web pages or whatever in a separate process), then memory space to memory space communications are going to be much faster than the bottleneck that the network can create — even in a relatively dedicated network backbone environment.

Remember that this is the exception, though, not the rule. The instance where this works best usually meets the following criteria:

- The systems have a very high level of interaction.
- The systems have little to do beyond their interactions (the activity that's causing all the interaction is the main thing that the systems do).
- Only one of the two processes is CPU intensive and only one is I/O intensive.

If in doubt, go with conventional thinking on this and separate the processing into two or more systems.

I/O vs. CPU Intensive

I can just hear a bunch of you out there yelling ''Both!'' If that's the case, then I hope you have a very large budget — but we'll talk about that scenario, too. Assuming you haven't installed yet, it's guesswork. While almost anything you do in SQL Server is data-based and will, therefore, certainly require a degree of I/O, how much of a burden your CPU is under varies widely depending on the types of queries you're running:

Low CPU Load	High CPU Load
Simple, single-table queries and updates	Large joins
Joined queries over relatively small tables	Aggregations (`SUM`, `AVG`, etc.) Sorting of large result sets

With this in mind, let's focus in a little closer on each situation.

I/O Intensive

I/O-intensive tasks should cause you to focus your budget more on the drive array than on the CPU(s). Notice that I said the drive ''array'' — I'm not laying that out as an option. In my not-so-humble opinion on this matter, if you don't have some sort of redundancy arrangement on your database storage mechanism, then you have certainly lost your mind. Any data worth saving at all is worth protecting — we'll talk about the options there in just a moment.

Before we get into talking about the options on I/O, let's look briefly into what I mean by I/O intensive. In short, I mean that a lot of data retrieval is going on, but the processes being run on the system are almost exclusively queries (not complex business processes), and those do not include updates that require wild calculations. Remember — your hard drives are, more than likely, the slowest thing in your system (short of a CD-ROM) in terms of moving data around.

A Brief Look at RAID

RAID; it brings images of barbarian tribes raining terror down on the masses. Actually, most of the RAID levels are there for creating something of a fail-safe mechanism against the attack of the barbarian called ''lost data.'' If you're not a RAID aficionado, then it might surprise you to learn that not all RAID levels provide protection against lost data.

RAID originally stood for *Redundant Array of Inexpensive Disks*. The notion was fairly simple — at the time, using a lot of little disks was cheaper than using one great big one. In addition, an array of disks meant that you had multiple drive heads at work and could also build in (if desired) redundancy.

Since drive prices have come down so much (I'd be guessing, but I'd bet that drive prices are, dollar per meg, far less than 1 percent of what they were when the term RAID was coined), I've heard other renditions of what RAID stands for. The most common are Random Array of Independent Disks (this one seems like a contradiction in terms to me) and Random Array of Individual Disks (this one's not that bad). The thing to remember, no matter what you think it's an acronym for, is that you have two or more drives working together — usually for the goal of some balance between performance and safety.

There are lots of places you can get information on RAID, but let's take a look at the three (well, four if you consider the one that combines two of the others) levels that are most commonly considered:

RAID Level	Description
RAID 0	a.k.a. Disk Striping without Parity. Out of the three that you are examining here, this is the one you are least likely to know. This requires at least three drives to work just as RAID 5 does. Unlike RAID 5, however, you get no safety net from lost data. (Parity is a special checksum value that allows reconstruction of lost data in some circumstances — as indicated by the time, RAID 0 doesn't have parity.) RAID 0's big claim to fame is giving you maximum performance without losing any drive space. With RAID zero, the data you store is spread across all the drives in the array (at least 3). While this may seem odd, it has the advantage of meaning that you always have three or more disk drives reading or writing your data for you at once. Under mirroring, the data is all on one drive (with a copy stored on a separate drive). This means you'll just have to wait for that one head to do the work for you.
RAID 1	a.k.a. Mirroring. For each active drive in the system, there is a second drive that ''mirrors'' (keeps an exact copy of) the information. The two drives are usually identical in size and type, and store all the information to each drive at the same time. (Windows NT has software-based RAID that can mirror any two volumes as long as they are the same size.) Mirroring provides no performance increase when writing data (you still have to write to both drives) but can, depending on your controller arrangement, double your read

Continued

RAID Level	Description
	performance since it will use both drives for the read. What's nice about mirroring is that as long as only one of the two mirrored drives fails, the other will go on running with no loss of data or performance (well, reads may be slower if you have a controller that does parallel reads). The biggest knock on mirroring is that you have to buy two drives to every one in order to have the disk space you need.
RAID 5	The most commonly used. Although, technically speaking, mirroring is a RAID (RAID 1), when people refer to using RAID, they usually mean RAID 5. RAID 5 works exactly as RAID 0 does with one very significant exception — parity information is kept for all the data in the array. Say, for example, that you have a five-drive array. For any given write, data is stored across all five of the drives, but a percentage of each drive (the sum of which adds up to the space of one drive) is set aside to store parity information. Contrary to popular belief, no one drive is the parity drive. Instead, some of the parity information is written to all the drives — it's just that the parity information for a given byte of data is not stored on the same drive as the actual data is. If any one drive is lost, then the parity information from the other drives can be used to reconstruct the data that was lost. The great thing about RAID 5 is that you get the multi-drive read performance. The downside is that you lose one drive's worth of space (if you have a three-drive array, you'll see the space of two; if it's a seven-drive array, you'll see the space of six). It's not as bad as mirroring in the price per megabyte category, but you still see great performance.
RAID 6	Raid can be considered to be something of an extension of RAID 5 and is generally only used in very large arrays (where the overhead of algorithm required to provide the extra redundancy can be spread out and therefore provides less waste on a per disk basis). RAID 6 provides extra parity encoding versus RAID 5, and the extra information can be utilized to recover from multiple drive loss. RAID 5 is generally less expensive at lower array sizes, but RAID 6 maintains a level of redundancy even while rebuilding a single failed drive.
RAID 10, (a.k.a. RAID 1 + 0) or RAID 0 + 1	RAID 10 offers the best of both RAID 0 and RAID 1 in terms of performance and data protection. It is, however, far and away the most expensive of the options discussed here. RAID 10 is implemented in a coupling of both RAID 1 (Mirroring) and RAID 0 (striping without parity). The end result is mirrored sets of striped data. You will also hear of RAID 0 + 1. These are striped sets of mirrored data. The end result in total drive count is the same, but RAID 10 performs better in recovery scenarios and is therefore what is typically implemented.
RAID 50	This is implemented by mirroring two RAID 5 arrays. While it is arguably the most redundant, it is still at risk of failure if two drives happen to fail in the same array. It is the most expensive of the options provided here, and generally only implemented in the most extreme of environments.

The long and the short of it is that RAID 5 is the de facto minimum for database installations. That being said, if you have a loose budget, then I'd actually suggest mixing things up a bit.

> *RAID 10 has become the standard in larger installations. For the average shop, however, RAID 5 will likely continue to rule the day for a while yet — perhaps that will change as we get into the era where even server level drives are measured in multi-tera-, peta-, and even exabytes. We certainly are getting there fast.*

What you'd like to have is at least a RAID 5 setup for your main databases but a completely separate mirrored set for your logs. People who manage to do both usually put both Windows and the logs on the mirror set and the physical databases on the RAID 5 array, but those with a little more cash to spend often put the O/S on a separate mirror set from the logs (with the data files still on their own RAID 5 array). Since I'm sure inquiring minds want to know why you would want to do this, let's make a brief digression into how log data is read and written.

Unlike database information, which can be read in parallel (thus why RAID 5 or 10 works so well performance-wise), the transaction log is chronology dependent — that is, it needs to be written and read serially to be certain of integrity. I'm not necessarily saying that physically ordering the data in a constant stream is required; rather, I'm saying that everything needs to be logically done in a stream. As such, it actually works quite well if you can get the logs into their own drive situation where the head of the drive will only seldom have to move from the stream from which it is currently reading and writing. The upshot of this is that you really want your logs to be in a different physical device than your data, so the reading and writing of data won't upset the reading and writing of the log.

> *Note that this sequential read/write performance of the mirror set disappears if you are keeping logs for multiple databases on the same mirror set (it has to jump around between the separate logs!).*

Logs, however, don't usually take up nearly as much space as the read data does. With mirroring, we can just buy two drives and have our redundancy. With RAID 5, we would have to buy three, but we don't see any real benefit from the parallel read nature of RAID 5. When you look at these facts together, it doesn't make much sense to go with RAID 5 for the logs or O/S.

> **You can have all the RAID arrays in the world, and they still wouldn't surpass a good backup in terms of long-term safety of your data. Backups are easy to take off-site, and are not subject to mechanical failure. RAID units, while redundant and very reliable, can also become worthless if two (instead of just one) drives fail. Another issue — what if there's a fire? Probably all the drives will burn up — again, without a backup, you're in serious trouble. We'll look into how to back up your databases in Chapter 22.**

CPU Intensive

On a SQL Server box, you'll almost always want to make sure that you go multiprocessor (yes, even in these days of multi-core processors), even for a relatively low-utilization machine. This goes a long way to preventing little ''pauses'' in the system that will drive your users positively nuts, so consider this part of things to be a given — particularly in this day of dual core processors. Keep in mind that the Workgroup version of SQL Server supports only up to two processors — if you need to go higher than

that, you'll need to go up to either Standard (four processors) or the Enterprise edition (which is limited only by your hardware and budget).

> *Even if you're only running SQL Server Express — which supports only one processor — you'll want to stick with the dual-proc box if at all possible. Remember, there is more going on in your system than SQL Server, so having that other proc available to perform external operations cuts down on lag on your SQL Server.*

Perhaps the biggest issue of all, though, is memory. This is definitely one area that you don't want to short change. In addition, remember that if you are in a multiprocessor environment (and you should be), then you are going to have more things going on at once in memory. In these days of cheap memory, no SQL Server worth installing should ever be configured with less than 512MB of RAM — even in a development environment. Production servers should be equipped with no less than 2GB of RAM — quite likely more.

Things to think about when deciding how much RAM to use include:

- How many user connections will there be at one time (each one takes up space)? Each connection takes up about 24K of memory (it used to be even higher). This isn't really a killer since 1,000 users would only take up 24MB, but it's still something to think about.
- Will you be doing a lot of aggregations and/or sorts? These can be killers depending on the size of the data set you're working with in your query.
- How large is your largest database? If you have only one database, and it is only 1GB (and, actually, most databases are much smaller than people think), then having 4GB of RAM probably doesn't make much sense depending on how many queries you're running simultaneously and exactly what actions they are taking.
- The Workgroup edition of SQL Server 2008 only supports addressing of memory up to 3GB. If you need more than this, you'll need to go with at least the Standard edition.

In addition, once you're in operation — or when you get a fully populated test system up and running — you may want to take a look at your cache-hit ratio in perfmon. We'll talk about how this number is calculated a little bit in Chapter 21. For now, it's sufficient to say that this can serve as something of a measurement for how often we are succeeding at getting things out of memory rather than off disk (memory is going to run much, much faster than disk). A low cache-hit ratio is usually a certain indication that more memory is needed. Keep in mind though, that a high ratio does not necessarily mean that you shouldn't add more memory. The read-ahead feature of SQL Server may create what is an artificially high cache-hit ratio and may disguise the need for additional memory.

OLTP vs. OLAP

The needs between these two systems are often at odds with each other. We discuss some of the design differences in Chapter 24, so I hope you will come to have a concept of just how different the design considerations can be.

In any case, I'm going to keep my ''from a hardware perspective'' recommendation short here:

> **If you are running databases to support both of these kinds of needs, run them on different servers — it's just that simple.**

I can't stress enough the need to separate these two. A large data warehouse import, export, or even a large report run can cause significant turnover in your OLTP procedure and/or data caches and simply decimate the performance of your system for what can be many users (and, therefore, a whole lot of cost).

On-Site vs. Off-Site

It used to be that anything that would be SQL Server–based would be running on-site with those who were responsible for its care and upkeep. If the system went down, people were right there to worry about reloads and to troubleshoot.

In the Internet era, many installations are co-located with an Internet service provider (ISP). The ISP is responsible for making sure that the entire system is backed up — they will even restore according to your directions — but they do not take responsibility for your code. This can be very problematic when you run into a catastrophic bug in your system. While you can always connect remotely to work on it, you're going to run into several configuration and performance issues, including:

- **Security** — Remote access being open to you means that you're also making it somewhat more open to others who you may not be interested in having access. My two bits' worth on this is to make sure that you have very tight routing and port restrictions in place. For those of you not all that network savvy (which includes me), this means that you restrict what IP addresses are allowed to be routed to the remote server, what ports they have available, and even what protocols (SSL vs. non-SSL) are allowed through.
- **Performance** — You're probably going to be used to the 100 Mbps to 1 Gbps network speeds that you have around the home office. Now you're communicating via virtual private network (VPN) over the Internet or, worse, dialup, and you are starting to hate life (things are SLOW!).
- **Responsiveness** — It's a bit upsetting when you're running some e-commerce site or whatever and you can't get someone at your ISP to answer the phone, or they say that they will get on it right away and hours later you're still down. Make sure you investigate your remote hosting company very closely — don't assume that they'll still think you're important after the sale.
- **Hardware maintenance** — Many co-hosting facilities will not do hardware work for you. If you have a failure that requires more than a reloading, you may have to travel to the site yourself or call yet another party to do the maintenance — that means that your application will be offline for hours or possibly days.

If you're a small shop doing this with an Internet site, then off-site can actually be something of a saving grace. It's expensive, but you'll usually get lots of bandwidth plus someone to make sure that

the backups actually get done — just make sure that you really check out your ISP. Many of them don't know anything about SQL Server, so make sure that expertise is there.

> *One recent trend in major ISPs has been to locate major hosting facilities in far more remote locations than you might, at first, expect. This is usually done for accessibility to water (for cooling), cheap power, or both (near hydroelectric facilities seems to be popular). In many ways, this shouldn't matter, but think about it if you're using a third-party hardware support company — does that support company have appropriate staff located near the facility where you will be hosted?*
>
> *If you were thinking of your hosting company as being located in a major metropolitan area, then you would reasonably assume that your hosting company had a large number of support staff within 30–60 minutes' response time of your ISP location. If, however, your ISP is, let's say, ''outside Portland, Oregon,'' you may want to make sure that ''outside'' doesn't mean 60 or 80 miles away. If it is, check with your support company about just how many people they keep on staff truly close to your ISP location.*

The Risks of Being Down

How long and how often can I afford to be down? This may seem like a silly question. When I ask it, I often get this incredulous look. For some installations, the answer is obvious — they can't afford to be down, period. This number is not, however, as high as it might seem. You see, the only true life-and-death kinds of applications are the ones that are in acute medical applications or are immediately tied to safety operations. Other installations may lose money — they may even cause bankruptcy if they go down — but that's not life and death either.

That being said, it's really not as black and white as all that. There is really something of a continuum in how critical downtime is. It ranges from the aforementioned medical applications at the high end to data-mining operations on old legacy systems at the low end (usually — for some companies, it may be all they have). The thing that pretty much everyone can agree on for every system is that downtime is highly undesirable.

So, the question becomes one of just how undesirable is it? How do we quantify that?

If you have a bunch of bean counters (I can get away with saying that since I was one) working for you, it shouldn't take you all that long to figure out that there are a lot of measurable costs to downtime. For example, if you have a bunch of employees sitting around saying that they can't do anything until the system comes back up, then the number of affected employees times their hourly cost (remember, the cost of an employee is more than just his or her wages) equals the cost of the system being down from a productivity standpoint. But wait, there's more. If you're running something that has online sales — how many sales did you lose because you couldn't be properly responsive to your customers? Oops — more cost. If you're running a plant with your system, then how many goods couldn't be produced because the system was down — or, even if you could still build them, did you lose quality assurance or other information that might cost you down the line?

I think by now you should be able to both see and sell to your boss the notion that downtime is very expensive — how expensive depends on your specific situation. Now the thing to do is to determine just how much you're willing to spend to make sure that it doesn't happen.

Lost Data

There's probably no measuring this one. In some cases, you can quantify this by the amount of cost you're going to incur reconstructing the data. Sometimes you simply can't reconstruct it, in which case you'll probably never know for sure just how much it cost you.

Again, how much you want to prevent this should affect your budget for redundant systems as well as things like backup tape drives and off-site archival services.

Is Performance Everything?

More often than not, the answer is no. It's important, but just how important has something of diminishing returns to it. For example, if buying those extra 10 percent of CPU power is going to save you two seconds per transaction — that may be a big deal if you have 50 data entry clerks trying to enter as much as they can a day. Over the course of a day, seemingly small amounts of time saved can add up. If each of those 50 clerks is performing 500 transactions a day, then saving two seconds per transaction adds up to over 13 man hours (that's over one person working all day!). Saving that time may allow you to delay a little longer in adding staff. The savings in wages will probably easily pay for the extra computing power.

The company next door may look at the situation a little differently, though — they may only have one or two employees; furthermore, the process that they are working in might be one where they spend a lengthy period of time just filing out the form — the actual transaction that stores it isn't that big of deal. In such a case, their extra dollars for the additional speed may not be worth it.

Driver Support

Let's start off by cutting to the chase — I don't at all recommend that you save a few dollars (or even a lot of dollars) when buying your server by purchasing it from some company like ''Bob's Pretty Fine Computers.'' Remember all those risks? Now, try introducing a strange mix of hardware and driver sets. Now imagine when you have a problem — you're quickly going to find all those companies pointing the finger at each other saying, ''It's their fault!'' Do you really want to be stuck in the middle?

What you want is the tried and true — the tested — the known. Servers — particularly data servers — are an area to stick with well-known, trusted names. I'm not advocating anyone in particular (no ads in this book!), but I'm talking very mainstream people like Dell, IBM, HP, and so on. Note that, when I say well-known, trusted names, I mean names that are known in servers. Just because someone sells a billion desktops a year doesn't mean they know anything about servers — it's almost like apples and oranges. They are terribly different.

By staying with well-known equipment, in addition to making sure that you have proper support when something fails, it also means that you're more likely to have that equipment survive upgrades well into the future. Each new version of the O/S only explicitly supports just so many pieces of equipment — you want to be sure that yours is one of them.

The Ideal System

Let me preface this by saying that there is no one ideal system. That being said, there is a general configuration (size excluded) that I and a very large number of other so-called ''experts'' seem to almost universally push as where you'd like to be if you had the budget for it. What we're talking about is drive arrangements here (the CPU and memory tends to be relative chicken feed budget- and setup-wise).

What you'd like to have is a mix of mirroring and RAID 5 or 10. You place the O/S and the logs on the mirrored drives (ideally on separate mirror sets). You place the data on the RAID 5/10 array. That way, the O/S and logs — which both tend to do a lot of serial operations — have a drive setup all of their own without being interfered with by the reads and writes of the actual data. The data has a multi-head read/write arrangement for maximum performance, while maintaining a level of redundancy.

Summary

Performance could be, and should be, in a book by itself (indeed, there is a Wrox title around the very subject). There's simply just too much to cover and get acquainted with to do it all in one or even several chapters. The way I've tried to address this is by pointing out performance issues throughout the book, so you could take them on a piece at a time. This chapter is all about the first of two different slants I'm taking on it — design (addressing performance *before* it is a problem). In our next chapter, we'll look at how we can identify and address performance issues when our system is already live. It's important to note that the techniques discussed there are ones you may want to also utilize while you're still in test so you can tweak your design accordingly.