Robert L. Bogue
MS MVP, MCSE, MCSA:Security
317-844-5310
Rob.Bogue@ThorProjects.com

Performance Improvement: Bigger and Better

In this four part series on performance we've reviewed the fundamentals of assessing performance including using the tools built into Windows to make these assessments. We've covered the considerations for session state, and we've walked through the benefits and problems with caching. However, we've not covered in detail what to do once you've assessed performance or how to leverage what you've learned about session state and caching to solve real world problems. In this article we'll be focused on isolating problems into solvable units and what to do when you believe that things just can't fixed.

Tracking Down Problems

In most cases, the process of improving performance is about 90% finding the problem and 10% fixing the problem.

If you've found out that you have a performance problem and have managed to isolate it as a CPU problem on the web front end servers, then what? The answer — no matter what you've isolated the problem to — is to continue to isolate it further until you no longer can. In most cases, the process of improving performance is about 90% finding the problem and 10% fixing the problem. So the most coarse metrics — those we talked about in the first part of this article series — are the first step on looking at the problem to determine what it may be. By implementing these metrics at each server in a system it is usually, but not always, possible to determine the key bottleneck for a system.

Break It Down

It's easy to say to break a problem down — but how do you do actually do that? In short, you do it by trying to break apart the problem into smaller and smaller units. If you're seeing a problem when an application is creating invoices, then there are several ways to break the problem down. You can purposefully break or skip the actual invoice generation piece in the code (comment it out, set it to a null handler, etc.) With the actual generation of the invoice broken you can see the impact on fetching the data for the invoices without having to consider the rendering code. This can help you to isolate where the issue is coming from and reduce the amount of detective work that you need to perform to find the offending code or construct.

If you've got multiple roles in your architecture, for instance database and front end, on the same server you can split them on to separate servers. This will allow you to determine which of the roles is causing the load. Anything that your architecture allows you to split should be split — even if it's only temporary so that the performance can be better understood.

Changing the Scenery

Another way to locate the root cause of a performance problem is to change the circumstances of the problem to either attempt to make the problem go away — or to make the problem worse. On the surface making a problem worse doesn't seem like a good idea — and generally speaking in production you're right. However, what if you suspect that there's a particular aspect of the program that is a problem — or a particular aspect of the data which may be causing issues? It may be the right thing to create an unrealistic set of circumstances in a development or testing environment to try to make the problem easier to find.

Whether you change the scenery or change it to make it worse or better you'll be closer to finding your performance issues just in knowing what environmental thing is contributing to the problem.

It's not always possible to do this; however, it is possible in more cases than one would expect. You can take all of the production data and place it on a development system that is less powerful than the production system so even a much smaller set of load will cause the performance problems to crop up.

Loops and Latency

Sometimes, however, the problems will not show their ugly heads. They don't look like a bottleneck on the server; instead everything appears to be working correctly. And yet, the site isn't performing like you expect or want it to. If you can't locate a bottleneck then you're probably on the hunt for a loop and some latency.

I clearly remember on portal I was working on that was taking over four seconds to log users in. We weren't seeing processing spikes. We weren't seeing database servers being hammered. In fact, the whole system seemed to be just humming along but we definitely couldn't deny that there was something wrong because users were indeed taking several seconds to get logged in.

We isolated the issue when we used a network monitoring tool to see what the network traffic from the portal server was. Imagine our surprise when we found that of the four seconds to log in nearly 3.8 seconds of that was the portal server talking to Active Directory. As it turns out, the design called for building a list of the groups that the user belonged to — either directly or indirectly. The problem with this in Active Directory is that in order to determine the entire list of groups that someone belongs to you have to do a query and get the membership for each group that they're a member of. This process is recursive. So each group required evaluating other groups and so on.

Because of the nature of the group memberships the system was walking through nearly 1000 groups per user. Of course, this is bad. However, that doesn't by itself explain 3.8 seconds. When you add to this a slight misconfiguration in Active Directory that caused the requests to be routed to an Active Directory server nearly 1000 miles away — with a wire speed latency of 40ms per request — it becomes easier to see how 3.8 seconds can get chewed up — and not show up anywhere as a bottleneck. It wasn't a bottleneck. It was simple latency multiplied by a large number.

In this case the short term solution was easy — fix the Active Directory misconfiguration. The longer term problem meant rearchitecting group memberships and changing the code to only walk one level of group nesting instead of an arbitrary number. With all of the changes the logon times for the application dropped to just over one second.

Many other performance issues that I've seen have been caused by active directory or DNS misconfigurations — but I'm careful not to jump to that conclusion particularly since the problem is generally not one-sided as the story above indicates.

Art and Science

Performance improvement is one part science — the mechanics of capturing details, making observations, etc. Performance improvement is also one part art. The art is knowing where to look for the cause of a particular problem. Obviously the coarse grained performance numbers can help you look for disk issues on the SQL server or CPU performance of the front end web application — but what if you're seeing both? Therein is the art. Knowing what to look for and knowing what is most likely not the problem takes experience and guesswork. While it's possible to teach the mechanics (as we've done in this series), it's not necessarily possible to teach the art component of performance improvement.

The challenge of anything that requires art and science is knowing when you need the art and when you need the science. In performance improvement the art is knowing what to test. The science is doing the testing — making the observations. You must be clear that it's the testing and observations that drives the activity — drives the next round of problem identification or the quest to create a solution. Performance issues are rarely solved by guesswork without the testing to support and validate that the guess is correct.

New Bottlenecks

> Removing any one problem or any one bottleneck won't necessarily resolve your performance issues completely.

One word of caution about finding bottlenecks in systems is that they're likely not the last bottleneck you'll find. You'll discover that your front end web servers are overwhelmed with processing. You'll add more servers, newer servers, or more processors and your CPU bottleneck will disappear. It will be replaced with a new bottleneck — at a new higher level of load.

Sometimes the next bottleneck is easy to spot but more often than not — they're impossible to see while the first bottleneck is controlling the scalability of the system. The trick here is to make sure that you realize that removing any one problem or any one bottleneck won't necessarily resolve your performance issues completely — it will just make them better. If you don't get to the performance or scalability level you want you'll get to go through the whole process again and isolate the parts of the solution which are creating the next bottleneck — until you are able to solve enough problems that the system performs adequately.

Splitting Workloads

While discussing the potential bottlenecks, I mentioned things that could be done to address the issues with memory, disk, and network. Adding additional CPUs for CPU issues is relatively obvious. However, these scaling techniques are techniques that all work on the same server. They're not techniques that will help once you've maxed out a single server. In this section we'll look at strategies for improving performance and scalability by splitting workloads across multiple servers.

Splitting Workloads to Their Own Servers

The first thing to do when you're trying to improve performance and scalability is to isolate workloads on to their own servers. If you've got the application pieces servicing the users on a server and the database on the same server you'll want to split those roles because in doing so you'll eliminate the competition for the same finite resources. However, doing this may cause issues if you've not tested it. Suddenly you'll have to deal with double-hop issues where authentication can't be retransmitted from the client to a process on another server. The NTLM protocol in Windows can't be retransmitted from one server to another and so if you're using NTLM and your database server is on the same machine — all is well. When you move it the authentication can't be transmitted so the application breaks because the users can't log into the database. Obviously, you can switch the database over to using a single SQL account rather than active directory accounts. You can also change authentication over to Kerberos which does allow for the delegation of authentication (when the application pool account is trusted for delegation.)

Despite this the move from a single server with all of the workloads on it to a set of servers is relatively trivial. It's also, generally speaking, relatively easy to test in development — by using a centralized database server from the developer workstations or by pointing one developer machine to another developer's machine for databases.

Multiple Front End Web Servers

It's probably no surprise that if you have one web server that you can improve performance and scalability by adding another. Certainly this works up to a point. However, it means that you have to consider as set of challenges that weren't important when the solution was on a single server. We've already covered load balancers and their settings. We've also covered session state and caching both of which are dramatically impacted by the decision to move to multiple front end web servers. (See Performance Improvement — Session State and Performance Improvement - Caching).

Migrating to a multiple server model — horizontal scaling — is the most effective way to improve the performance of a web application when the front end web servers are the bottleneck. However, because moving from one front end web server to multiple front end web servers means so much upheaval it's also a relatively expensive and risky way to improve performance and scalability.

Multiple Backend Database Servers

Database platforms are specifically designed to be scalable because they often become the central point on the back end that is hard to break apart. Database system software is extremely optimized and designed to leverage all of the resources available to them.

Memory, which is used quite effectively for caching, is a key way that they mitigate the need to access the relatively slower disks. They optimize reads by organizing the requests in a way that should be easier for the disk to respond to. By organizing the requests into clusters and into a relatively straight line across the disk (called elevator seeking) database servers can make better use of the disk resources that they have.

However, despite these optimizations there will be a point that it will simply be impossible to scale a single server. The server will run out of CPU bandwidth or bandwidth on the network. Breaking the back end database on to multiple database servers is one way to exceed this scalability limit. This is done by selecting some data that isn't related to the other data (and thus there's no need to do cross-database queries) and moving it to another database server (or clustered database servers).

Commonly any databases that are being used for session state or caching are moved to another database server while the primary data stores for the application are left on the same server as long as possible. Because coordinated caches and managing session state in a database makes for a very high IO operation set of databases and because these databases are by definition not related to other data in the system, they are ideal to be moved to other database servers or clusters.

Summary

Performance improvement often means breaking it down and building it back up. You'll have to break down the problem until you have a single, definable issue that you can develop strategies for resolving. It also means building up more servers, more memory, and more hardware to make more resources available for handling the load.