



**The Association of System
Performance Professionals**

The **Computer Measurement Group**, commonly called **CMG**, is a not for profit, worldwide organization of data processing professionals committed to the measurement and management of computer systems. CMG members are primarily concerned with performance evaluation of existing systems to maximize performance (eg. response time, throughput, etc.) and with capacity management where planned enhancements to existing systems or the design of new systems are evaluated to find the necessary resources required to provide adequate performance at a reasonable cost.

This paper was originally published in the Proceedings of the Computer Measurement Group's 2008 International Conference.

For more information on CMG please visit <http://www.cmq.org>

Copyright 2008 by The Computer Measurement Group, Inc. All Rights Reserved

Published by The Computer Measurement Group, Inc., a non-profit Illinois membership corporation. Permission to reprint in whole or in any part may be granted for educational and scientific purposes upon written application to the Editor, CMG Headquarters, 151 Fries Mill Road, Suite 104, Turnersville, NJ 08012. Permission is hereby granted to CMG members to reproduce this publication in whole or in part solely for internal distribution with the member's organization provided the copyright notice above is set forth in full text on the title page of each item reproduced. The ideas and concepts set forth in this publication are solely those of the respective authors, and not of CMG, and CMG does not endorse, guarantee or otherwise certify any such ideas or concepts in any application or usage. Printed in the United States of America.

Scaling Strategies and Tactics for Dynamic Web Applications

By:

Richard Campbell, Product Evangelist, Strangeloop Networks
&
Kent Alstad, Chief Technical Officer, Strangeloop Networks

Today's dynamic web applications are characterized by growing complexity and unpredictable loads and traffic patterns. As a result, they typically suffer from performance and scaling issues. Addressing these issues requires a recognition that modern web applications and the underlying networks that support them have become inextricably linked, placing new demands on networks and on application developers. To deliver acceptable performance, networking professionals must work more closely with application developers than has previously been required. This paper proposes a method for accurately measuring performance in dynamic web applications and offers several strategies for optimizing application performance and scalability.

1. INTRODUCTION

The tasks of accurately measuring and managing application performance are not new challenges. Performance management professionals have been studying these problems for years, initially in local-area network (LAN)/wide-area network (WAN) environments, and over the past two decades, in Internet Protocol (IP) networks. A recent trend, however, is driving the need for even more accurate application performance measurements and more effective performance optimizations: the increasing feature-richness of web applications.

While once strictly the domain of e-commerce web sites, IP networks and the Internet have become the transport media of choice for a variety of essential business processes. Users in virtually every industry now rely on IP-based applications daily to access information, process transactions, and communicate with users and systems that may be located anywhere in the world. As IP-based systems are called on to do more for businesses, users in turn are demanding more from the applications. They expect these business tools to behave not as web "sites," but as dynamic, data-driven web applications. To meet these demands, application developers and the vendors that support them have created a new generation of tools and strategies for making user interfaces richer and more "desktop"-like. Unfortunately, the added complexity of these rich features often takes a toll on application performance.

In the drive to boost performance in modern web applications and account for more variable payloads and traffic patterns, applications are becoming more closely tied to the network that supports them, and vice versa. Faced with this emerging reality, performance management professionals need to recognize the growing role that the network plays in application performance and scalability, and become more involved in the development and optimization of dynamic web applications than was required in the past. Ultimately, to achieve optimal performance in this new generation of web applications, networking professionals and application developers must work together and adopt new strategies for architecting both the network and the application.

This paper outlines techniques for accurately measuring performance and isolating performance problems in modern web applications and networks. The paper then examines three strategies – specialization, code optimization, and distribution – for improving the scalability (and ultimately, the performance) of dynamic web applications. Finally, the paper explores several specific optimization tactics, and offers examples based on one of the most widely used application development platforms, Microsoft's ASP.NET. Together, these techniques provide a framework for isolating specific elements of an application for independent scaling and for efficiently distributing workload across multiple servers.

It is the combination of all three strategies that ultimately offers the largest potential benefits in performance and scalability, but at the cost of increasing the complexity of managing the application and diagnosing issues. Networking professionals have an essential role to play in implementing performance optimizations for a given application – both in working with developers to identify the most effective application- and network-specific strategies, and in providing an empirical, data-driven framework to isolate issues and test real-world application performance.

2. MEASURING PERFORMANCE

The first step in optimizing any dynamic web application is creating a test measurement procedure to isolate the source of the performance degradation. Naturally, this requires a detailed understanding of the application and the underlying network, and how both are functioning in production.

A well-designed test protocol provides a detailed baseline of current application performance for comparison against the proposed optimization. The challenge, however, is to create a set of measurements that cover the behavior of the application from end to end. Omitting even a single aspect of application behavior can render the entire testing process meaningless. For example, a proposed optimization might involve using asynchronous JavaScript and extensible markup language (AJAX) to move the rendering of data into hypertext markup language (HTML) from server computation to client computation. There is a cost to downloading the script, but such an optimization will likely provide some performance benefit. Yet if the testing protocol does not include the cost of the JavaScript computation time on the client, it cannot accurately predict the actual benefit that the optimization will provide.

Sevcik and Wetzel of NetForecast previously proposed [1] a fundamental equation for use in measuring the performance of applications running over LANs, WANs, and the Internet. (Figure 1.)

Figure 1: Sevcik and Wetzel LAN/WAN/Internet Performance Equation

$$R \approx \frac{\text{Payload}}{\text{Bandwidth}} + \text{AppTurns}(\text{RTT}) + C_s + C_c$$

This equation examines the larger question of LAN/WAN/Internet application performance. With a few minor modifications, however, it can be employed

to examine web application performance specifically. (Figure 2.)

Figure 2: Web-Oriented Performance Equation

$$R \approx \frac{\text{Payload}}{\text{Bandwidth}} + \text{RTT} + \frac{\text{AppTurns}(\text{RTT})}{\text{Concurrent Requests}} + C_s + C_c$$

The elements of the equation are defined as follows:

- “**R**” is response time, which includes the entire time elapsed from the user requesting a page (by clicking a link, entering an address, etc.) to the entire page being rendered on the user’s computer. Response time may be measured in seconds or milliseconds.
- “**Payload**” is the total number of bytes sent to the browser, including the page itself and all resource files, such as cascading style sheets (CSS), JavaScript, images, etc.
- “**Bandwidth**” is the minimal bandwidth (bits per second) across all network links between the user and the application server. The slowest link is typically the user’s access line to the network, such as a digital subscriber line (DSL) or cable broadband link for Internet users. Bandwidth may be asymmetric on such links, with significantly higher downlinks than uplinks. Usable bandwidth may also be reduced by the effects of conflicting traffic (congestion) and protocol efficiency (e.g., transport control protocol [TCP] window). Typically, bandwidth across all links is averaged together to create a single figure.
- “**AppTurns**” accounts for the number of resource files a given page needs. These resource files will include CSS, JavaScript, images and any other files retrieved by the browser in the process of rendering the page. In the equation, the HTML page is accounted for separately by adding in round trip time (RTT) before the AppTurns expression.
- “**RTT**” is round trip time. This is the amount of time it takes, irrespective of bytes transferred, to communicate from the browser to the server and back again. Every request pays a minimum of one RTT for the web page itself. RTT is typically measured in milliseconds.
- “**Concurrent Requests**” indicates the number of simultaneous requests a browser will make for resource files. For practical purposes, the default concurrency value for most modern web applications using contemporary web browsers is two. (Microsoft

Internet Explorer defaults to two concurrent connections for Hypertext Transport Protocol [HTTP] 1.1 communications, as per RFC2616. [2] Firefox also adheres to RFC2616 guidelines and limits HTTP 1.1 to two concurrent requests. These limits can be adjusted but developers are advised not to alter them. [3] In the future, however, Microsoft Internet Explorer 8 will default to six concurrent requests. [4])

- “Cs” represents the compute time on the server. This is the time it takes for the application code to run, retrieve data from the database and compose the response to be sent to the browser. It is typically measured in milliseconds.
- “Cc” is the compute time on the client. This is the time it takes for a browser to actually render the HTML on the screen, execute JavaScript, implement CSS rules, etc.

As performance management professionals know well, measuring most of these elements (total response time, payload, bandwidth) is a straightforward proposition, easily achievable using any number of widely available tools. The dimension unique to web applications, however, is obtaining discrete measurements of server and client computation times, which requires some coding. On the server side, testers can add some relatively straightforward code to an application page to note the exact time at the start of the execution of a page and subtract that from the current time at the end of the execution of the page.

The same holds true when measuring client computation time. Testers can code JavaScript to execute at the top of the HTML page to note the start time and then subtract the time at the point that the OnLoad event fires when the page is completed.

The challenging part of handling the computation times of client and server is storing the results effectively. Often the simplest solution is to create a “debug mode” of the web application that displays those times directly into the page. This is not a scalable technique, but it is highly effective for debugging. Indeed, thorough performance testers may choose to utilize the debug mode to display all elements of the performance equation. Doing so allows testers to routinely render the performance equation elements on the browser and more easily isolate the source of performance problems. For example, consider a web application with users located on a different continent than the application servers, accessing the application over low-bandwidth connections. The performance equation will

immediately reflect high ping times (>200 milliseconds) and low bandwidth (<500 kilobits per second). Such measurements would indicate that users are highly sensitive to the total payload and number of round trips required by the application – likely far more sensitive than the developers testing the same application with a high-speed LAN connection to local servers. Adding the capability to easily recruit any user in any location to turn on the debug mode and report the resulting performance equation values allows performance management professionals to immediately identify where the performance problems lie.

Armed with the accurate metrics, performance management professionals can achieve a much more detailed understanding of how the application functions for users. They can then make recommendations for directing optimization resources that are far more likely to yield significant results for the application.

3. SOLVING SCALABILITY ISSUES

The methodology described above deals chiefly with isolating performance problems in various elements of a web application. In real-world dynamic web applications, however, poor performance often arises as a result of scalability issues – the inability of some aspects of the application to scale as they are called on by larger numbers of concurrent users – rather than the poor performance of any individual element. Scaling problems often don’t appear until the worst possible time – once the application is deployed and well into the adoption curve. When problems show up due to load,, the issue is likely scaling

Solving the scaling problem is not a simple task – hence the need for accurate performance metrics and data-driven testing. Every application scales differently, and network professionals will need to work closely with developers to craft an effective strategy for scaling a given application. However, while the specific interventions that may be employed in a given application will vary, there are three overarching scaling strategies that can be employed: specialization, code optimization, and distribution. The following sections explore each strategy in detail. [6]

3.1 Specialization

Specialization is often the scaling strategy that can yield the most impressive results in improving application performance. Fundamentally, specialization aims to break the scaling problem (and effectively, the application) into smaller pieces with the goal of making the problem easier to solve. This

strategy also tends to yield more cost-effective optimizations, especially when it isolates a part of the application that is especially difficult to scale. Specialization cannot be artificially imposed on an application, however. Network professionals and developers must find the natural partitioning points; the functions that can be easily separated and that will benefit from specialized processing.

One natural partitioning strategy is by resource type, such as images. In fact, moving static resource files away from the application servers – whether such files are images, CSS, JavaScript or other files – can be one of the most effective ways to improve the scalability of dynamic web applications. A Microsoft Internet Information Services (IIS) server that has been specifically tuned for an ASP.NET application, for example, is by default not specifically tuned to serve static resource files. Incorporating a separate group of IIS servers well tuned to serve such files can make a substantial difference in scalability. Of course, the ultimate benefit of this type of specialization depends on the application and the level of demand such resource files create for it.

Image servers offer a compelling example of specialization, in that they move some input/output (I/O)-intensive work away from application servers in a straightforward and easily measured way. But there are many other possible specialization strategies. Processing power presents another example, and a common problem with application servers that are doing compression or encryption (for Secure Sockets Layer [SSL] communication, for example). Indeed, implementing dedicated SSL servers has been a commonly employed technique for scaling web sites since the 1990s. Network professionals can even employ specialized hardware devices for compression and SSL termination.

Performance management professionals can also consider the development of platform-specific specializations. Some within the ASP.NET development community, for example, have proposed partitioning applications between ASP.NET and business objects servers – creating a separate tier to perform data access work, complex computations, etc., independent of the actual generation of the web pages to be sent to the user's browser. This specific strategy can carry a cost, however, in that it creates a need for out-of-process calls or remote calls between web servers and business object servers, which can create significant additional overhead.

While a variety of strategies are possible, the primary factor that should guide any specialization decision is whether the proposed strategy can achieve a known benefit. This area in particular is one in which performance management professionals can play a

critical role by emphasizing the need for accurate testing as a pre-requisite of any specialization effort. Along these lines, performance management professionals may be better suited than others in the development and optimization process to reinforce the notion that the fastest scaling solutions are not always the best. The goal for a well-scaling application is to achieve consistency of performance, not necessarily peak performance. Ultimately, an effective scaling optimization should narrow the performance range of the servers as the load increases. Whether an application has one user or a thousand, the scalability goal is to render a given page in the same amount of time.

3.2 Code Optimization

Code optimization is the strategy to which developers often turn first when a performance problem arises, even if that performance problem is actually a scaling problem. While optimizing code can yield benefits, developers need to be sure they are addressing the right problem. Once again, network and performance management professionals can play an important role in this process by reinforcing the need for testing and data-driven optimizations.

Fortunately, most scaling issues ultimately center around some aspect of Cs (compute time on the server). Virtually every other aspect of the performance equation scales linearly. Bandwidth can always be added as needed, and the compute time on the client does not change as the number of clients increases. All other elements of the performance equation also remain consistent as the application scales, except for compute time on the server.

As developers engage in optimizations of server code to scale more effectively, network professionals can assist in this process by helping developers test their assumptions and proposed optimizations to ensure that they will actually function as desired. Network professionals and developers should use profiling tools to analyze the application and learn where it spends the most time. The entire process should be empirical: testing to find which code to improve, improving the code, retesting to confirm that the optimization has delivered the expected results. The process is also likely to be continuous, especially in larger-scale sites. As Jim Benedetto, vice president of technology for MySpace has said, "Our developers have repeatedly redesigned the web site's software to scale, but the job is never done. It's kind of like painting the Golden Gate Bridge, where every time you finish, it's time to start over again." [5]

3.3 Distribution

Many organizations facing scaling issues attempt to solve the problem through distribution alone, by simply adding more hardware. Certainly, adding hardware can help an application scale. But without the other scaling strategies discussed, the return on distribution can be small. Indeed, sometimes, adding more hardware offers no benefit at all. The specialization and code optimization strategies discussed in the previous two sections directly benefit distribution.

Specialization allows for distribution of component parts of the application, as needed. If the network has incorporated separate, specialized image servers, for example, scaling image services independently of the rest of the application becomes a simple task. Optimization also provides dividends for distribution by reducing the amount of work needed for a given operation. This translates directly into fewer servers needed to scale to the same number of users. In theory, organizations that have already invested the time in specialization and optimization should be able to follow a few simple steps to implement a distribution strategy: add servers, duplicate the application across the servers, implement load balancing between the servers. In practice, however, implementing an effective distribution strategy is a more complicated endeavor.

3.3.1 LOAD-BALANCING

While duplicating servers is a relatively straightforward process (with a variety of tools available to accomplish it), the first hurdle in effective distribution is employing efficient load balancing. There are a number of load balancing technologies available. Indeed, any organization using Microsoft Windows servers likely already owns one: Network Load Balancing (NLB). NLB is a service that is included with all editions of Windows 2003 and 2008 Server, and serves as a good example of a commonly employed, software-based load-balancing solution.

With NLB, every server is an equal partner in the load balancing relationship. All servers employ the same algorithm for load balancing and all listen on a shared virtual IP for all traffic. Based on the load balancing algorithm, each server knows which server should be working on a given request. Every server in the cluster sends out a heartbeat to let the other servers know it is alive. When a server fails, the heartbeat for that server stops and the other servers compensate automatically.

NLB works well when an application has a large number of users making fairly similar requests every time. It becomes less effective in dynamic web

applications, however, as it cannot compensate for certain requests creating much more load than others. Hardware load balancing solutions are available that can better address this scenario and offer a greater variety of options for load balancing, but the most effective hardware solutions may be beyond the budget of many organizations.

3.3.2 ADDRESSING AFFINITIES

Ultimately, implementing load balancing is not the most significant challenge associated with effective distribution. A larger problem is eliminating affinity. Effectively, some application processes that are perfectly straightforward when employed on a single server become extremely complex when multiple servers are involved. The classic example of this problem is session. When an application uses only a single web server, storing session data in the web server makes sense. The session data is right where the application needs it. But what happens to session data when the environment is expanded to more than one web server?

One approach is to keep the session data in the web server and use a technique known as “stickiness” or “affinity.” Essentially, this means that the first request from a given user is load-balanced and after that, all subsequent requests from that user are sent to the same server as the first request. This is a simple approach, supported by every load balancing solution, and may even be viable in smaller-scale networks. In the long term, however, this affinity causes many more problems than it solves. The best way to illustrate this is to examine a dynamic web application based on ASP.NET.

Keeping session data in-process in an ASP.NET application may be fast, but if the ASP.NET worker process recycles, all those sessions are lost. And worker processes may recycle for many reasons. Under high load, IIS might recycle the worker process of ASP.NET because it thinks it is stuck. Indeed, some versions of IIS recycle ASP.NET worker processes every 23 hours by default. These mechanisms can be adjusted, but in all cases, users remain vulnerable to losing their session data while it is in-process. This issue may not be as important in smaller-scale applications, but for applications serving thousands or millions of users, it becomes a significant problem. And recycling ASP.NET worker processes is just one of many potential affinity issues.

If the network is engineered to load balance by IP address, eventually one server is going to get hit by a “megaproxy” Internet service provider, such as AOL, and be unable to service that entire load on its own. Updating servers with a new version of the application

also becomes more difficult. Network engineers must either wait for hours for users to finish using the application or annoy some segment of users by dropping their sessions. Reliability also becomes an issue. If the network loses a server, the application loses many sessions.

Eliminating affinity should be a key goal of any distribution strategy. This means moving session out-of-process – in effect, taking a response-time hit to provide a scalability increase. For ASP.NET applications, session data should be recorded in a place where all web servers can access it, either on the SQL Server or State Server, and configured using web.config.

Implementing out-of-process session in ASP.NET applications also requires an additional coding effort. Any classes that will be stored in the Session object need to be marked with the Serializable attribute. That means that all data in the class needs to be serializable or marked as NonSerialized, so it will be ignored. If developers fail to mark up the class, when the serializer runs to store session data out-of-process, it will return errors. Typically, implementing such coding changes is a relatively trivial task. In ASP.NET applications that use non-serializable data types in their classes, however – especially if those non-serializable data types are part of a third-party library that developers cannot code for—the process becomes more problematic.

Moving session out-of-process can also reveal other issues, such as session objects in an ASP.NET application using too much data. This problem quickly becomes apparent when the session object is transferred back and forth across the network twice (once to retrieve it at the beginning of the page, once to return it at the end of the page) for every page request.

After addressing the session affinity, network professionals and developers can turn to other affinity issues (such as Membership and Role in ASP.NET applications). Each affinity is likely to pose its own challenge. But for dynamic applications to truly scale, developers and network professionals should search for and eliminate every affinity they can find.

4. TACTICAL OPTIMIZATIONS

The previous sections of this paper have focused on high-level strategies for scaling dynamic web applications. The most effective scaling and performance optimizations, however, will address the specific characteristics of the application and, by default, the specific characteristics of the platform in which the application runs.

The following sections discuss a series of specific scaling and performance optimization tactics, employing examples specific to the widely used application development platform ASP.NET. [7] As with all previous strategies discussed, any optimization effort should be guided by empirical data regarding the areas of the application that could benefit most from optimization, and by a rigorous testing protocol.

Tactics discussed include:

- Payload Optimization Tactics
 - Compression
 - ViewState
 - AJAX
- Caching Optimization Tactics
 - Memory Issues
 - Expiration
 - Evaluating Caching Tactics
- Database Scaling Tactics

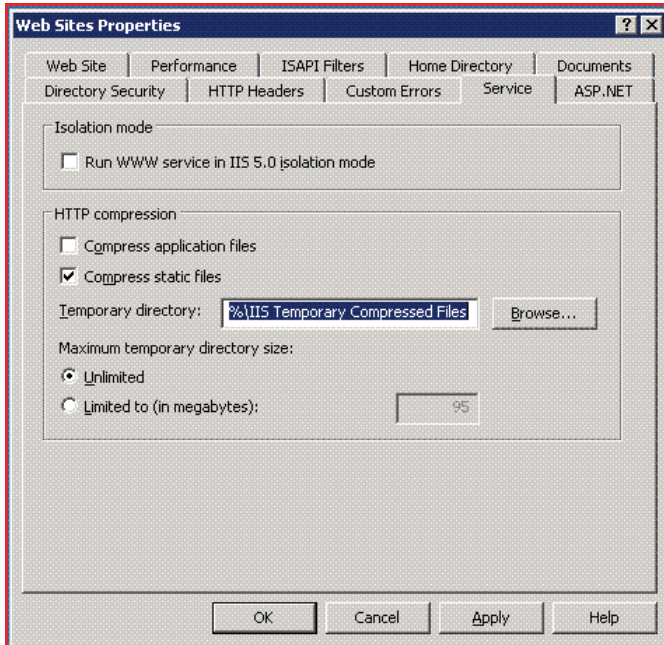
4.1 Payload Optimization Tactics

As the performance equation introduced above makes plain, payload can play a significant role in overall application performance, especially in bandwidth-limited environments. Cutting down payload size can offer a number of benefits in several areas. Response times improve as a result of shipping less data. Applications may scale more effectively as a result of having to move fewer bytes. Minimizing payload can even offer savings in bandwidth costs.

4.1.1 COMPRESSION

One of the simplest techniques for decreasing payload size in dynamic web applications is using compression. In an ASP.NET application, IIS6 allows developers to specify whether to compress only static files, dynamically generated responses (ASP.NET pages, for example) or both (Figure 3.) For static files, IIS6 compresses on demand, storing these files in a specified compressed files cache. For dynamically generated responses, no copy is stored. The response is compressed every time. IIS7 functions slightly differently by incorporating more intelligence into the compression process, compressing only files that are used frequently.

Figure 3: Configuring Compression Server-Wide in IIS6



Compression costs processor cycles, which is typically not a problem on a dedicated web server. But when the processor gets too busy, IIS7 automatically suspends compression efforts.

Organizations can also use dedicated devices for compression that operate independent of the web server itself.

4.1.2 VIEWSTATE

Another area that network professionals and developers can consider for reducing payload in ASP.NET applications is ViewState. Most web controls for ASP.NET use some ViewState, and on control-intensive pages, it is not uncommon for ViewState to grow to thousands of bytes. Naturally, the first step in addressing ViewState in a web application is accurately assessing the size of the ViewState on web pages. As a rule, developers should be encouraged to turn off ViewState features on controls where they are not needed, or even consider eliminating some controls that use excessive ViewState. Fortunately, most modern web controls are sensitive to the ViewState problem and provide some granular control of the amount of ViewState they use.

4.1.3 AJAX

One of the most dramatic and increasingly popular techniques for reducing payload in dynamic web applications is AJAX. However, while AJAX reduces the perceived sized of the payload for the user, the total number of bytes sent to the browser is in fact

increased. With AJAX, the “parent” page is smaller, so that initial render times are faster. Individual elements in that page then make their own requests to the server to populate data. Effectively, AJAX spreads the payload out over time, giving the user something to look at while other elements of the page load. As a result, using AJAX can improve the user experience overall, but network professionals and developers should use the performance equation to measure the true costs of this approach.

For example, AJAX typically increases compute time on the client, sometimes dramatically, to the point that performance is unacceptable. AJAX also creates a new class of roundtrips to the server in order to populate individual elements. If these extra trips are replacing former entire page requests, they will provide a net decrease in round trips. But in many cases, the total number of round trips for a given user will actually increase. Ultimately, using AJAX may actually require application servers to work harder, even though that work may be spread out over more time.

While AJAX is not a panacea, it does offer some remarkable benefits and is well worth considering as part of any performance optimization effort. However, network professionals should again emphasize the need to test carefully and repeatedly. Any commitment to a large-scale AJAX implementation should be based on an empirical comparison of the benefits of this approach versus the true costs of the migration.

4.2 Caching Optimization Tactics

Experts in scaling dynamic web applications often mention caching as a key strategy for optimizing application performance. Fundamentally, caching is about moving data closer to the user. In a typical application, before any significant optimization work has been undertaken, virtually all the data the user needs is in the database and retrieved from the database with every request. Caching alters that behavior. The drawback to caching is the increased complexity of the application. In many cases, however, the scalability and performance benefits outweigh that cost.

Several forms of caching are possible. The ASP.NET platform supports three: page caching (also known as output caching), partial-page caching and programmatic (also known as data) caching.

Page caching is the simplest form of caching. In ASP.NET, developers simply add an `@OutputCache` directive to the ASP.NET page, including a rule for when to expire it. For example, developers can specify that the page should be cached for 60

seconds. With that directive in place, the first request of that page will process normally, accessing the database and whatever other resources are needed to generate the page. After that, the page is held in memory on the web server for 60 seconds and all requests during that time are served directly from memory.

Partial-page caching addresses the blunt reality of page caching: that virtually no web page in the world is so static that the entire page can be cached for any length of time. Partial-page caching allows portions of a page to be marked as cachable so that only the parts of the page that do change regularly are computed. The process is complicated but often effective.

Arguably the most powerful (and most complex) form of caching is programmatic caching. This technique focuses not on pages themselves, but on the objects within them. The most common use of programmatic caching is to cache data retrieved from the database.

The overarching benefit of all forms caching is obvious: it is much easier for the server to deliver a page or object from memory than to compute it. However there are two challenges with caching that need to be considered: memory issues and expiry.

4.2.1 MEMORY ISSUES

On a busy server, memory becomes a significant issue when caching, especially in the ASP.NET platform, for a variety of reasons. Whenever an ASP.NET page is computed, it uses some memory. The .NET Framework is designed to allocate memory very quickly but release it relatively slowly, through automatic Garbage Collection. This memory allocation scheme can raise a number of issues in dynamic web applications, but for the purposes of caching, the key point is that on a busy web server, the memory space available for the ASP.NET application is in high demand. Ideally, most of that memory usage is temporary, as it is when creating variables and structures for computing a web page. But when it comes to persistent memory objects, such as in-process session and cache objects, memory usage becomes much more problematic. And these problems typically arise only when the application is very busy.

Consider the following scenario: A consumer-facing web application is experiencing heavy load as a result of a promotion, and thousands of users are hitting the web site. To preserve performance, the application and network are designed to cache portions of pages and groups of data objects whenever possible. Each page request from a user consumes a bit of memory,

so the amount of consumed memory keeps going up. The more users, the faster that amount increases. There are also the big jumps in memory consumption from the cache objects and session objects. As the total memory used approaches 80 percent, ASP.NET calls a Garbage Collection event. The garbage collector works its way through the memory space, shuffling down persisted memory objects (like cache objects and session objects) and freeing up memory that is no longer used (i.e., the memory that was used to compute the web pages). The process of freeing up unused memory is fast, but shuffling persisted objects is slow. So the more persisted objects in memory, the slower the Garbage Collection process.

Note that while Garbage Collection is processing, no pages can be served by that ASP.NET server. Everything is held in a queue, awaiting Garbage Collection completion. IIS can exacerbate such scenarios. If the delay in Garbage Collection takes too long, IIS may decide that the process is frozen and cycle the worker thread. Doing so does free up significant memory – at the cost of all of the persisted memory objects on which active users are relying. ASP.NET now offers a patch that will automatically expel objects from the programmatic cache if the server gets low on memory. While this mechanism can provide some immediate relief for this issue and prevent a crash, it is not a long-term solution. After all, if the application is coded to use cache, anything that is expelled from the cache, because of Garbage Collection, will likely be cached again.

There is some relief for this memory issue if the application will run on a 64 bit server. Instead of being limited to 2 GB, the limitation will be the physical memory of the machine. ASP.NET applications, which typically can read achieve a working set of approximately 800 MB on a 32 bit server will be able to use working sets 5 times or more larger. While 64 bit means developers don't need to spend as much time on memory management, for large implementations it simply moves the problem out. The application will be able to withstand greater load before encountering these memory issues.

4.2.2 EXPIRATION

While memory usage must be addressed, the most difficult problem in any caching strategy is accounting for changes in the underlying data. Indeed, expiration of caching is the biggest challenge developers face in implementing caching in any form. Literally within moments of something being cached, it has the potential to be out of date. A simple example comes from the database, such as the number of products available for sale. In the initial incarnation of the product page, every rendering of that page involves a request from the database for the number of products

still in inventory. Analyzing those requests will likely reveal that 99 percent of the time, the same number is retrieved over and over again, leading to the temptation to cache that content.

A simple way to cache it would be over time, for example, caching the inventory of the product for one hour. The downside to this strategy is that if a customer buys the product, and then goes back to the page and sees that the inventory is still the same, they may doubt the accuracy of the entire application and complain. Far more challenging, of course, is the scenario of a customer seeing that inventory is available and going to buy the product when it is actually sold out.

Obviously, a simple time-based expiration scheme cannot suffice. An alternative might be to cache the inventory count until someone buys the product, and then expire the cache object. This approach appears more functional, but it does not address networks using more than one server. Depending on which server a customer communicates with, he or she will see different inventory counts for the product. Additionally, considering that in many organizations, receiving (which adds more inventory) is not even integrated into the web application, even this more sophisticated caching strategy may not be viable.

Synchronizing expirations between servers can be done, but must be done carefully. The amount of chatter that can be generated between web servers goes up geometrically as the number of cache objects and web servers increases. Effectively, employing this caching mechanism sets a limit on the number of servers that can be added to the application before the network chatter overwhelms it.

The impact of cache expiration on performance must also be considered for scalable design. If cache checking and blocking strategies validated for high load conditions, expiring a cache object can cause significant problems. For example, consider an expensive query that takes 30 seconds to return from the database and is called frequently. That query seems ripe for caching, and the code for handling cache objects is likely simple. The program is designed to cache the object after the first query. With simply blocking applied, the process then becomes:

- When a query arrives, the application first checks to see if the cache object is populated.
- If it is, it uses the data from the cache object.
- If not, it executes the code to retrieve the data from the database, and the application sets a block for additional requests.
- After 30 seconds, the query completes and application populates the cache object with that data, removes the lock;

- And code continues to execute as normal.

Now consider this situation under heavy load – one query per second. In the 30 seconds that it takes for the first query to run 29 other requests will come in. Each will check the cache object, find it unpopulated, and attempt to populate it, but encounter the lock. When the first lock is released, query number 2 will run, another lock will be placed and queries 3 – 30 will continue to wait. Some queries could be waiting several minutes, instead of 30 seconds, clearly unacceptable, and unplanned poor performance.

One approach to solving this is to add a second check, so that after the lock is removed subsequent requests check to see if the cache is populated before attempting to run their query.

This example of poor locking code seems simple, but is frequently the cause of degraded performance, and demonstrates the complexity of caching code. It also highlights another issue of caching code: It requires experienced developers to write and maintain. This is a more subtle cost, effectively increasing the price of all development going forward. The increase in complexity caused by caching slows down all future feature development and requires more training, more testing and more debugging.

4.2.3 EVALUATING CACHING TACTICS

While the performance benefits of caching can be tremendous, implementing caching mechanisms that work well can be an extremely difficult process. Additionally, since caching adds complexity to the application and the network, it should be used judiciously. Once again, performance management professionals have an important role to play in assisting developers with testing processes and gathering empirical data to demonstrate that a caching strategy will indeed deliver significant benefits before undertaking it.

In addition, performance management professionals should ensure that any caching strategy is tested against complex real-world scenarios. Network professionals and developers should know exactly what happens in their environment during simultaneous requests or when expirations come quickly before moving forward with the caching effort.

4.3 Database Scaling Tactics

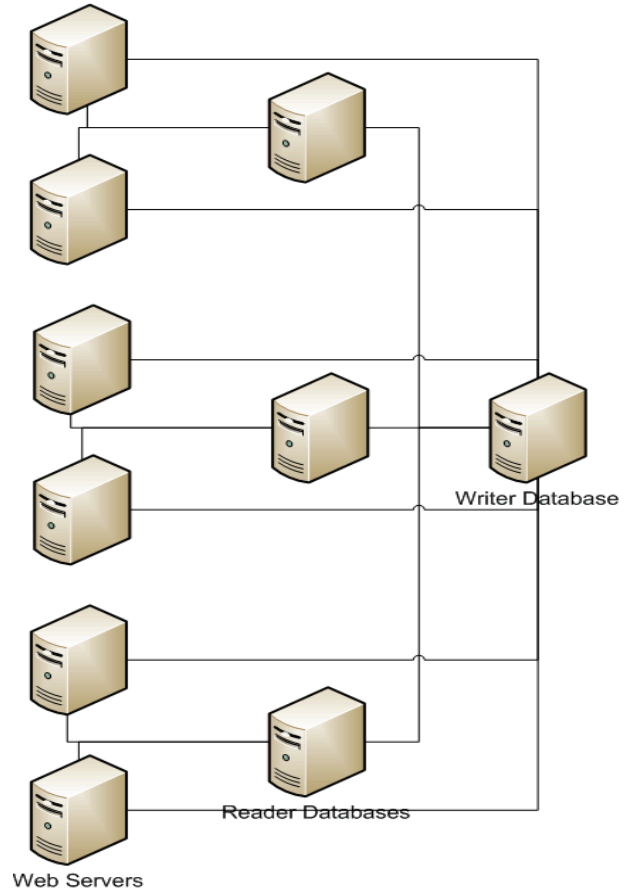
The typical approach used in scaling dynamic web applications is to scale out rather than scale up. Between the thread and memory limitations of

platforms such as ASP.NET and the short-term nature of web requests, scaling out with multiple mid-range web servers is often far more practical than building a highly optimized web server that does everything required in one box. When it comes to scaling databases, however, the standard practice is to scale up: to implement one massive appliance, or perhaps two in a failsafe clustered configuration (although only one is actually running the database at any given time). Eventually however, in every large-scale web application, a single database cannot handle the load, and scaling out is required.

Scaling out databases is entirely feasible. Network professionals and developers need only apply the same strategies as applied to the web application itself. The first step is always specialization – breaking the database into logical partitions. Those partitions can be data-centric, perhaps by region. This model creates multiple databases that each contain a portion of the whole database. One server could have East Coast data, for example, while the other has West Coast data. The assumption underlying this strategy is that most people on the East Coast only need access to the East Coast data, and those on the West Coast need West Coast data, with very few users requiring data from both coasts. In effect, this strategy increases the performance of the database for the majority of users at the expense of the few.

The largest-scale web applications go a step farther, partitioning their databases into readers and writers. (Figure 4.) The reader databases are read-only and receive their data from the writer databases via replication. All querying of data comes from the reader databases, and they are optimized for reading as fast as possible. Reader databases are by their nature very distributable. However, partitioning databases into readers and writers is difficult, hence the reason it is typically only used by the largest scale sites.

Figure 4. Distributed Database Architecture in which Web Servers Read from a Local Database but Write to the Central Database



All write requests for the application are sent to the writer databases, which might be partitioned. The writer databases are tuned to write efficiently. Replication moves the new data out to the reader databases.

The drawback of creating specialized databases is that this introduces lag time. A write now takes a certain amount of time to be distributed to the reader databases. If the latency is acceptable, however, the scaling potential is enormous.

5. CONCLUSION

As dynamic web applications continue to deliver richer features, controls and user experiences, performance management professionals will be continually challenged to maintain and improve performance. By developing accurate testing protocols, and implementing sound scaling strategies and performance optimization tactics based on the results of those protocols, performance management professionals and developers can stay ahead of evolving application demands.

As long as an application continues to grow, however, network professionals should expect to see their efforts to scale that application and maintain performance grow as well. Scaling techniques that work well for 10,000 simultaneous users are often not as effective with 100,000 users, and the rules change once again with 1,000,000 users. Ultimately, the ideal scaling strategy will be unique to every dynamic web application and to each application's unique evolution.

The key to successfully implementing any scaling strategy or optimization tactic, however, is employing empirical testing to guide optimization decisions and resource allocation. Performance management professionals should continually emphasize the need to test to ensure that optimization efforts are directed where they are needed most. Network professionals and developers must continually test their work to confirm that optimizations have actually delivered the expected improvement.

At the end of a scaling development cycle, performance management professionals need performance metrics that identify the processes that are most impacting performance, as this is the starting point for the next round of improvements. Even as one round closes and the application is now fast enough for the users today, network professionals and developers must begin work on improvements for the users of tomorrow.

6. REFERENCES

[1] P. Sevcik and R. Wetzel, "NetForecast Report: Field Guide to Application Delivery Systems," 2006.

[2] Internet Engineering Task Force Network Working Group, RFC 2616, section 8.1.4.

[3] Microsoft Corporation, Microsoft Knowledge Base, Article 183110.

[4] Microsoft Corporation, "How do I make my site 'light up' with Internet Explorer 8?" Windows Internet Explorer 8 Readiness Toolkit

[5] D. Carr, "Inside MySpace: The Story," *Baseline Magazine*, 2007.

[6] K. Alstad and R. Campbell, "Scaling Strategies & Tactics," MSDN Magazine, 2008

[7] K. Alstad and R. Campbell, "Scaling Strategies & Tactics," MSDN Magazine, 2008