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

Performance Improvements: Caching

If you're looking at performance and you want to get some quick wins, the obvious place to start is caching. Caching as a concept is focused exclusively around improving performance. It's been used in disk controllers, processors, and other hardware devices since nearly the beginning of computing. Various software methods have been devised to do caching as well. Fundamentally caching has one limitation — managing updates — and several decisions. In this article, we'll explore the basic options for caching and their impact on performance.

Note:*You can learn about other performance improvements in the articles, Performance Improvements: Understanding and Performance Improvements: Session State.*

Cache Updates

Caching replaces slower operations—like making calls to a SQL server to an instantiate an object—with faster operations like reading a serialized copy of the object from memory. This can dramatically improve the performance of reading the object; however, what happens when the object changes from time to time? Take, for instance, a common scenario where the user has a cart of products. It's normal, and encouraged, for the user to change what's in their shopping cart. However, if you're displaying a quick summary of the items in a user's cart on each page, it may not be something that you want to read from the database each time.

That's where managing cache updates comes in, you have to decide how to manage these updates and still have a cache. At the highest level you have two different strategies. The first set of strategies is a synchronized approach where it's important to maintain synchronization of the object at all times. The second strategy is a lazy (or time) based updating strategy where having a completely up-to-date object is nice but not essential. Say for instance that you had an update to a product's description to include a new award the product has won — that may not be essential to know about immediately in the application.

Synchronized Update

With a synchronized update, sometimes called coherent cache, you're trying to make sure that updates are synchronized between all of the consumers of the cache. When you're on a single server this is pretty easy. You have access to an in-memory cache and you just clear the value and update it with the new value. However, in a farm where you have multiple servers you have a substantially harder problem to solve. You must now use some sort of a coordination point (like a database, out of process memory store, etc.) and take the performance impact of that synchronization, or you must develop a coordination strategy for the cache.

The problem with the synchronized approach is that synchronization reduces the performance of the cache, which is of course the whole reason why it exists. There are implementations of this strategy that require only a small of the synchronization data to be shared, and there are strategies that essentially move the cache into a serialized database not unlike persisted session state. In either strategy the impact of synchronization is not trivial.

The synchronization strategy is good when there's a relatively large amount of change in the cache. For instance, if you needed to know the last few products a user looked at, that's going to change rapidly, then you may want not want to use a synchronized strategy for that data.

Another approach to synchronized cache is to use a coordination approach. With this approach you believe that the number of updates will be small and because of that you require extra actions when the cache is updated, rather than requiring extra activities on every (or nearly every) read. In general the kinds of data that you want to cache have relatively low volatility (rate of change) so the coordination approach to keeping cache's synchronized is often the best choice for synchronized cache. In this situation, when the object changes, and that change is persisted to whatever the back end store is, the object notifies the cache manager and the cache manager notifies all of the servers running the application.

For instance, in a farm with three servers, the cache manager clears the cache on the currently running servers and then signals the other cache managers — say via a web service — that they should clear their cache for a specific object. The number of calls needed obviously increases the larger the farm but as a practical matter the frequency is relatively low.

The final approach to a synchronized cache is having it client managed via a cookie. This strategy is really only viable for user data/session cache when the cached data is small (so that it can be retransmitted without impact), volatile (so that another strategy won't work well), and non-sensitive (so that the user having the information doesn't have any potential security implications.) The example of the recently visited products is a great example of a cache that could be pushed down to the client. This solves the need for it to be managed in cache.

Lazy Update

In some situations such as the product description changing to include a new award for the product the change needs to be visible to all of the members of the farm and in every session eventually but the applications usefulness doesn't rely upon an absolute up to the minute cache to function correctly. In these situations there are two basic approaches that can be applied based on your scenario. First, you can allow the cache to expire after a certain passage of time. For instance, caching information about a product is useful but perhaps you decide that the product should only be cached for ten minutes. This decision is made because most of the time if the product isn't accessed within the previous ten minutes it's not needed any longer.

Depending upon your scenario you can choose to apply a fixed window (e.g. every ten minutes the product object in memory is updated) or a sliding window (e.g. only after ten minutes of inactivity is the product object in memory updated). Of course, we're really talking about expiring the cache so in both scenarios it's really just that the objects will be cleared

from the cache at those intervals and when they are reloaded or regenerated could potentially be much longer, but the older data would only be returned while the object is in cache.

When you know specifically when the object data is going to change, say for instance during a batched update in the middle of the night, you can choose to update the entire cache, generally for a class of objects, at the same time. The effect of this is that you are able to maintain cache efficiency except for the onetime per day when you know that updates are happening.

Read-Through

The final scenario that introduces a bit of awareness in your objects that caching is involved is to allow for a read-through strategy where the consumer of the cached objects can specifically request that the object be read from “source of truth” rather than relying upon the cache. This can be useful when there are specific scenarios where it’s critical to know the exact right answer. For instance, consider the importance of having the exact right value during a checkout on a commerce site.

Read-Through strategies don’t solve the issues around a cache getting stale. They just recognize that there are specific times when read operations are critical and other situations where they’re less critical. In recognizing this the length of time that a cache can be scale can be longer since critical operations will still use the correct data.

Caching Options

Whether your caching needs are not volatile at all or are very volatile you have another decision to make about what level at which to cache data. There are techniques built into ASP.NET which allow you to cache the output of a control as well as caching individual objects like we've been discussing. With both of these options available the question is bound to come up about which of these techniques you should use or whether you should use both.

The key benefit of output caching is that once the data is processed for output once it doesn't have to be reprocessed. In other words, if you have some complex algorithm for building a set of data or a graph, then caching the output of that process would prevent reprocessing the same data over and over again. Caching the output is also good when you only need a small subset of the data to create the output and caching the objects themselves would require a larger cache.

However, as a general rule the use of output caching produces a larger cache because you may have a few different visual representations of the same data. For instance, you might have a control which is a vertical sidebar representation of a cart, the actual cart display page, and a summary indicator for the number of items which appears on every page. These controls would likely require more memory if cached at the output level than would be required to cache the cart and rebuild them.

Of course, for this kind of cache disk files can be used. SharePoint, for instance, uses this strategy for page output caching. This approach can be valuable particularly if most of what you're generating is coming from a database and thus is relatively expensive to get to. Nearly all content in a SharePoint site comes from a database. Of course, if you decide

on a strategy that utilizes the disks on the front in web servers you'll want to make sure that the disk performance is acceptable. This generally means going for SAS (Serially Attached SCSI) instead of SATA disks to get better responsiveness.

The reality is that processing power is cheap and is rarely the bottleneck in the system. More frequently the coordination point — the database — is generally the bottleneck in any large system. Because of that reprocessing a bit to regenerate the user interface isn't generally as big an issue as mitigating the impact on the back end database server.

Cache Management

Once you've concluded whether you want to cache the output or the objects you're still not done, you still have to consider what items you're going to put into the cache and what priority you're going to give them should the cache manager need to make decisions about which objects need to get thrown out of the cache to make room for new items.

As a practical matter cache's can't be of infinite size. There's a limited amount of memory in a server, there's a limited amount of entries in a database before performance suffers, etc. Knowing that you have a fixed size cache to work with the question often comes up as to which objects should end up in cache, for how long, and which objects should be purged first if there are pressures to reduce the cache size.

Selecting objects to cache is really a balance between four factors: volatility, frequency of access, cost to recreate, and size in cache. The ideal candidate for caching is one that never changes, has a really high frequency of use and cost to recreate and is very small. Of course, real objects don't have all of these attributes. In the real world objects may be relatively volatile but don't cost much to recreate. (These objects may not even need to be cached at all let alone cached with a high priority.) Similarly, you may find that an invoice is hard to recreate but is relatively infrequently accessed and is fairly large in memory.

The greater the performance impact by caching the object the more important it is to keep in memory. In review, slowly changing (low volatility), highly accessed, hard to recreate, and small storage objects are the best to maintain in a cache. The more an object fits the characteristics the more important it is to keep in cache.

Summary

Caching is a useful tool but it has its own issues and limitations. Using caching means you have to consider when the cached data should expire and how to invalidate it. It also means a renewed concern for the amount of memory in use on the server and in the process. Caching can greatly improve performance but it can also lull you into a false sense of security. In some cases it can even make the performance worse. The next stop in our journey to improve performance is to evaluate ways to track down problems, and use bigger and better hardware to solve problems.