NCache™

Whitepaper

4 Ways to Improve ASP.NET Performance Under Peak Loads

By

Iqbal Khan

April 18, 2015

Table of Content

Introduction

ASP.NET is becoming very popular for developing web applications and many of these applications are high traffic in nature and serving millions of users. As a result, these applications have a large impact on business and are therefore very important.

Interestingly, the ASP.NET application architecture is very scalable at the application tier. And, the HTTP protocol is also stateless. Both of these mean that you can run your ASP.NET application in a load-balanced web farm where each HTTP request is routed to the most appropriate web server. This allows you to easily add more web servers to your web farm as your user traffic increases. And, this makes your ASP.NET application tier very scalable. But what do I mean by scalability here?

Scalability is essentially the ability to deliver high performance even under peak loads. So, if your ASP.NET application page response time is very fast with 10 users and it stays as fast with 100,000 users, then your ASP.NET application is scalable. But, if your ASP.NET response time slows down as you increase the number of users, then your application is not scalable.

The Problem: Scalability Bottlenecks

Despite a very scalable architecture at the application tier, ASP.NET applications today are facing major scalability bottlenecks. These bottlenecks are occurring in four different areas as following:

1. Application Database
2. ASP.NET Session State Storage
3. ASP.NET View State
4. ASP.NET Page Execution for Static Output

Let me explain each in more detail below.

Application Database

Application database like SQL Server or Oracle quickly becomes a scalability bottleneck as you increase transaction load. This happens because although you can scale up the database tier by purchasing a more powerful database server, you cannot scale out by adding more servers to the database tier. For example, it is very common to see 10-20 servers at the application tier, but you cannot do the same at database tier.

ASP.NET Session State Storage

Additionally, ASP.NET Session State needs to be stored somewhere. And, the out of box options provided by Microsoft are InProc, StateServer, and SqlServer. Unfortunately, all three options have major performance and scalability issues.

InProc and StateServer force you to use sticky sessions and send all HTTP requests on the same server where session was created. If you configure StateServer as a stand-alone server to avoid sticky sessions, then StateServer becomes a single point of failure and its performance also becomes a major issue. And, SqlServer stores ASP.NET sessions in SQL Server database as BLOBs. And, there are serious performance and scalability problems with this approach.

ASP.NET View State

ASP.NET View State is an encoded hidden string (often 100's of KB in size) that is sent to the user's browser as part of the HTTP response. The browser doesn't do anything with it and returns it back to the web server in case of an HTTP Post-Back. This slows down ASP.NET page response, puts more burden on web server Network Cards, and also consumes a lot of extra bandwidth. And, as you know bandwidth is not cheap.

ASP.NET Page Execution for Static Output

Finally, ASP.NET framework executes an ASP.NET page against a user request, even if the page output doesn't change from the previous request. This may be okay in low transaction environments. But in a high transaction environment where you're already stretching all the resources to their limits, this extra execution can become quite costly and a scalability bottleneck.

As the saying goes "the strength of any chain is only as strong as its weakest link". So, as long as there are scalability bottlenecks anywhere in ASP.NET application environment, the entire application slows down and even grinds to a halt.

And, ironically this happens under peak loads when you're doing highest levels of business activity. Therefore, the impact of any slowdown or a downtime is much more costly for your business.

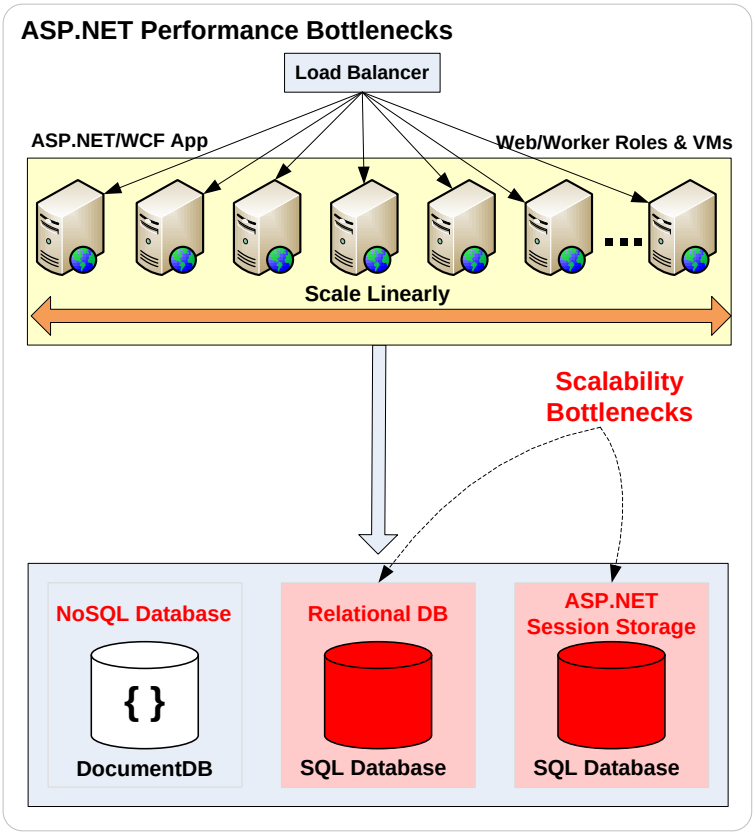

Figure 1: ASP.NET Facing Scalability Bottlenecks

NoSQL Database Not the Answer

NoSQL database movement started as a result of the scalability problems in relational databases. NoSQL database partitions the data onto multiple servers and allows you to scale-out just like the application tier.

But, NoSQL database requires you to abandon your relational database and put your data into a NoSQL database. And, this is easier said than done for a host of reasons and in fact not possible in a lot of cases.

NoSQL databases do not have the same data management and searching capability as relational databases and want you to store data in a totally different manner than relational. Additionally, the ecosystem surrounding relational databases is too strong to abandon for most businesses.

As a result, NoSQL databases are useful only when you're dealing with unstructured data. And, most ASP.NET applications are dealing with business data that is predominately structure and fit for relational databases. As a result, this data cannot be moved into a NoSQL database easily.

And, even those who end up using a NoSQL database do so for a small subset of their total data that can be considered unstructured. And, they use NoSQL database along with their existing relational database.

Therefore, majority of times you will need to live with your relational database and find another solution for your scalability problems. Fortunately, there is a very viable solution and I will discuss that below.

The Solution: In-Memory Distributed Cache

The solution to all the problems mentioned above is to use In-Memory Distributed Cache in your application deployment like NCache. NCache is an Open Source distributed cache for .NET that is extremely fast and linearly scalable. Think of it as an in-memory object store that is also distributed. Being in-memory makes it extremely fast and being distributed makes it linearly scalable.

The nice thing about an In-Memory Distributed Cache like NCache is that it does not ask you to stop using your existing relational database. You can use the cache on top of your relational database because the cache removes all of the relational database scalability bottlenecks.

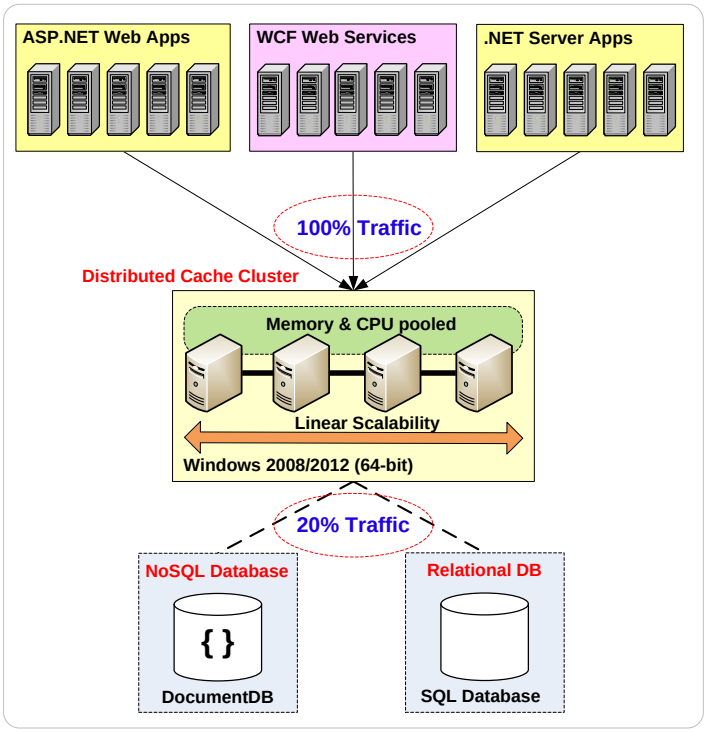

Figure 2: NCache Provides Linearly Scalability

So how is an In-Memory Distributed Cache like NCache more scalable than a relational database? Well, NCache forms a cluster of cache servers and pools together the CPU, memory and other resources from all these servers.

And, NCache allows you to add cache servers at runtime without either stopping the cache or the application. And, this enables you to linearly scale your application and handle extreme transaction loads. This is something you cannot do with your relational database.

In-Memory Distributed Cache like NCache scales linearly by allowing you to add cache servers to the cache cluster (the caching tier) at runtime. But, what type of performance numbers should you expect from a solution like NCache.

Below are NCache performance numbers. You can see full NCache performance benchmark here.

Cluster Size	Reads Per Second	Writes Per Second
2-node cluster	50,000	32,000
3-node cluster	74,000	48,000
4-node cluster	98,000	64,000
5-node cluster	122,000	80,000
6-node cluster	146,000	96,000

Figure 3: Performance Numbers for NCache

As you can see, an In-Memory Distributed Cache like NCache provides sub-millisecond performance for reads and writes and allows you to scale your transaction capacity linearly by simply adding more cache servers.

Let's now see how an In-Memory Distributed Cache like NCache solves various scalability bottlenecks mentioned above.

Application Data Caching

Application Data Caching enables you to remove your database bottlenecks. In-Memory Distributed Cache like NCache allows you to cache application data and reduce those expensive database trips. You can expect to divert 70-90% of database traffic to the In-Memory Distributed Cache. This reduces pressure on your database and allows it to perform faster and handle larger transaction loads without slowing down.

```
Customer Load(string customerId)
{
   // Key format: Customer:PK:1000
   string key = "Customers:CustomerID:" + customerId;

   Customer cust = (Customer) _cache[key];

   if (cust == null)
   {  // Item not in cache so load from db
       LoadCustomerFromDb(cust);

       // Add item to cache for future reference
       _cache.Insert(key, cust);
   }
   return cust;
}
```

Figure 4: Using In-Memory Distributed Cache for App Data Caching

Application data caching means you cache whatever application data you get from your relational database. This is usually in the form of domain objects (also called entities). Here is an example on how to use a distributed cache like NCache for application data caching.

ASP.NET Session State Caching

In-Memory Distributed Cache like NCache is also a great place to store your ASP.NET Session State. It is much faster and more scalable than all three options mentioned above (InProc, StateServer, and SqlServer). NCache is faster because it is in-memory and provides a key-value interface with the value being an "object" which an ASP.NET Session State is. And, it is scalable because it is a distributed cache.

And, NCache also intelligently replicates sessions through its rich caching topologies so even if a cache server goes down, there is no session data loss. This replication is needed because NCache provides an in-memory store and memory is violate storage.

NCache also speeds up your serialization of the ASP.NET Session State object that is required before it can be stored out-of-process. NCache does this by using its Dynamic Compact Serialization feature that is 10 times faster than regular .NET serialization. You can use this feature without making any code changes.

You can plug in NCache Session State Provider (SSP) module to your ASP.NET application by making some changes in your web.config file as shown below.

```
<system.web>
   ...
   <assemblies>
      <add assembly="Alachisoft.NCache.SessionStoreProvider, Version=4.3.0.0,
                  Culture=neutral, PublicKeyToken=CFF5926ED6A53769" />
   </assemblies>

   <sessionState    cookieless="false" regenerateExpiredSessionId="true"
                    mode="Custom" customProvider="NCacheSessionProvider"
              timeout="20">
      <providers>
         <add name="NCacheSessionProvider"
             type="Alachisoft.NCache.Web.SessionState.NSessionStoreProvider"
             useInProc="false" cacheName="myDistributedCache"
             enableLogs="false“ writeExceptionsToEventLog="false” />
      </providers>
   </sessionState>
   ...
</system.web>
```

Figure 5: Plug-in NCache as ASP.NET Session State Provider (SSP) in Web.Config

ASP.NET View State Caching

I have already described how ASP.NET View State is an encoded string sent by the web server to the user's browser which then returns it back to the web server in case of an HTTP Post Back. But, with the help of an In-Memory Distributed Cache like NCache, you can cache this ASP.NET View State on the server and only send a small unique ID in place of it.

NCache has implemented an ASP.NET View State caching module through a custom ASP.NET Page Adaptor and ASP.NET PageStatePersister. This way, NCache intercepts both HTTP request and response. At the response time, NCache removes the "value" portion of encoded string and caches it and instead puts a unique identifier (a cache key) in this "value".

Then, when the next HTTP request comes, it intercepts it again and replaces the unique identifier with the actual encoded string that it has put in the cache earlier. This way, the ASP.NET page doesn't notice anything different and uses the encoded string containing the View State the way it did before.

The example below shows an ASP.NET View State encoded string without caching and also what happens when caching is incorporated.

ASP.NET View State without Caching

```
<input id="__VIEWSTATE"
       type="hidden"
       name="__VIEWSTATE"
       value="/wEPDwUJNzg0MDMxMDA1D2QWAmYPZBYCZg9kFgQCAQ9kFgICBQ9kFgJmD2QWAgIBD
              xYCHhNQcm2aW91c0NvbnRyb2xNb2RlCymIAU1pY3Jvc29mdC5TaGFyZVBvaW50Lld
              lYkNvbnRyb2xzLlNQQ29udHJbE1vZDA1XzRlMjJfODM3Y19kOWQ1ZTc2YmY1M2IPD
              xYCHhNQcm2aW91c0NvbnRyb2xNb2RlCymIAU1pY3Jvc29mdC5TaGFyZVBvaW50Lld
              lYkNvbnRyb2xzLlNQQ29udHJbE1vZDA1XzRlMjJfODM3Y19kOWQ1ZTc2YmY1M2IPD
              ... ==" />
```

ASP.NET View State with Caching

```
<input id="__VIEWSTATE"
       type="hidden"
       name="__VIEWSTATE"
       value="vs:cf8c8d3927ad4c1a84da7f891bb89185" />
```

Figure 6: ASP.NET View State Encoded String with or without Caching

ASP.NET Output Cache for Static Page Output

ASP.NET provides an ASP.NET Output Cache Framework to address the issue of excessive page execution even when the page output doesn't change. This framework allows you to cache the output of either the entire page or some portions of the page so the next time this page is called, it will not be executed and instead its cached output will be display. Displaying an already cached output is much faster than executing the entire page again.

```
<caching>
  <outputCache defaultProvider ="NOutputCacheProvider">
    <providers>
      <add name="NOutputCacheProvider"
       type="Alachisoft.NCache.OutputCacheProvider.NOutputCacheProvider,
             Alachisoft.NCache.OutputCacheProvider, Version=x.x.x.x,
             Culture=neutral, PublicKeyToken=1448e8d1123e9096"

             cacheName="myDistributedCache" exceptionsEnabled="false"
             writeExceptionsToEventLog="false" enableLogs="true" />"
    </providers>
  </outputCache>
</caching>
```

Figure 7: Setting up ASP.NET Output Cache Provider for NCache in Web.Config

NCache has implemented an ASP.NET Output Cache Provider for .NET 4.0 or later versions. This allows you to plug in NCache seamlessly and without any programming effort. In case of NCache, this provider is for an In-Memory Distributed Cache that spans multiple servers. So, if your ASP.NET application is running in a load-balanced web

farm, the page output cached from server 1 is immediately available to all other servers in the web farm. Below is how you can plug in NCache as ASP.NET Output Cache Provider.

Distributed Cache Architecture

High traffic ASP.NET applications cannot afford to go down especially during peak hours. For these types of applications, there are three important architectural goals that a good In-Memory Distributed Cache like NCache provides. They are:

1. High availability
2. Linear scalability
3. Data replication and reliability

Let me explain each area below.

High Availability

One of the most important architectural goals of NCache is to achieve high availability and cache elasticity. And, it does that through the following architectural capabilities:

1. **Self-healing peer-to-peer cache cluster:** NCache builds a cluster of cache servers over TCP/IP. This cluster has a peer-to-peer architecture that means there are not master/slave nodes and no majority-rule clustering. Instead, each node is an equal peer. This enables NCache to handle situations where any node could go down and the cluster automatically adjusts itself and continues running, and there is no interruption for your application.
2. **Dynamic configuration:** This means you don't have to hard-code things in configuration files. This is because NCache propagates a lot of configuration information to cache clients (meaning your applications) at runtime. So, when you add a cache server at runtime, the cluster membership is automatically updated and the cluster informs all the cache clients about this change. There are a host of other configuration changes that are handled in the same fashion.
3. **Connection failover support:** This is a capability in which when a cache server goes down, the cache cluster and the cache clients are able to continue working without any interruption. In case of cache cluster, I've already discussed its self-healing quality that addresses this situation. In case of cache clients, this means the cache client continue working by interacting with other cache servers in the cluster.

Data Replication with Linear Scalability

Since In-Memory Distributed Cache like NCache uses memory as the store, it must provide data replication in order to ensure reliability. But, at the same time, it cannot compromise on linear scalability because that is the most important reason for using a distributed cache like NCache.

Here are some NCache Caching Topologies that help achieve both of these goals.

1. **Partitioned Cache:** NCache partitions the cache based on the number of cache servers and assigns one partition to each cache server. It also adjusts the number of partitions when you add or remove cache servers at runtime. Partitioning is the primary way of ensuring linear scalability because as you add more servers, this caching topology increases the overall storage size and also CPU processing power.
2. **Partitioned-Replica Cache:** In addition to partitioning, NCache also provides replicas for each partition. These replicas reside on different cache servers than the partition itself to ensure that if a cache server

goes down along with its partition, then the replica immediately becomes available. This way, data reliability is provided. By replicating each partition only once on another cache server, NCache achieves data reliability without compromising linear scalability.

3. **Client Cache (Near Cache):** Another very important capability of NCache is Client Cache. This is a local cache that sits on the cache client machine (namely your web or app server) and can even be InProc (meaning it resides within your application process). This is essentially a cache on top of a cache and provides extreme performance gains along with increasing scalability of NCache itself because the traffic even to the caching tier drops.

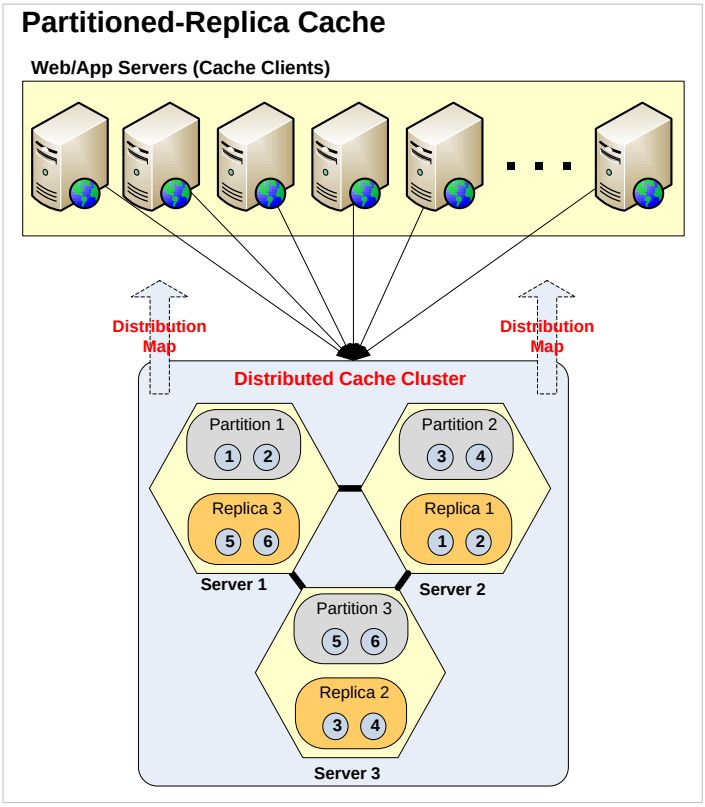

Figure 8: Partition-Replica Caching Topology of NCache

As you can see, Partitioned-Replica Cache puts one partition and one replica on each cache server. And, it ensures that the replica is always on a different cache server for reliability purposes.

Conclusion

I have tried to highlight the most common performance and scalability bottlenecks that ASP.NET applications face today and show you how to overcome these by using an In-Memory Distributed Cache like NCache. NCache is an Open Source distributed cache for .NET and Java applications. So, you can use it without any restrictions. You can find out more about NCache at the following links.

NCache Details	Edition Comparison	Download NCache
NCache vs AppFabric	NCache vs Redis	NCache vs Memcached

