**Varnish, Memcached, Redis, and HTTP Caching for Increased Web App Performance**

The web would not be the same without caching. It is an essential element to every web platform and whether you realize it or not we all benefit from multiple forms of caching every time we browse the web. Just as one does not run to the grocery store every time they need milk, software should not go through an entire data request cycle every time it needs data. This is the idea of caching in a nutshell - store data in fast access memory that will likely be needed soon. Let’s examine the various forms of caching and how caching can be best utilized to increase web performance.

*If you have not read the preceding post,*[*Introduction to Large Scale, High-Availability Web Architectures*](http://austincorso.com/2015/12/01/large-scale-high-availability-web-architectures.html)*, please be sure to do so.*

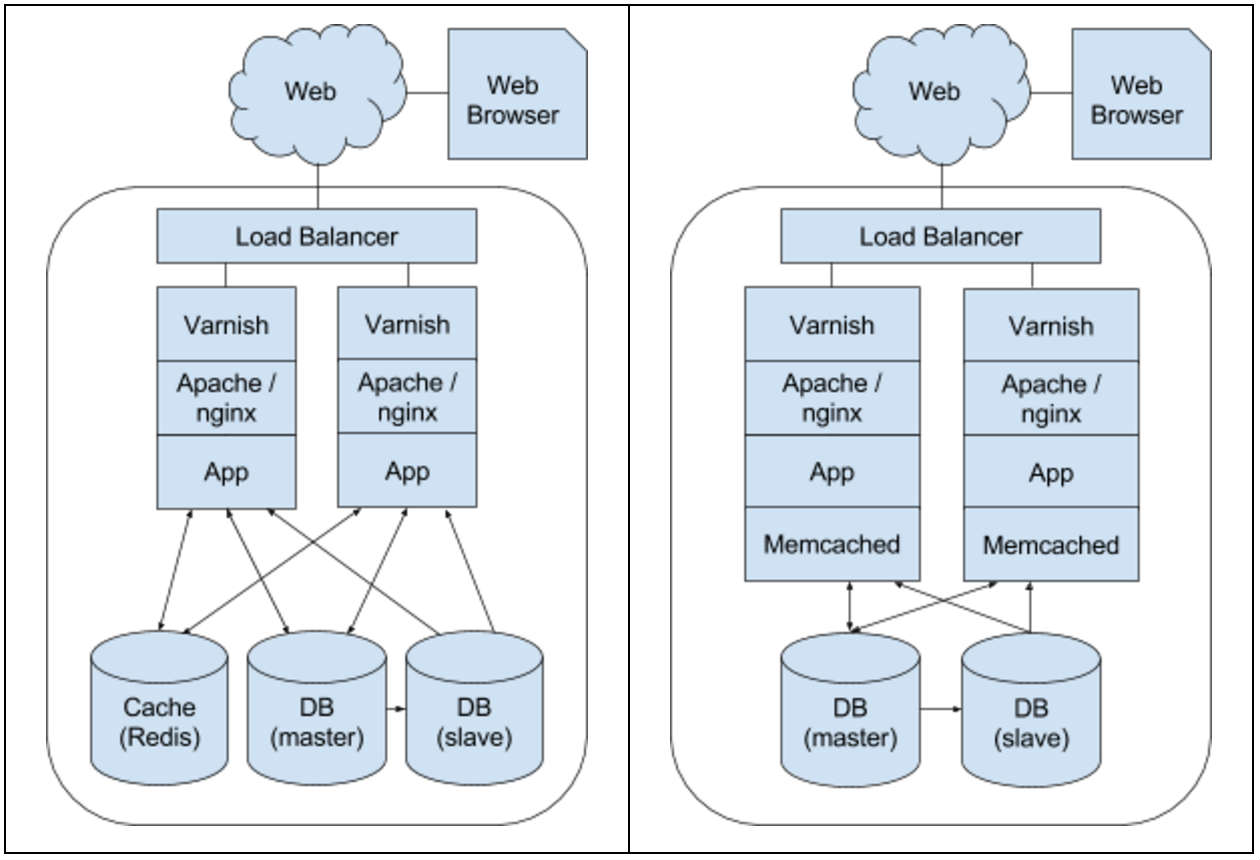
Web caches come in many forms: within one’s web browser, through content delivery networks (CDNs), by internet service providers (ISPs) utilizing DNS caching, and within web platforms through proxy caching and database caching, to name a few.

### **Caching through the Browser and CDNs**

Let’s discuss browser and CDN caching first. These caches remember resources that have already been loaded so upon subsequent page requests, data such as images, CSS files, etc. can be reused and not re-downloaded. This allows a reduction in the quantity of data downloaded and speeds up navigation. Web developers can instruct both browsers and content delivery networks as to how long assets should be cached for through an Apache or Nginx configuration file such as an .htaccess file. This configuration file performs several responsibilities, one of which is setting caching headers. Said caching headers include the “Expires”, “Max-age”, “Last-modified”, and “E-tag” headers, together allowing for fine control over how long assets shall be cached for. Each provides different capabilities - the expires tag, a timestamp when an asset expires; the max-age tag, the number of seconds to cache an asset for; the last-modified tag, a timestamp when an asset was last modified; and the e-tag, a unique identifier for asset comparison. Often caching configuration will be done on a per-asset-type basis, though more granular control is available. For example, assets not expected to change often, such as a logo, may warrant a large max-age caching header; assets changing on a set period of time, such as Google’s daily-changing cartoon, may warrant an expires tag; and assets changing frequently and sporadically, such as a user avatar may warrant an e-tag or last-modified caching header.

I won’t cover too much on DNS caching, only that the domain name system (DNS) matches web addresses with IP addresses and to prevent the need to look these up on every page request, ISP’s utilize DNS caching.

Now let’s re-examine our web architecture to see how one may utilize caching to improve its performance.



In the above diagrams, you will see several new additions with our new caching mechanisms in place, most notably proxy caching through Varnish and database caching through either Memcached or Redis.

### **Proxy Caching with Varnish**

Varnish is a caching HTTP reverse proxy or HTTP accelerator that sits in front of one’s application layer, caching rendered responses for re-use later. When Varnish receives requests from the web it checks whether that response was requested recently and whether a valid response has been stored in its cache, if so it is able to bypass the backend and immediately return the stored rendered response. When this happens, responses are typically delivered within a matter of microseconds! This not only speeds up response time, but also reduces the load on one’s application and database tiers. One can configure which assets are cached, again, through HTTP headers, this time the “Cache-Control” header.

Such a mechanism is tremendously beneficial for highly-dynamic platforms, such as social networks and content management systems (CMS’s) like Drupal and Word press. Nowadays you would be hard pressed to find a large-scale web platform not utilizing Varnish, with notable companies such as Twitter, Facebook, and Wikipedia all utilizing it.

### **Caching with Memcached**

Making queries to a relational database such as MySql, Oracle, or PostgreSql is typically the bottleneck in terms of speed and performance within a web platform. It requires data to be read from disk and then queried upon. Though most production-ready databases will already utilize advanced caching techniques for datasets, indexes, and query result sets; an in-memory caching mechanism for storing objects separate from the database grants tremendous performance gains. Two very popular such tools are Memcached and Redis.

Memcached had quickly risen in popularity due to its ease of setup and distributed nature. Typically, Memcached is configured with an instance on each webhead or within a bank of machines. You can think of each of these instances as buckets within the overarching Memcached hash table. A hash function then maps keys to buckets. Once in each bucket Memcached utilizes a second hash function to locate the key within said bucket. This allows Memcached to store and retrieve data in constant time O (1), while keeping instances independent of one another. Additional nodes can be added easily and nodes failing can have their access rerouted.

Setting up Memcached is easy. Upon installing through your favorite package manager, Memcached will be started automatically every time your server boots. To integrate it, one just needs to update each of their database calls with an additional call to Memcached. Write operations then write to the database and Memcached; Read operations are first performed within Memcached and only if the data is stale or non-existent is the database queried. Assuming one has abstracted their data access calls within their codebase this will be a relatively painless task.

### **Caching with Redis**

Redis is newer than Memcached, though has already gained mass adoption with large brands such as Twitter, GitHub, and Airbnb all utilizing it. Redis is like Memcached in that it too is an in-memory caching store, but has several additional benefits that has aided its adoption.

The first key differentiating feature Redis introduces is the ability to store non-string values such as lists, sets, and even hash tables. Along with these data types, Redis supports atomic operations such as intersections and unions, as well as sorting.

The second key feature Redis offers is persistence to disk. Though Redis is indeed an in-memory data store, Redis offers the ability to persist data to disk both through snapshotting (RDB) and maintaining a log of all operations performed (AOF). This is tremendously useful for several reasons. The obvious benefit is in the case of service failure, one’s cached data is not lost. A not so obvious benefit but vital nonetheless is that this allows one’s platform to avoid the overload of requests against the database when the service is restarted and the cache has no data.

In the past Redis has been typically implemented as a single instance, but newer releases (3.0+) have enabled distributed setups akin to Memcached. This allows Redis to accommodate much larger data sets through clustering. Multiple instances provide increased memory, increased computational capacity, and increased network capacity.

Whether one utilizes Memcached or Redis, either will provide a tremendous boost to a web platform’s performance and efficiency, storing objects in-memory and reducing load on one’s database tier.

Modern high-availability, large scale web platforms must utilize various forms of caching, several of which we touched on here briefly. Together these caching mechanisms will make one’s architecture significantly more scalable and allow one’s web platform to handle an increasing amount of traffic.

<http://austincorso.com/2015/12/11/varnish-memcached-redis-http-caching.html>