BASICS OF SCALING: CACHE EVERYTHING

I do a lot of work on websites that needs to scale fairly well, but I tend to use that mentality for every project. Part of scaling is performance, and the better your app performs (e.g. the more requests per second it can handle) the cheaper it is.

One very easy way to improve your application's performance is to add caching. If you don't currently have caching, you'll probably see a massive benefit, depending on your application architecture. There are many different types of caching, with varying degrees if difficulty to implement.

I've included some very surface level details on caching in this post. These posts aren't meant to be comprehensive tutorials, but to make you aware of various techniques used to scale your applications better.

HTTP Accelerators/Reverse Proxies

You have HTTP accelerator style caching such as Varnish, which tend to be very easy to setup and can give you huge performance improvements if you have a lot of cacheable content (e.g. on a WordPress blog). Varnish is one of my goto resources for fast and easy scaling. Instead of having Apache or Nginx as your

public facing server component, you'd have Varnish. You set Varnish to listen on port 80, and you'd change Apache/Nginx/Whatever to listen on an alternate port like 8080.

You configure Varnish with Apache/Nginx as a backend, where it will direct requests that aren't currently cached. That's all you need for a very basic setup, but it's likely you'll want to customize Varnish's caching logic to be a bit more permissive (for example, Varnish won't cache a page if there are cookies in the request). I'll leave the detailed setup instructions for another blog post, since it's a very large topic (you can search for some tutorials to get started).

Varnish can be the key to saving your website from crashing during large traffic spikes (like if you get posted to Reddit or another popular site). When properly configured, Varnish can continue serving up cached pages even if the backend is offline. Also, Varnish stores it's cache in memory and is highly optimized at what it does (for example, it does very few system calls to avoid frequent context switching, they've implemented most system calls themselves, in user space).

Page Caches

Then you have page caching techniques within your application. WordPress, Symfony, and Rails for example, all have mechanisms to cache the output of pages

as static files and use those, skipping much of the slow parts of your application (e.g. database queries). Some systems even allow you to store page caches in memory instead of on the filesystem, using a builtin mechanism or an external application like memcached.

Page caching techniques vary wildly depending on your application or framework, so I won't go too deep into it here.

The gist of it is that if you have pages filled with content that doesn't change super frequently and isn't dynamic for every request, you can typically render it out to a static file and serve that until the content changes. This avoids the overhead of database calls and complicated application logic, but still has high overhead as most dynamic languages like Ruby/Python/PHP have a lot of overhead for every request. It's at least an order of magnitude slower for these languages to serve a cached resource than Varnish.

Also worth noting is that many systems including Varnish support something called Edge-Side Includes (similarly there are Server-Side Includes). If the majority of your page is static, but there are one or two dynamic parts (e.g. user details), you can statically cache the page and use a ESI or SSI to include the dynamic parts when the page is served. This tends to be higher performance than

re-rendering the entire page (again, this varies based on your individual application).

Another technique to improve the cache-ability of your pages for both page caching and reverse proxies is to use AJAX to fill in dynamic data after the page loads. One of my websites has every page cached, so I use AJAX to populate user data. Once the page loads, I make a request to a JSON endpoint (not cached) which returns the user's current info. I then use JavaScript to place this in the page.

In-Memory Caching

Another very common technique for caching is using an in-memory data store like Redis or Memcached, located on the same machine as the webserver (sometimes this isn't the case though). You use these in-memory data stores to temporarily store data that is very slow to compute. Then in your application, you'd check the memory store first, and if it's not there, you fall back to the actual code. After computing the results, you should then add it to the memory store.

Examples of using this are for caching database calls (example: you have a settings table that you fetch every request. It's far faster to store it in memory than to do a database call every request), or slow computations. You shouldn't use in-memory

data stores for any information you want to keep, as they are considered volatile storage (nothing persists after a restart).

Some programming languages also support shared memory built right into the language. These systems allow you to cache data in a shared block of memory and access it from any process. However, I recommend using memcached or something similar instead, as they tend to be more powerful and well optimized.

Cache Busting

Cache busting is the action of clearing/deleting/expiring a cache item. It's commonly used when a resource has changed (for example, my blog cache busts Varnish whenever I write a new post). It's commonly a source of confusion or issues when caching is newly introduced (why can't I see my changes?). Most frameworks have libraries for dealing with this, but if you're going with a homegrown approach, it's important to consider this. You may want to hook into your models to clear relevant caches for example.

Closing Notes

Caching is used heavily by every large website (e.g.Facebook/Google/Twitter/etc). Facebook operates one of the largest memcached clusters in the world, consisting of hundreds of terabytes of RAM[1][2]. These websites save millions of dollars a year in server resources by employing a multitude of caching techniques.

Another lesson that is often learned the hard way, caching can be a nightmare if you don't design your application with caching in mind from the very beginning. Adding caching in after the fact will cause many issues (e.g. resources not being cache busted automatically when a resource is changed). Naming is the other very important part of caching. It pays off to think of a good, scalable naming scheme before you get started.

Source: http://brandonwamboldt.ca/basics-of-scaling-cache-everything-1495/