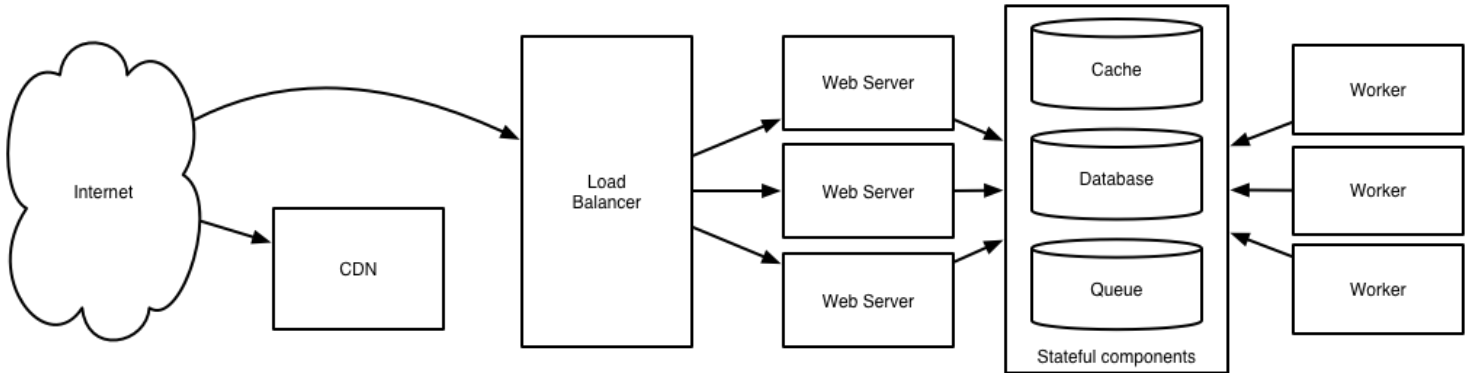


Introduction to architecting systems for scale.

April 4, 2011.

Few computer science or software development programs attempt to teach the building blocks of scalable systems. Instead, system architecture is usually picked up on the job by working through the pain of a growing product or by working with engineers who have already learned through that suffering process.

In this post I'll attempt to document some of the scalability architecture lessons I've learned while working on systems at Yahoo! and Digg.

I've attempted to maintain a color convention for diagrams:

- *green* is an external request from an external client (an HTTP request from a browser, etc),
- *blue* is your code running in some container (a Django app running on mod_wsgi, a Python script listening to RabbitMQ, etc), and
- *red* is a piece of infrastructure (MySQL, Redis, RabbitMQ, etc).

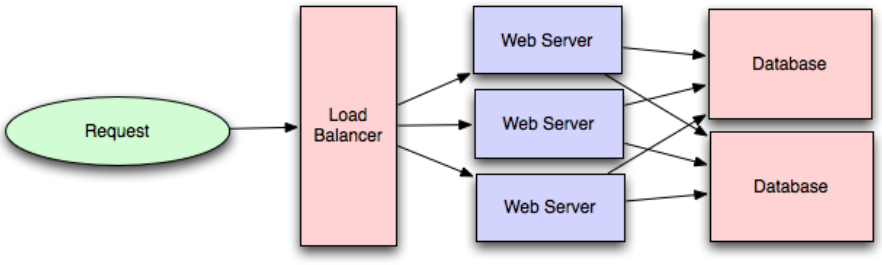
Load balancing

The ideal system increases capacity linearly with adding hardware. In such a system, if you have one machine and add another, your capacity would double. If you had three and you add another, your capacity would increase by 33%. Let's call this *horizontal scalability*.

On the failure side, an ideal system isn't disrupted by the loss of a server. Losing a server should simply decrease system capacity by the same amount it increased overall capacity when it was added. Let's call this *redundancy*.

Both horizontal scalability and redundancy are usually achieved via load balancing.

(This article won't address *vertical scalability*, as it is usually an undesirable property for a large system, as there is inevitably a point where it becomes cheaper to add capacity in the form on additional machines rather than additional resources of one machine, and redundancy and vertical scaling can be at odds with one-another.)



Load balancing is the process of spreading requests across multiple resources according to some metric (random, round-robin, random with weighting for machine capacity, etc) and their current status (available for requests, not responding, elevated error rate, etc).

Load needs to be balanced between user requests and your web servers, but must also be balanced at every stage to achieve full scalability and redundancy for your system. A moderately large system may balance load at three layers:

- user to your web servers,
- web servers to an internal platform layer,
- internal platform layer to your database.

There are a number of ways to implement load balancing.

Smart clients

Adding load-balancing functionality into your database (cache, service, etc) client is usually an attractive solution for the developer. Is it attractive because it is the simplest solution? Usually, no. Is it seductive because it is the most robust? Sadly, no. Is it alluring because it'll be easy to reuse?

Tragically, no.

Developers lean towards smart clients because they are developers, and so they are used to writing software to solve their problems, and smart clients are software.

With that caveat in mind, what is a smart client? It is a client which takes a pool of service hosts and balances load across them, detects downed hosts and avoids sending requests their way (they also have to detect recovered hosts, deal with adding new hosts, etc, making them fun to get working decently and a terror to setup).

Hardware load balancers

The most expensive–but very high performance–solution to load balancing is to buy a dedicated hardware load balancer (something like a Citrix NetScaler). While they can solve a remarkable range of problems, hardware solutions are remarkably expensive, and they are also "non-trivial" to configure.

As such, generally even large companies with substantial budgets will often avoid using dedicated hardware for all their load-balancing needs; instead they use them only as the first point of contact from user requests to their infrastructure, and use other mechanisms (smart clients or the hybrid approach discussed in the next section) for load-balancing for traffic within their network.

Software load balancers

If you want to avoid the pain of creating a smart client, and purchasing dedicated hardware is excessive, then the universe has been kind enough to provide a hybrid: software load-balancers.

HAProxy is a great example of this approach. It runs locally on each of your boxes, and each service you want to load-balance has a locally bound port. For example, you might have your platform machines accessible via `localhost:9000`, your database read-pool at `localhost:9001` and your database write-pool at `localhost:9002`. HAProxy manages healthchecks and will remove and return machines to those pools according to your configuration, as well as balancing across all the machines in those pools as well.

For most systems, I'd recommend starting with a software load balancer and moving to smart clients or hardware load balancing only with deliberate need.

Caching

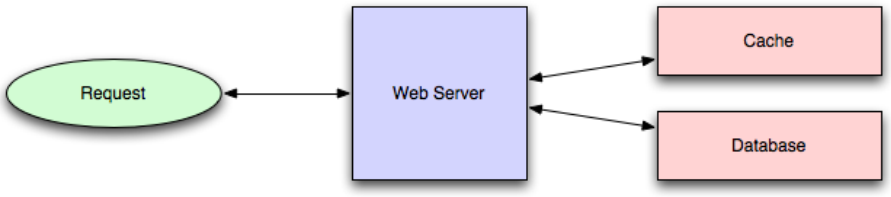
Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible.

Caching consists of: precalculating results (e.g. the number of visits from each referring domain for the previous day), pre-generating expensive indexes (e.g. suggested stories based on a user's click history), and storing copies of frequently accessed data in a faster backend (e.g. Memcache instead of PostgreSQL.

In practice, caching is important earlier in the development process than load-balancing, and starting with a consistent caching strategy will save you time later on. It also ensures you don't optimize access patterns which can't be replicated with your caching mechanism or access patterns where performance becomes unimportant after the addition of caching (I've found that many heavily optimized Cassandra applications are a challenge to cleanly add caching to if/when the database's caching strategy can't be applied to your access patterns, as the datamodel is generally inconsistent between the Cassandra and your cache).

Application vs. database caching

There are two primary approaches to caching: application caching and database caching (most systems rely heavily on both).

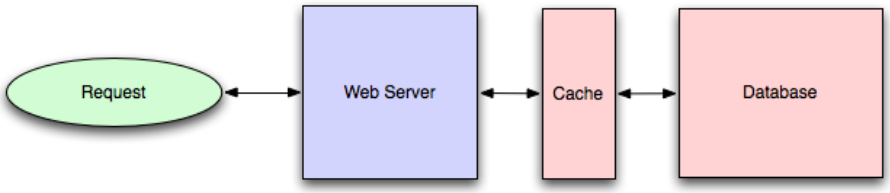


Application caching requires explicit integration in the application code itself. Usually it will check if a value is in the cache; if not, retrieve the value from the database; then write that value into the cache (this value is especially common if you are using a cache which observes the least recently used caching algorithm). The code typically looks like (specifically this is a *read-through cache*, as it reads the value from the database into the cache if it is missing from the cache):

```
key = "user.%s" % user_id
user_blob = memcache.get(key)
if user_blob is None:
    user = mysql.query("SELECT * FROM users WHERE user_id=\"%s\"", user_id)
    if user:
        memcache.set(key, json.dumps(user))
    return user
```

```
else:
    return json.loads(user_blob)
```

The other side of the coin is database caching.



When you flip your database on, you're going to get some level of default configuration which will provide some degree of caching and performance. Those initial settings will be optimized for a generic usecase, and by tweaking them to your system's access patterns you can generally squeeze a great deal of performance improvement.

The beauty of database caching is that your application code gets faster "for free", and a talented DBA or operational engineer can uncover quite a bit of performance without your code changing a whit (my colleague Rob Coli spent some time recently optimizing our configuration for Cassandra row caches, and was succcessful to the extent that he spent a week harassing us with graphs showing the I/O load dropping dramatically and request latencies improving substantially as well).

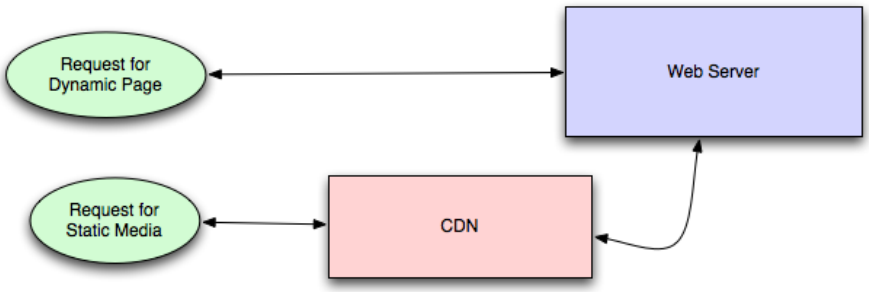
In-memory caches

The most potent–in terms of raw performance–caches you'll encounter are those which store their entire set of data in memory. Memcached and Redis are both examples of in-memory caches (caveat: Redis can be configured to store some data to disk). This is because accesses to RAM are orders of magnitude faster than those to disk.

On the other hand, you'll generally have far less RAM available than disk space, so you'll need a strategy for only keeping the hot subset of your data in your memory cache. The most straightforward strategy is least recently used, and is employed by Memcache (and Redis as of 2.2 can be configured to employ it as well). LRU works by evicting less commonly used data in preference of more frequently used data, and is almost always an appropriate caching strategy.

Content distribution networks

A particular kind of cache (some might argue with this usage of the term, but I find it fitting) which comes into play for sites serving large amounts of static media is the *content distribution network.*



CDNs take the burden of serving static media off of your application servers (which are typically optimzed for serving dynamic pages rather than static media), and provide geographic distribution. Overall, your static assets will load more quickly and with less strain on your servers (but a new strain of business expense).

In a typical CDN setup, a request will first ask your CDN for a piece of static media, the CDN will serve that content if it has it locally available (HTTP headers are used for configuring how the CDN caches a given piece of content). If it isn't available, the CDN will query your servers for the file and then cache it locally and serve it to the requesting user (in this configuration they are acting as a read-through cache).

If your site isn't yet large enough to merit its own CDN, you can ease a future transition by serving your static media off a separate subdomain (e.g. `static.example.com`) using a lightweight HTTP server like Nginx, and cutover the DNS from your servers to a CDN at a later date.

Cache invalidation

While caching is fantastic, it does require you to maintain consistency between your caches and the source of truth (i.e. your database), at risk of truly bizarre applicaiton behavior.

Solving this problem is known as *cache invalidation*.

If you're dealing with a single datacenter, it tends to be a straightforward problem, but it's easy to introduce errors if you have multiple codepaths writing to your database and cache (which is almost always going to happen if you don't go into writing the application with a caching strategy already in mind). At a high level, the solution is: each time a value changes, write the new value into the cache (this is called a *write-through* cache) or simply delete the current value from the cache and allow a read-through cache to populate it later (choosing between read and write through

caches depends on your application's details, but generally I prefer write-through caches as they reduce likelihood of a stampede on your backend database).

Invalidation becomes meaningfully more challenging for scenarios involving fuzzy queries (e.g if you are trying to add application level caching in-front of a full-text search engine like SOLR), or modifications to unknown number of elements (e.g. deleting all objects created more than a week ago).

In those scenarios you have to consider relying fully on database caching, adding aggressive expirations to the cached data, or reworking your application's logic to avoid the issue (e.g. instead of `DELETE FROM a WHERE...`, retrieve all the items which match the criteria, invalidate the corresponding cache rows and then delete the rows by their primary key explicitly).

Off-line processing

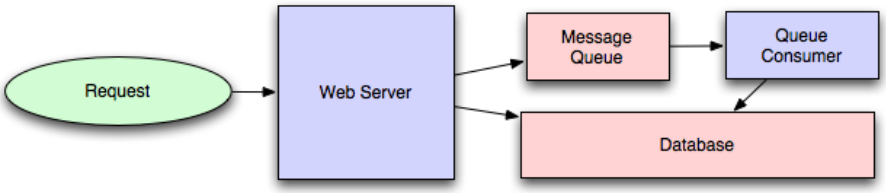
As a system grows more complex, it is almost always necessary to perform processing which can't be performed in-line with a client's request either because it is creates unacceptable latency (e.g. you want to want to propagate a user's action across a social graph) or it because it needs to occur periodically (e.g. want to create daily rollups of analytics).

Message queues

For processing you'd like to perform inline with a request but is too slow, the easiest solution is to create a message queue (for example, RabbitMQ). Message queues allow your web applications to quickly publish messages to the queue, and have other consumers processes perform the processing outside the scope and timeline of the client request.

Dividing work between off-line work handled by a consumer and in-line work done by the web application depends entirely on the interface you are exposing to your users. Generally you'll either:

1. perform almost no work in the consumer (merely scheduling a task) and inform your user that the task will occur offline, usually with a polling mechanism to update the interface once the task is complete (for example, provisioning a new VM on Slicehost follows this pattern), or
2. perform enough work in-line to make it appear to the user that the task has completed, and tie up hanging ends afterwards (posting a message on Twitter or Facebook likely follow this pattern by updating the tweet/message in your timeline but updating your followers' timelines out of band; it's simple isn't feasible to update all the followers for a Scobleizer in real-time).



Message queues have another benefit, which is that they allow you to create a separate machine pool for performing off-line processing rather than burdening your web application servers. This allows you to target increases in resources to your current performance or throughput bottleneck rather than uniformly increasing resources across the bottleneck and non-bottleneck systems.

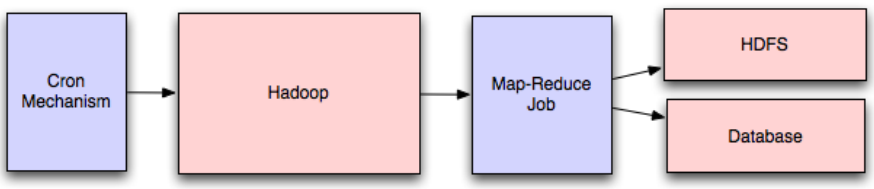
Scheduling periodic tasks

Almost all large systems require daily or hourly tasks, but unfortunately this seems to still be a problem waiting for a widely accepted solution which easily supports redundancy. In the meantime you're probably still stuck with cron, but you could use the cronjobs to publish messages to a consumer, which would mean that the cron machine is only responsible for scheduling rather than needing to perform all the processing.

Does anyone know of recognized tools which solve this problem? I've seen many homebrew systems, but nothing clean and reusable. Sure, you can store the cronjobs in a Puppet config for a machine, which makes recovering from losing that machine easy, but it would still require a manual recovery, which is likely acceptable but not perfect.

Map-reduce

If your large scale application is dealing with a large quantity of data, at some point you're likely to add support for map-reduce, probably using Hadoop, and maybe Hive or HBase.

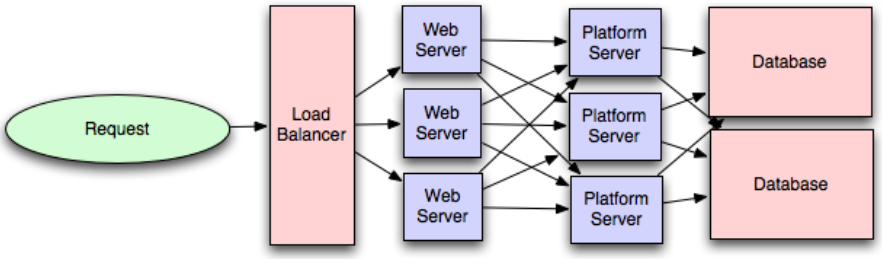


Adding a map-reduce layer makes it possible to perform data and/or processing intensive operations in a reasonable amount of time. You might use it for calculating suggested users in a social graph, or for generating analytics reports.

For sufficiently small systems you can often get away with adhoc queries on a SQL database, but that approach may not scale up trivially once the quantity of data stored or write-load requires sharding your database, and will usually require dedicated slaves for the purpose of performing these queries (at which point, maybe you'd rather use a system designed for analyzing large quantities of data, rather than fighting your database).

Platform layer

Most applications start out with a web application communicating directly with a database. This approach tends to be sufficient for most applications, but there are some compelling reasons for adding a platform layer, such that your web applications communicate with a platform layer which in turn communicates with your databases.



First, separating the platform and web application allow you to scale the pieces independently. If you add a new API, you can add platform servers without adding unnecessary capacity for your web application tier. (Generally, specializing your servers' role opens up an additional level of configuration optimization which isn't available for general purpose machines; your database machine will usually have a high I/O load and will benefit from a solid-state drive, but your well-configured application server probably isn't reading from disk at all during normal operation, but might benefit from more CPU.)

Second, adding a platform layer can be a way to reuse your infrastructure for multiple products or interfaces (a web application, an API, an iPhone app, etc) without writing too much redundant boilerplate code for dealing with caches, databases, etc.

Third, a sometimes underappreciated aspect of platform layers is that they make it easier to scale an organization. At their best, a platform exposes a crisp product-agnostic interface which masks implementation details. If done well, this allows multiple independent teams to develop utilizing the platform's capabilities, as well as another team implementing/optimizing the platform itself.

I had intended to go into moderate detail on handling multiple data-centers, but that topic truly deserves its own post, so I'll only mention that cache invalidation and data replication/consistency become rather interesting problems at that stage.

I'm sure I've made some controversial statements in this post, which I hope the dear reader will argue with such that we can both learn a bit. Thanks for reading!