Introduction to architecting systems for scale.

April 4, 2011. Filed under [infrastructure](https://lethain.com/tags/infrastructure/)[architecture](https://lethain.com/tags/architecture/)

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](http://engineering.twitter.com/2010/06/perfect-stormof-whales.html) 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!](http://yahoo.com/) and [Digg](http://digg.com/).

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

1. *green* is an external request from an external client (an HTTP request from a browser, etc),
2. *blue* is your code running in some container (a Django app running on [mod\_wsgi](http://code.google.com/p/modwsgi/), a Python script listening to [RabbitMQ](http://www.rabbitmq.com/), etc), and
3. *red* is a piece of infrastructure (MySQL, [Redis](http://redis.io/), 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:

1. user to your web servers,
2. web servers to an internal platform layer,
3. 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](http://www.citrix.com/English/ps2/products/product.asp?contentID=21679)). 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](http://haproxy.1wt.eu/) 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](http://memcached.org/) instead of [PostgreSQL](http://www.postgresql.org/).

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](http://cassandra.apache.org/) 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](http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used)). 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](http://memcached.org/) and [Redis](http://redis.io/) 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](http://en.wikipedia.org/wiki/RAM_disk) 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](http://en.wikipedia.org/wiki/Cache_algorithms#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](http://nginx.org/), 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](http://lucene.apache.org/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](http://www.rabbitmq.com/)). 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](http://twitter.com/#!/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](http://en.wikipedia.org/wiki/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](http://www.puppetlabs.com/) 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](http://en.wikipedia.org/wiki/MapReduce), probably using [Hadoop](http://hadoop.apache.org/), and maybe [Hive](http://hive.apache.org/) or [HBase](http://hbase.apache.org/).

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.

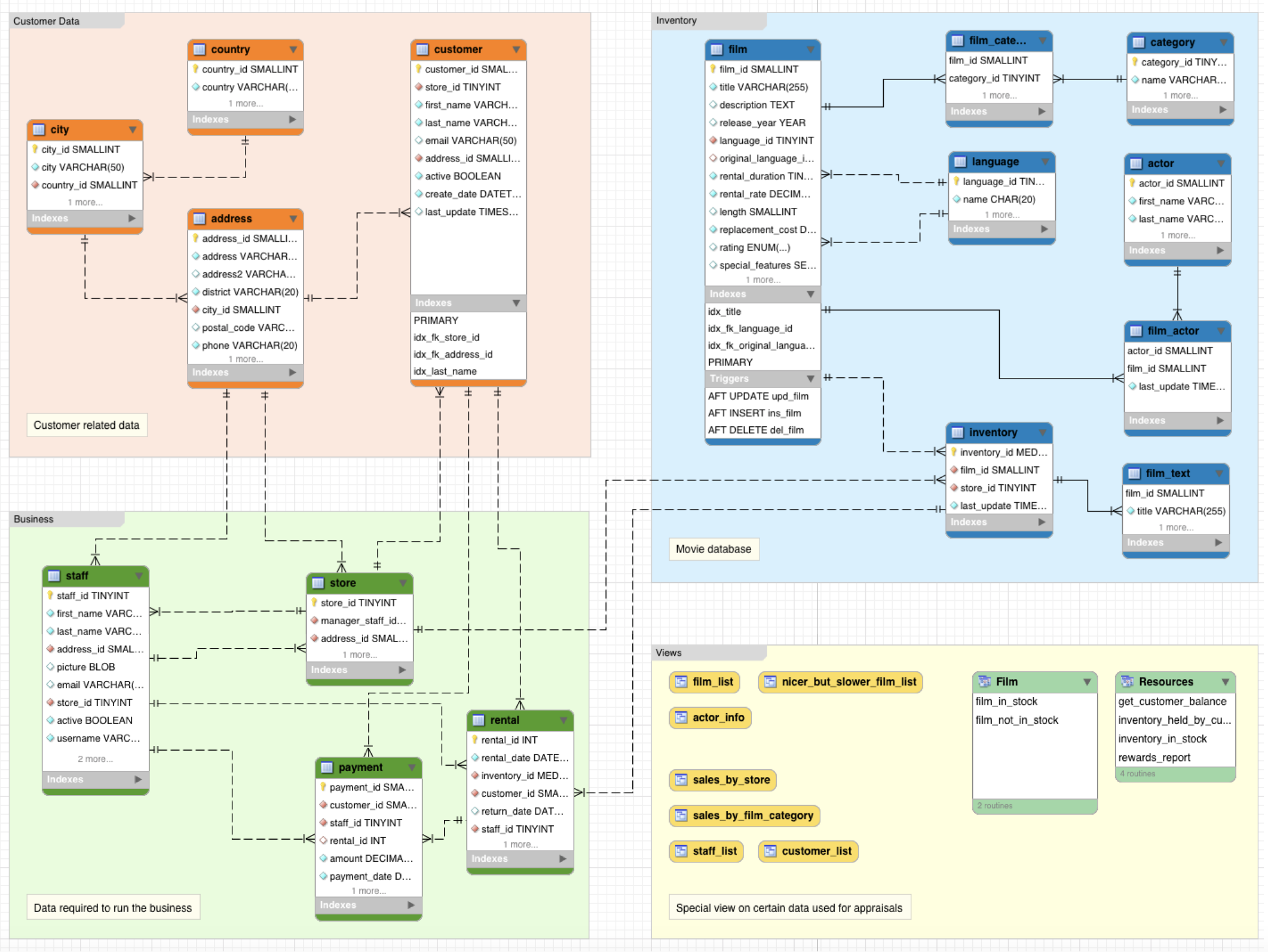
**EMG 549 Software Development and Architecture Homework 1**

**Question 0. Working with the Sakila MySQL database**

In this exercise, we will be working with the Sakila MySQL database. It can be downloaded by following the instructions at [http://dev.mysql.com/doc/sakila/en/sakilaD installation.html](http://dev.mysql.com/doc/sakila/en/sakilaD%20installation.html)

The database Files are downloaded at [https://www.dropbox.com/s/803itqyqzw8w174/ sakilaDdb.zip?dl=0](https://www.dropbox.com/s/803itqyqzw8w174/%20sakilaDdb.zip?dl=0)

Download the Files and follow the instructions completely.

Examine the structure of the tables by looking at MySQL workbench or the included PDF File. Following image is enlarged at [https://www.dropbox.com/s/guc1zu70u0dn7bt/ Untitled 2.png?dl=0](https://www.dropbox.com/s/guc1zu70u0dn7bt/%20Untitled%202.png?dl=0)

**Give SQL queries to answer the following questions and Run these queries print answers:**

1. How many distinct countries are there? 2. Find out the top 5 countries with most number of clients. 3. What are the names of all the languages in the database (sorted alphabetically)?

4. Return the full names (First and last) of actors with “SON” in their last name, ordered by their First name.

Hint: To search for a text in a Field use the Filter: WHERE First\_name LIKE ‘%son%’ ;

This will search for “SON” anywhere in the Field. 5. Create a list of Films and their corresponding categories. 6. Create a list of categories and the number of Films for each category. 7. Create a list of actors and the number of movies by each actor. 8. List the Film id and titles of those Films that are not in inventory. 9. Find a list of customers who have not rented a movie yet. 10. Find the number of **English** Films having category of ‘**Documentary’**.

**Question 1. Read the following Case Study on Intro to System Architecture and Answer following questions:**

1. What is web server? Give example name of a web server? Why is it used?
2. What is message queuing system? Where is it used? What are the features of Messaging system?
3. What is load balancer? Difference between Hardware and Software load balancer?
4. What is caching? Different types of caching? Why is caching needed? What happens when caching is not present?
5. What is platform as service , infrastructure as service and software as service?
6. What is CDN? Please explain in detail. Where is CDN used in apps you use on daily basis on your smart phone or TV or Computer?
7. What is Map Reduce? What is distributed computing? Explain data gravity?

**Introduction to architecting systems for scale.**

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](http://engineering.twitter.com/2010/06/perfect-stormof-whales.html) 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!](http://yahoo.com/) and [Digg](http://digg.com/).

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](http://code.google.com/p/modwsgi/), a Python script listening to [RabbitMQ](http://www.rabbitmq.com/), etc), and
* *red* is a piece of infrastructure (MySQL, [Redis](http://redis.io/), RabbitMQ, etc).

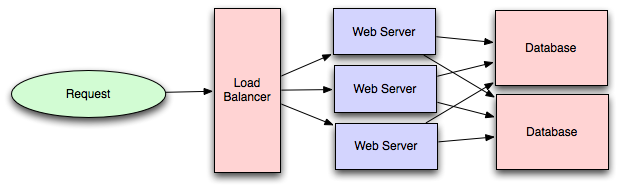
Load balancing

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](http://www.citrix.com/English/ps2/products/product.asp?contentID=21679)). 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](http://haproxy.1wt.eu/) 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**

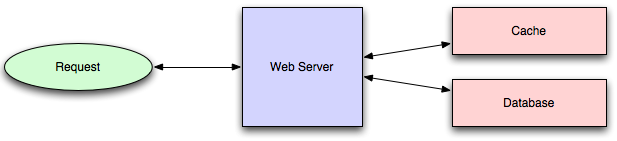
Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible.

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](http://memcached.org/) instead of [PostgreSQL](http://www.postgresql.org/).

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](http://cassandra.apache.org/) 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](http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used)). 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:

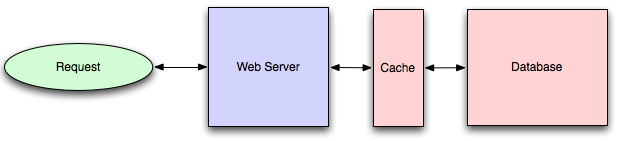
memcache.set(key, json.dumps(user))

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).

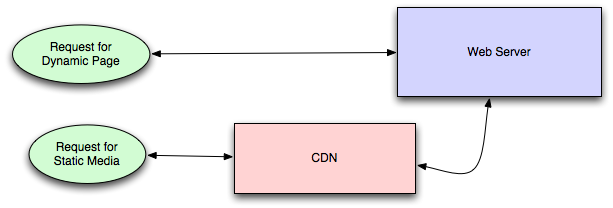
**In-memory caches**

The most potent–in terms of raw performance–caches you'll encounter are those which store their entire set of data in memory. [Memcached](http://memcached.org/) and [Redis](http://redis.io/) 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](http://en.wikipedia.org/wiki/RAM_disk) 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](http://en.wikipedia.org/wiki/Cache_algorithms#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](http://nginx.org/), 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](http://lucene.apache.org/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

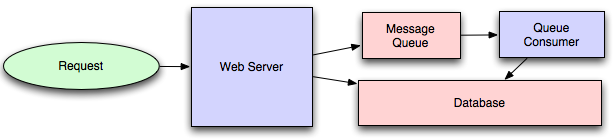
As a system grows more complex, it is almost always necessary to perform processing which can't be performed in-line with a client's request either because it is creates unacceptable latency (e.g. you want to want to propagate a user's action across a social graph) or it because it needs to occur periodically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

* 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
* 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](http://twitter.com/#!/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.

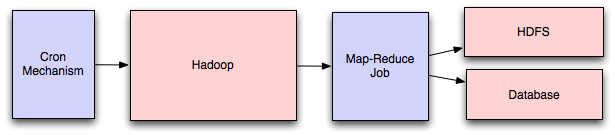
**Scheduling periodic tasks**

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](http://en.wikipedia.org/wiki/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](http://www.puppetlabs.com/) 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](http://en.wikipedia.org/wiki/MapReduce), probably using [Hadoop](http://hadoop.apache.org/), and maybe [Hive](http://hive.apache.org/) or [HBase](http://hbase.apache.org/).

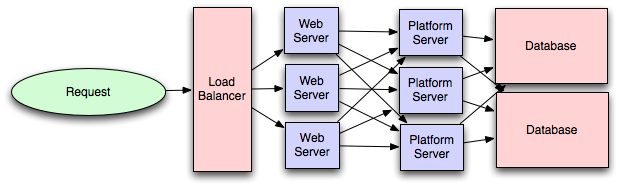


Adding a map-reduce layer makes it possible to perform data and/or processing intensive operations in a reasonable amount of time. You might use it for calculating suggested users in a social graph, or for generating analytics reports.

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!

**Question 2. Read the following case study on designing News Feed System like Facebook and answer following questions:**

1. What is news feed system and why is it important business like Facebook?
2. Please describe your understanding of Redis and Cassandra distributed database systems. Explain purpose and salient features of each database system. Explain creative use of these system in Newsfeed system
3. Describe traditional approach and message box based approach to design News feed system like Facebook.
4. How do we scale news feed system with Cassandra.
5. Full case study at <https://thenewstack.io/building-scalable-news-feed-applications-using-redis-and-cassandra/>

**Designing News Feed System like Facebook**



**Question 3: Read following article on Google Architecture and answer following questions.**

1. After reading following article please describe how Google is different and why google is king of scalability
2. Summarize Google File System and how is it different from Windows or Linux file system
3. What is Map Reduce w.r.t. Google Architecture. What is Google Big Table?
4. Please write your perspective and most effective lessons learned from Google Architecture.

[**Google Architecture**](http://highscalability.com/blog/2008/11/22/google-architecture.html)

Google is the King of scalability. Everyone knows Google for their large, sophisticated, and fast searching, but they don't just shine in search. Their platform approach to building scalable applications allows them to roll out internet scale applications at an alarmingly high competition crushing rate. Their goal is always to build a higher performing higher scaling infrastructure to support their products. How do they do that?

**Information Sources**

1. [Video: Building Large Systems at Google](http://video.google.com/videoplay?docid=-5699448884004201579)
2. [Google Lab: The Google File System](http://labs.google.com/papers/gfs.html)
3. [Google Lab: MapReduce: Simplified Data Processing on Large Clusters](http://labs.google.com/papers/mapreduce.html)
4. [Google Lab: BigTable](http://labs.google.com/papers/bigtable.html).
5. [Video: BigTable: A Distributed Structured Storage System](http://video.google.com/videoplay?docid=7278544055668715642).
6. [Google Lab: The Chubby Lock Service for Loosely-Coupled Distributed Systems](http://labs.google.com/papers/chubby.html).
7. [How Google Works](http://www.baselinemag.com/article2/0,1540,1985514,00.asp) by David Carr in Baseline Magazine.
8. [Google Lab: Interpreting the Data: Parallel Analysis with Sawzall](http://labs.google.com/papers/sawzall.html).
9. [Dare Obasonjo's Notes on the scalability conference.](http://www.25hoursaday.com/weblog/2007/06/25/GoogleScalabilityConferenceTripReportMapReduceBigTableAndOtherDistributedSystemAbstractionsForHandlingLargeDatasets.aspx)

**Platform**

1. Linux
2. A large diversity of languages: Python, Java, C++

**What's Inside?**

**The Stats**

1. Estimated 450,000 low-cost commodity servers in 2006
2. In 2005 Google indexed 8 billion web pages. By now, who knows?
3. Currently there over 200 GFS clusters at Google. A cluster can have 1000 or even 5000 machines. Pools of tens of thousands of machines retrieve data from GFS clusters that run as large as 5 petabytes of storage. Aggregate read/write throughput can be as high as 40 gigabytes/second across the cluster.
4. Currently there are 6000 MapReduce applications at Google and hundreds of new applications are being written each month.
5. BigTable scales to store billions of URLs, hundreds of terabytes of satellite imagery, and preferences for hundreds of millions of users.

**The Stack**

Google visualizes their infrastructure as a three layer stack:

1. Products: search, advertising, email, maps, video, chat, blogger
2. Distributed Systems Infrastructure: GFS, MapReduce, and BigTable.
3. Computing Platforms: a bunch of machines in a bunch of different data centers
4. Make sure easy for folks in the company to deploy at a low cost.
5. Look at price performance data on a per application basis. Spend more money on hardware to not lose log data, but spend less on other types of data. Having said that, they don't lose data.

**Reliable Storage Mechanism With GFS (Google File System)**

1. Reliable scalable storage is a core need of any application. GFS is their core storage platform.
2. Google File System - large distributed log structured file system in which they throw in a lot of data.
3. Why build it instead of using something off the shelf? Because they control everything and it's the platform that distinguishes them from everyone else. They required:  
   - high reliability across data centers  
   - scalability to thousands of network nodes  
   - huge read/write bandwidth requirements  
   - support for large blocks of data which are gigabytes in size.  
   - efficient distribution of operations across nodes to reduce bottlenecks
4. System has master and chunk servers.  
   - Master servers keep metadata on the various data files. Data are stored in the file system in 64MB chunks. Clients talk to the master servers to perform metadata operations on files and to locate the chunk server that contains the needed they need on disk.  
   - Chunk servers store the actual data on disk. Each chunk is replicated across three different chunk servers to create redundancy in case of server crashes. Once directed by a master server, a client application retrieves files directly from chunk servers.
5. A new application coming on line can use an existing GFS cluster or they can make your own. It would be interesting to understand the provisioning process they use across their data centers.
6. Key is enough infrastructure to make sure people have choices for their application. GFS can be tuned to fit individual application needs.

**Do Something With The Data Using MapReduce**

1. Now that you have a good storage system, how do you do anything with so much data? Let's say you have many TBs of data stored across a 1000 machines. Databases don't scale or cost effectively scale to those levels. That's where MapReduce comes in.
2. MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.
3. Why use MapReduce?  
   - Nice way to partition tasks across lots of machines.  
   - Handle machine failure.  
   - Works across different application types, like search and ads. Almost every application has map reduce type operations. You can precompute useful data, find word counts, sort TBs of data, etc.  
   - Computation can automatically move closer to the IO source.
4. The MapReduce system has three different types of servers.  
   - The Master server assigns user tasks to map and reduce servers. It also tracks the state of the tasks.  
   - The Map servers accept user input and performs map operations on them. The results are written to intermediate files  
   - The Reduce servers accepts intermediate files produced by map servers and performs reduce operation on them.
5. For example, you want to count the number of words in all web pages. You would feed all the pages stored on GFS into MapReduce. This would all be happening on 1000s of machines simultaneously and all the coordination, job scheduling, failure handling, and data transport would be done automatically.  
   - The steps look like: GFS -> Map -> Shuffle -> Reduction -> Store Results back into GFS.  
   - In MapReduce a map maps one view of data to another, producing a key value pair, which in our example is word and count.  
   - Shuffling aggregates key types.  
   - The reductions sums up all the key value pairs and produces the final answer.
6. The Google indexing pipeline has about 20 different map reductions. A pipeline looks at data with a whole bunch of records and aggregating keys. A second map-reduce comes a long, takes that result and does something else. And so on.
7. Programs can be very small. As little as 20 to 50 lines of code.
8. One problem is stragglers. A straggler is a computation that is going slower than others which holds up everyone. Stragglers may happen because of slow IO (say a bad controller) or from a temporary CPU spike. The solution is to run multiple of the same computations and when one is done kill all the rest.
9. Data transferred between map and reduce servers is compressed. The idea is that because servers aren't CPU bound it makes sense to spend on data compression and decompression in order to save on bandwidth and I/O.

**Storing Structured Data In BigTable**

1. BigTable is a large scale, fault tolerant, self managing system that includes terabytes of memory and petabytes of storage. It can handle millions of reads/writes per second.
2. BigTable is a distributed hash mechanism built on top of GFS. It is not a relational database. It doesn't support joins or SQL type queries.
3. It provides lookup mechanism to access structured data by key. GFS stores opaque data and many applications needs has data with structure.
4. Commercial databases simply don't scale to this level and they don't work across 1000s machines.
5. By controlling their own low level storage system Google gets more control and leverage to improve their system. For example, if they want features that make cross data center operations easier, they can build it in.
6. Machines can be added and deleted while the system is running and the whole system just works.
7. Each data item is stored in a cell which can be accessed using a row key, column key, or timestamp.
8. Each row is stored in one or more tablets. A tablet is a sequence of 64KB blocks in a data format called SSTable.
9. BigTable has three different types of servers:  
   - The Master servers assign tablets to tablet servers. They track where tablets are located and redistributes tasks as needed.  
   - The Tablet servers process read/write requests for tablets. They split tablets when they exceed size limits (usually 100MB - 200MB). When a tablet server fails, then a 100 tablet servers each pickup 1 new tablet and the system recovers.  
   - The Lock servers form a distributed lock service. Operations like opening a tablet for writing, Master aribtration, and access control checking require mutual exclusion.
10. A locality group can be used to physically store related bits of data together for better locality of reference.
11. Tablets are cached in RAM as much as possible.

**Hardware**

1. When you have a lot of machines how do you build them to be cost efficient and use power efficiently?
2. Use ultra cheap commodity hardware and built software on top to handle their death.
3. A 1,000-fold computer power increase can be had for a 33 times lower cost if you you use a failure-prone infrastructure rather than an infrastructure built on highly reliable components. You must build reliability on top of unreliability for this strategy to work.
4. Linux, in-house rack design, PC class mother boards, low end storage.
5. Price per wattage on performance basis isn't getting better. Have huge power and cooling issues.
6. Use a mix of collocation and their own data centers.

**Misc**

1. Push changes out quickly rather than wait for QA.
2. Libraries are the predominant way of building programs.
3. Some are applications are provided as services, like crawling.
4. An infrastructure handles versioning of applications so they can be release without a fear of breaking things.

**Future Directions For Google**

1. Support geo-distributed clusters.
2. Create a single global namespace for all data. Currently data is segregated by cluster.
3. More and better automated migration of data and computation.
4. Solve consistency issues that happen when you couple wide area replication with network partitioning (e.g. keeping services up even if a cluster goes offline for maintenance or due to some sort of outage).

**Lessons Learned**

1. **Infrastructure can be a competitive advantage**. It certainly is for Google. They can roll out new internet services faster, cheaper, and at scale at few others can compete with. Many companies take a completely different approach. Many companies treat infrastructure as an expense. Each group will use completely different technologies and their will be little planning and commonality of how to build systems. Google thinks of themselves as a systems engineering company, which is a very refreshing way to look at building software.
2. **Spanning multiple data centers is still an unsolved problem**. Most websites are in one and at most two data centers. How to fully distribute a website across a set of data centers is, shall we say, tricky.
3. **Take a look at Hadoop** if you don't have the time to rebuild all this infrastructure from scratch yourself. Hadoop is an open source implementation of many of the same ideas presented here.
4. **An under appreciated advantage** of a platform approach is junior developers can quickly and confidently create robust applications on top of the platform. If every project needs to create the same distributed infrastructure wheel you'll run into difficulty because the people who know how to do this are relatively rare.
5. **Synergy isn't always crap**. By making all parts of a system work together an improvement in one helps them all. Improve the file system and everyone benefits immediately and transparently. If every project uses a different file system then there's no continual incremental improvement across the entire stack.
6. **Build self-managing systems that work without having to take the system down**. This allows you to more easily rebalance resources across servers, add more capacity dynamically, bring machines off line, and gracefully handle upgrades.
7. **Create a Darwinian infrastructure**. Perform time consuming operation in parallel and take the winner.
8. **Don't ignore the Academy**. Academia has a lot of good ideas that don't get translated into production environments. Most of what Google has done has prior art, just not prior large scale deployment.
9. **Consider compression**. Compression is a good option when you have a lot of CPU to throw around and limited IO.

**Update 2:** [Sorting 1 PB with MapReduce](http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html). PB is not peanut-butter-and-jelly misspelled. It's 1 petabyte or 1000 terabytes or 1,000,000 gigabytes. *It took six hours and two minutes to sort 1PB (10 trillion 100-byte records) on 4,000 computers* and the results were replicated thrice on 48,000 disks.  
**Update:** [Greg Linden](http://glinden.blogspot.com/2008/01/mapreducing-20-petabytes-per-day.html) points to a new Google article [MapReduce: simplified data processing on large clusters](http://labs.google.com/papers/mapreduce-osdi04.pdf). Some interesting stats: 100k MapReduce jobs are executed each day; more than 20 petabytes of data are processed per day; more than 10k MapReduce programs have been implemented; machines are dual processor with gigabit ethernet and 4-8 GB of memory.

**Question 4. Answer following questions after reading “Designing Instagram” Article:**

1. Explain Instagram Architecture and draw stack diagram. Please refer to techstacks.io (<https://techstacks.io/stacks/instagram/)> an explain different technology stack components utilized and salient features and importance of each component of the stack.
2. Explain briefly capacity estimation for Instagram
3. What is data sharding w.r.t. Instagram
4. Explain Ranking of Newsfeed and newsfeed generation methods associated with it
5. Explain Load Balancing and Caching @ Instagram. Why is it important?

**Designing Instagram**

Let's design a photo-sharing service like Instagram, where users can upload photos to share them with other users.

Similar Services: Flickr, Picasa

Difficulty Level: Medium

**1. What is Instagram?**

Instagram is a social networking service which enables its users to upload and share their photos and videos with other users. Instagram users can choose to share information either publicly or privately. Anything shared publicly can be seen by any other user, whereas privately shared content can only be accessed by a specified set of people. Instagram also enables its users to share through many other social networking platforms, such as Facebook, Twitter, Flickr, and Tumblr.

For the sake of this exercise, we plan to design a simpler version of Instagram, where a user can share photos and can also follow other users. The ‘News Feed’ for each user will consist of top photos of all the people the user follows.

**2. Requirements and Goals of the System**

We’ll focus on the following set of requirements while designing the Instagram:

**Functional Requirements**

1. Users should be able to upload/download/view photos.
2. Users can perform searches based on photo/video titles.
3. Users can follow other users.
4. The system should be able to generate and display a user’s News Feed consisting of top photos from all the people the user follows.

**Non-functional Requirements**

1. Our service needs to be highly available.
2. The acceptable latency of the system is 200ms for News Feed generation.
3. Consistency can take a hit (in the interest of availability), if a user doesn’t see a photo for a while; it should be fine.
4. The system should be highly reliable; any uploaded photo or video should never be lost.

**Not in scope:** Adding tags to photos, searching photos on tags, commenting on photos, tagging users to photos, who to follow, etc.

**3. Some Design Considerations**

The system would be read-heavy, so we will focus on building a system that can retrieve photos quickly.

1. Practically, users can upload as many photos as they like. Efficient management of storage should be a crucial factor while designing this system.
2. Low latency is expected while viewing photos.
3. Data should be 100% reliable. If a user uploads a photo, the system will guarantee that it will never be lost.

**4. Capacity Estimation and Constraints**

* Let’s assume we have 500M total users, with 1M daily active users.
* 2M new photos every day, 23 new photos every second.
* Average photo file size => 200KB
* Total space required for 1 day of photos

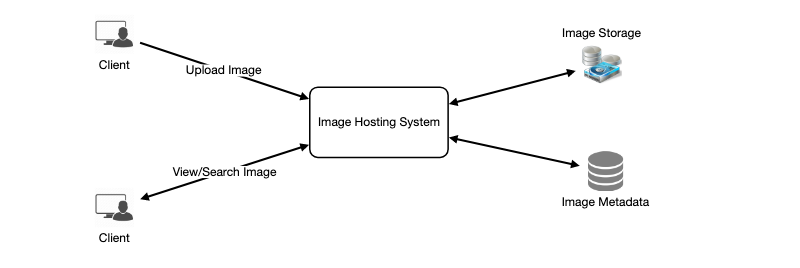
2M \* 200KB => 400 GB

* Total space required for 10 years:

400GB \* 365 (days a year) \* 10 (years) ~= 1425TB

**5. High Level System Design**

At a high-level, we need to support two scenarios, one to upload photos and the other to view/search photos. Our service would need some [object storage](https://en.wikipedia.org/wiki/Object_storage) servers to store photos and also some database servers to store metadata information about the photos.



**6. Database Schema**

💡      ***Defining the DB schema in the early stages of the interview would help to understand the data flow among various components and later would guide towards data partitioning.***

We need to store data about users, their uploaded photos, and people they follow. Photo table will store all data related to a photo; we need to have an index on (PhotoID, CreationDate) since we need to fetch recent photos first.



A straightforward approach for storing the above schema would be to use an RDBMS like MySQL since we require joins. But relational databases come with their challenges, especially when we need to scale them. For details, please take a look at [SQL vs. NoSQL](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5728116278296576/).

We can store photos in a distributed file storage like [HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop) or [S3](https://en.wikipedia.org/wiki/Amazon_S3).

We can store the above schema in a distributed key-value store to enjoy the benefits offered by NoSQL. All the metadata related to photos can go to a table where the ‘key’ would be the ‘PhotoID’ and the ‘value’ would be an object containing PhotoLocation, UserLocation, CreationTimestamp, etc.

We need to store relationships between users and photos, to know who owns which photo. We also need to store the list of people a user follows. For both of these tables, we can use a wide-column datastore like [Cassandra](https://en.wikipedia.org/wiki/Apache_Cassandra). For the ‘UserPhoto’ table, the ‘key’ would be ‘UserID’ and the ‘value’ would be the list of ‘PhotoIDs’ the user owns, stored in different columns. We will have a similar scheme for the ‘UserFollow’ table.

Cassandra or key-value stores in general, always maintain a certain number of replicas to offer reliability. Also, in such data stores, deletes don’t get applied instantly, data is retained for certain days (to support undeleting) before getting removed from the system permanently.

**7. Data Size Estimation**

Let’s estimate how much data will be going into each table and how much total storage we will need for 10 years.

**User:** Assuming each “int” and “dateTime” is four bytes, each row in the User’s table will be of 68 bytes:

UserID (4 bytes) + Name (20 bytes) + Email (32 bytes) + DateOfBirth (4 bytes) + CreationDate (4 bytes) + LastLogin (4 bytes) = 68 bytes

If we have 500 million users, we will need 32GB of total storage.

500 million \* 68 ~= 32GB

**Photo:** Each row in Photo’s table will be of 284 bytes:

PhotoID (4 bytes) + UserID (4 bytes) + PhotoPath (256 bytes) + PhotoLatitude (4 bytes) + PhotLongitude(4 bytes) + UserLatitude (4 bytes) + UserLongitude (4 bytes) + CreationDate (4 bytes) = 284 bytes

If 2M new photos get uploaded every day, we will need 0.5GB of storage for one day:

2M \* 284 bytes ~= 0.5GB per day

For 10 years we will need 1.88TB of storage.

**UserFollow:** Each row in the UserFollow table will consist of 8 bytes. If we have 500 million users and on average each user follows 500 users. We would need 1.82TB of storage for the UserFollow table:

500 million users \* 500 followers \* 8 bytes ~= 1.82TB

Total space required for all tables for 10 years will be 3.7TB:

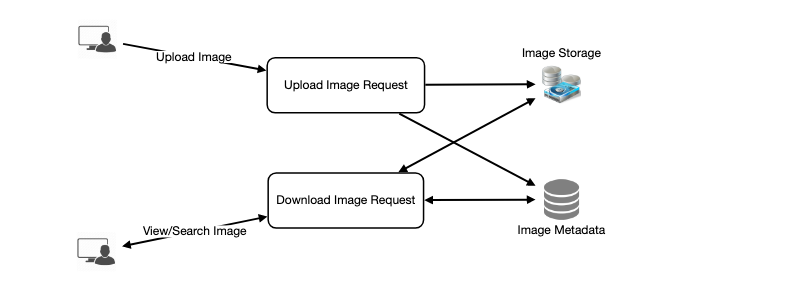
32GB + 1.88TB + 1.82TB ~= 3.7TB

**8. Component Design**

Photo uploads (or writes) can be slow as they have to go to the disk, whereas reads will be faster, especially if they are being served from cache.

Uploading users can consume all the available connections, as uploading is a slow process. This means that ‘reads’ cannot be served if the system gets busy with all the write requests. We should keep in mind that web servers have a connection limit before designing our system. If we assume that a web server can have a maximum of 500 connections at any time, then it can’t have more than 500 concurrent uploads or reads. To handle this bottleneck we can split reads and writes into separate services. We will have dedicated servers for reads and different servers for writes to ensure that uploads don’t hog the system.

Separating photos’ read and write requests will also allow us to scale and optimize each of these operations independently.



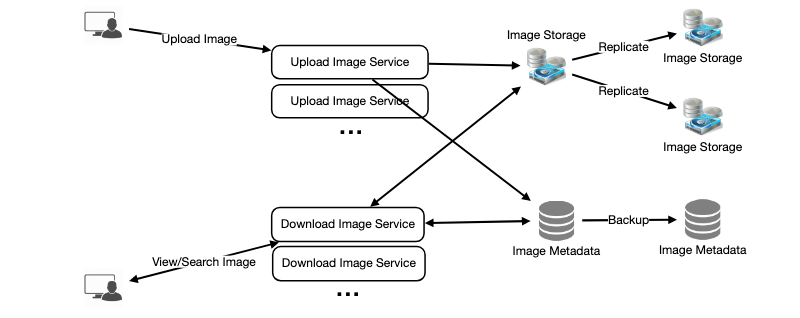
**9. Reliability and Redundancy**

Losing files is not an option for our service. Therefore, we will store multiple copies of each file so that if one storage server dies we can retrieve the photo from the other copy present on a different storage server.

This same principle also applies to other components of the system. If we want to have high availability of the system, we need to have multiple replicas of services running in the system, so that if a few services die down the system still remains available and running. Redundancy removes the single point of failure in the system.

If only one instance of a service is required to run at any point, we can run a redundant secondary copy of the service that is not serving any traffic, but it can take control after the failover when primary has a problem.

Creating redundancy in a system can remove single points of failure and provide a backup or spare functionality if needed in a crisis. For example, if there are two instances of the same service running in production and one fails or degrades, the system can failover to the healthy copy. Failover can happen automatically or require manual intervention.



**10. Data Sharding**

Let’s discuss different schemes for metadata sharding:

**a. Partitioning based on UserID** Let’s assume we shard based on the ‘UserID’ so that we can keep all photos of a user on the same shard. If one DB shard is 1TB, we will need four shards to store 3.7TB of data. Let’s assume for better performance and scalability we keep 10 shards.

So we’ll find the shard number by UserID % 10 and then store the data there. To uniquely identify any photo in our system, we can append shard number with each PhotoID.

**How can we generate PhotoIDs?** Each DB shard can have its own auto-increment sequence for PhotoIDs and since we will append ShardID with each PhotoID, it will make it unique throughout our system.

**What are the different issues with this partitioning scheme?**

1. How would we handle hot users? Several people follow such hot users and a lot of other people see any photo they upload.
2. Some users will have a lot of photos compared to others, thus making a non-uniform distribution of storage.
3. What if we cannot store all pictures of a user on one shard? If we distribute photos of a user onto multiple shards will it cause higher latencies?
4. Storing all photos of a user on one shard can cause issues like unavailability of all of the user’s data if that shard is down or higher latency if it is serving high load etc.

**b. Partitioning based on PhotoID** If we can generate unique PhotoIDs first and then find a shard number through “PhotoID % 10”, the above problems will have been solved. We would not need to append ShardID with PhotoID in this case as PhotoID will itself be unique throughout the system.

**How can we generate PhotoIDs?** Here we cannot have an auto-incrementing sequence in each shard to define PhotoID because we need to know PhotoID first to find the shard where it will be stored. One solution could be that we dedicate a separate database instance to generate auto-incrementing IDs. If our PhotoID can fit into 64 bits, we can define a table containing only a 64 bit ID field. So whenever we would like to add a photo in our system, we can insert a new row in this table and take that ID to be our PhotoID of the new photo.

**Wouldn’t this key generating DB be a single point of failure?** Yes, it would be. A workaround for that could be defining two such databases with one generating even numbered IDs and the other odd numbered. For the MySQL, the following script can define such sequences:

KeyGeneratingServer1:

auto-increment-increment = 2

auto-increment-offset = 1

KeyGeneratingServer2:

auto-increment-increment = 2

auto-increment-offset = 2

We can put a load balancer in front of both of these databases to round robin between them and to deal with downtime. Both these servers could be out of sync with one generating more keys than the other, but this will not cause any issue in our system. We can extend this design by defining separate ID tables for Users, Photo-Comments, or other objects present in our system.

**Alternately,** we can implement a ‘key’ generation scheme similar to what we have discussed in [Designing a URL Shortening service like TinyURL](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904).

**How can we plan for the future growth of our system?** We can have a large number of logical partitions to accommodate future data growth, such that in the beginning, multiple logical partitions reside on a single physical database server. Since each database server can have multiple database instances on it, we can have separate databases for each logical partition on any server. So whenever we feel that a particular database server has a lot of data, we can migrate some logical partitions from it to another server. We can maintain a config file (or a separate database) that can map our logical partitions to database servers; this will enable us to move partitions around easily. Whenever we want to move a partition, we only have to update the config file to announce the change.

**11. Ranking and News Feed Generation**

To create the News Feed for any given user, we need to fetch the latest, most popular and relevant photos of the people the user follows.

For simplicity, let’s assume we need to fetch top 100 photos for a user’s News Feed. Our application server will first get a list of people the user follows and then fetch metadata info of latest 100 photos from each user. In the final step, the server will submit all these photos to our ranking algorithm which will determine the top 100 photos (based on recency, likeness, etc.) and return them to the user. A possible problem with this approach would be higher latency as we have to query multiple tables and perform sorting/merging/ranking on the results. To improve the efficiency, we can pre-generate the News Feed and store it in a separate table.

**Pre-generating the News Feed:** We can have dedicated servers that are continuously generating users’ News Feeds and storing them in a ‘UserNewsFeed’ table. So whenever any user needs the latest photos for their News Feed, we will simply query this table and return the results to the user.

Whenever these servers need to generate the News Feed of a user, they will first query the UserNewsFeed table to find the last time the News Feed was generated for that user. Then, new News Feed data will be generated from that time onwards (following the steps mentioned above).

**What are the different approaches for sending News Feed contents to the users?**

**1. Pull:** Clients can pull the News Feed contents from the server on a regular basis or manually whenever they need it. Possible problems with this approach are a) New data might not be shown to the users until clients issue a pull request b) Most of the time pull requests will result in an empty response if there is no new data.

**2. Push:** Servers can push new data to the users as soon as it is available. To efficiently manage this, users have to maintain a [Long Poll](https://en.wikipedia.org/wiki/Push_technology#Long_polling) request with the server for receiving the updates. A possible problem with this approach is, a user who follows a lot of people or a celebrity user who has millions of followers; in this case, the server has to push updates quite frequently.

**3. Hybrid:** We can adopt a hybrid approach. We can move all the users who have a high number of follows to a pull-based model and only push data to those users who have a few hundred (or thousand) follows. Another approach could be that the server pushes updates to all the users not more than a certain frequency, letting users with a lot of follows/updates to regularly pull data.

For a detailed discussion about News Feed generation, take a look at [Designing Facebook’s Newsfeed](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5641332169113600).

**12. News Feed Creation with Sharded Data**

One of the most important requirement to create the News Feed for any given user is to fetch the latest photos from all people the user follows. For this, we need to have a mechanism to sort photos on their time of creation. To efficiently do this, we can make photo creation time part of the PhotoID. As we will have a primary index on PhotoID, it will be quite quick to find the latest PhotoIDs.

We can use epoch time for this. Let’s say our PhotoID will have two parts; the first part will be representing epoch time and the second part will be an auto-incrementing sequence. So to make a new PhotoID, we can take the current epoch time and append an auto-incrementing ID from our key-generating DB. We can figure out shard number from this PhotoID ( PhotoID % 10) and store the photo there.

**What could be the size of our PhotoID**? Let’s say our epoch time starts today, how many bits we would need to store the number of seconds for next 50 years?

86400 sec/day \* 365 (days a year) \* 50 (years) => 1.6 billion seconds

We would need 31 bits to store this number. Since on the average, we are expecting 23 new photos per second; we can allocate 9 bits to store auto incremented sequence. So every second we can store (2^9 => 512) new photos. We can reset our auto incrementing sequence every second.

We will discuss more details about this technique under ‘Data Sharding’ in [Designing Twitter](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5741031244955648).

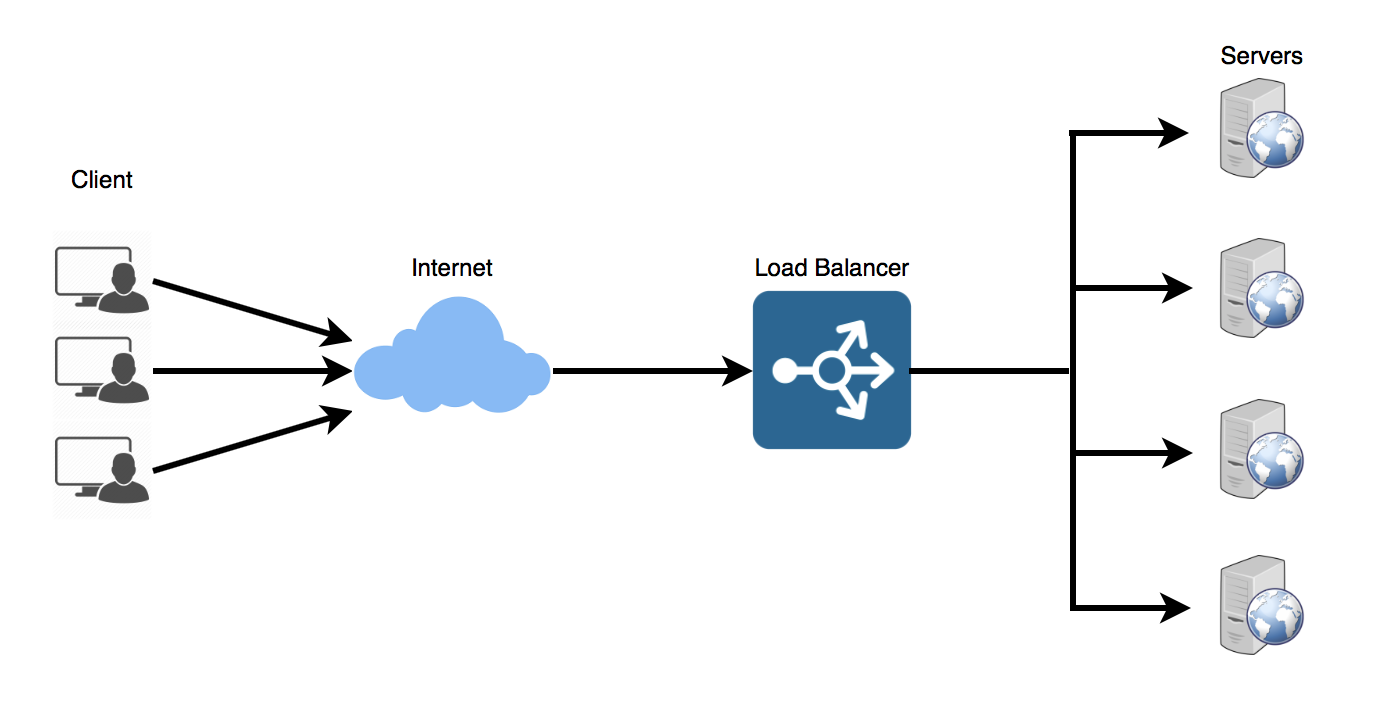
**13. Cache and Load balancing**

Our service would need a massive-scale photo delivery system to serve the globally distributed users. Our service should push its content closer to the user using a large number of geographically distributed photo cache servers and use CDNs (for details see [Caching](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5643440998055936/)).

We can introduce a cache for metadata servers to cache hot database rows. We can use Memcache to cache the data and Application servers before hitting database can quickly check if the cache has desired rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system. Under this policy, we discard the least recently viewed row first.

**How can we build more intelligent cache?** If we go with 80-20 rule, i.e., 20% of daily read volume for photos is generating 80% of traffic which means that certain photos are so popular that the majority of people read them. This dictates that we can try caching 20% of daily read volume of photos and metadata.

**Load Balancing**

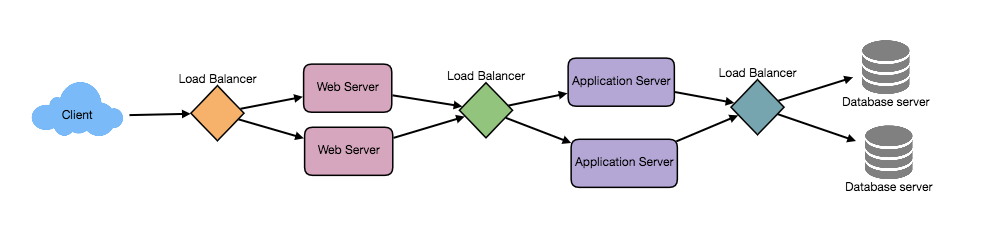


Load Balancer (LB) is another critical component of any distributed system. It helps to spread the traffic across a cluster of servers to improve responsiveness and availability of applications, websites or databases. LB also keeps track of the status of all the resources while distributing requests. If a server is not available to take new requests or is not responding or has elevated error rate, LB will stop sending traffic to such a server.

Typically a load balancer sits between the client and the server accepting incoming network and application traffic and distributing the traffic across multiple backend servers using various algorithms. By balancing application requests across multiple servers, a load balancer reduces individual server load and prevents any one application server from becoming a single point of failure, thus improving overall application availability and responsiveness.

To utilize full scalability and redundancy, we can try to balance the load at each layer of the system. We can add LBs at three places:

1. Between the user and the web server
2. Between web servers and an internal platform layer, like application servers or cache servers
3. Between internal platform layer and database.



**Benefits of Load Balancing**

1. Users experience faster, uninterrupted service. Users won’t have to wait for a single struggling server to finish its previous tasks. Instead, their requests are immediately passed on to a more readily available resource.
2. Service providers experience less downtime and higher throughput. Even a full server failure won’t affect the end user experience as the load balancer will simply route around it to a healthy server.
3. Load balancing makes it easier for system administrators to handle incoming requests while decreasing wait time for users.
4. Smart load balancers provide benefits like predictive analytics that determine traffic bottlenecks before they happen. As a result, the smart load balancer gives an organization actionable insights. These are key to automation and can help drive business decisions.
5. System administrators experience fewer failed or stressed components. Instead of a single device performing a lot of work, load balancing has several devices perform a little bit of work.

**Load Balancing Algorithms**

**How does the load balancer choose the backend server?**

Load balancers consider two factors before forwarding a request to a backend server. They will first ensure that the server they choose is actually responding appropriately to requests and then use a pre-configured algorithm to select one from the set of healthy servers. We will discuss these algorithms shortly.

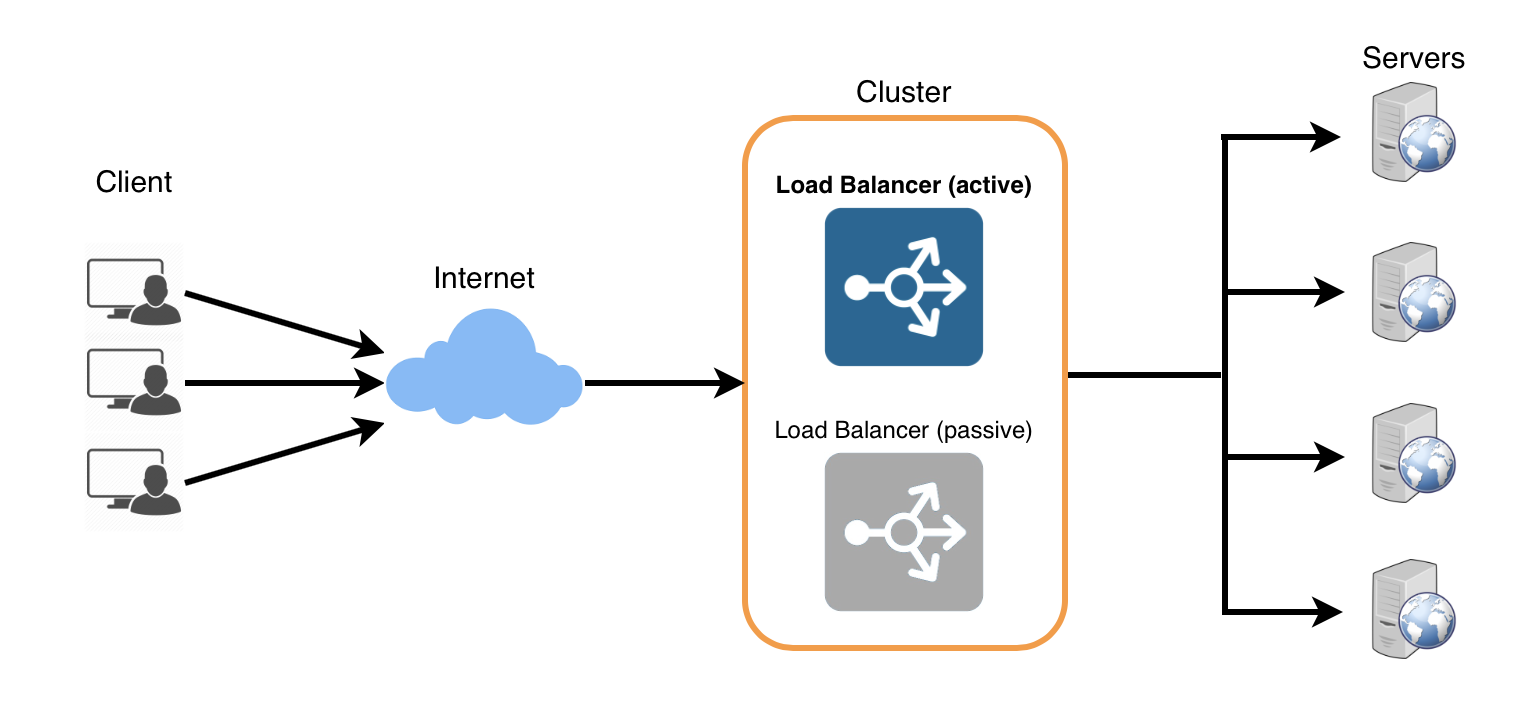
**Health Checks** - Load balancers should only forward traffic to “healthy” backend servers. To monitor the health of a backend server, “health checks” regularly attempt to connect to backend servers to ensure that servers are listening. If a server fails a health check, it is automatically removed from the pool, and traffic will not be forwarded to it until it responds to the health checks again.

There is a variety of load balancing methods, which use different algorithms for different needs.

1. **Least Connection Method** — This method directs traffic to the server with the fewest active connections. This approach is quite useful when there are a large number of persistent client connections which are unevenly distributed between the servers.
2. **Least Response Time Method** — This algorithm directs traffic to the server with the fewest active connections and the lowest average response time.
3. **Least Bandwidth Method** - This method selects the server that is currently serving the least amount of traffic measured in megabits per second (Mbps).
4. **Round Robin Method** — This method cycles through a list of servers and sends each new request to the next server. When it reaches the end of the list, it starts over at the beginning. It is most useful when the servers are of equal specification and there are not many persistent connections.
5. **Weighted Round Robin Method** — The weighted round-robin scheduling is designed to better handle servers with different processing capacities. Each server is assigned a weight (an integer value that indicates the processing capacity). Servers with higher weights receive new connections before those with less weights and servers with higher weights get more connections than those with less weights.
6. **IP Hash** — Under this method, a hash of the IP address of the client is calculated to redirect the request to a server.

**Redundant Load Balancers**

The load balancer can be a single point of failure; to overcome this, a second load balancer can be connected to the first to form a cluster. Each LB monitors the health of the other and, since both of them are equally capable of serving traffic and failure detection, in the event the main load balancer fails, the second load balancer takes over.



Following links have some good discussion about load balancers:

[1] [What is load balancing](https://avinetworks.com/what-is-load-balancing/)

[2] [Introduction to architecting systems](https://lethain.com/introduction-to-architecting-systems-for-scale/)

[3] [Load balancing](https://en.wikipedia.org/wiki/Load_balancing_(computing))

**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. Caches take advantage of the locality of reference principle: recently requested data is likely to be requested again. They are used in almost every layer of computing: hardware, operating systems, web browsers, web applications, and more. A cache is like short-term memory: it has a limited amount of space, but is typically faster than the original data source and contains the most recently accessed items. Caches can exist at all levels in architecture, but are often found at the level nearest to the front end where they are implemented to return data quickly without taxing downstream levels.

**Application server cache**

Placing a cache directly on a request layer node enables the local storage of response data. Each time a request is made to the service, the node will quickly return local cached data if it exists. If it is not in the cache, the requesting node will query the data from disk. The cache on one request layer node could also be located both in memory (which is very fast) and on the node’s local disk (faster than going to network storage).

What happens when you expand this to many nodes? If the request layer is expanded to multiple nodes, it’s still quite possible to have each node host its own cache. However, if your load balancer randomly distributes requests across the nodes, the same request will go to different nodes, thus increasing cache misses. Two choices for overcoming this hurdle are global caches and distributed caches.

**Content Distribution Network (CDN)**

CDNs are a kind of cache that comes into play for sites serving large amounts of static media. In a typical CDN setup, a request will first ask the CDN for a piece of static media; the CDN will serve that content if it has it locally available. If it isn’t available, the CDN will query the back-end servers for the file, cache it locally, and serve it to the requesting user.

If the system we are building isn’t yet large enough to have its own CDN, we can ease a future transition by serving the static media off a separate subdomain (e.g. [static.yourservice.com](http://static.yourservice.com/)) using a lightweight HTTP server like Nginx, and cut-over the DNS from your servers to a CDN later.

**Cache Invalidation**

While caching is fantastic, it does require some maintenance for keeping cache coherent with the source of truth (e.g., database). If the data is modified in the database, it should be invalidated in the cache; if not, this can cause inconsistent application behavior.

Solving this problem is known as cache invalidation; there are three main schemes that are used:

**Write-through cache:** Under this scheme, data is written into the cache and the corresponding database at the same time. The cached data allows for fast retrieval and, since the same data gets written in the permanent storage, we will have complete data consistency between the cache and the storage. Also, this scheme ensures that nothing will get lost in case of a crash, power failure, or other system disruptions.

Although, write through minimizes the risk of data loss, since every write operation must be done twice before returning success to the client, this scheme has the disadvantage of higher latency for write operations.

**Write-around cache:** This technique is similar to write through cache, but data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write operations that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a “cache miss” and must be read from slower back-end storage and experience higher latency.

**Write-back cache:** Under this scheme, data is written to cache alone and completion is immediately confirmed to the client. The write to the permanent storage is done after specified intervals or under certain conditions. This results in low latency and high throughput for write-intensive applications, however, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

**Cache eviction policies**

Following are some of the most common cache eviction policies:

1. First In First Out (FIFO): The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
2. Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
3. Least Recently Used (LRU): Discards the least recently used items first.
4. Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first.
5. Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first.
6. Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.

Following links have some good discussion about caching:  
[1] [Cache](https://en.wikipedia.org/wiki/Cache_(computing))  
[2] [Introduction to architecting systems](https://lethain.com/introduction-to-architecting-systems-for-scale/)

**Question 5. Read the following article on Different type of Software Architecture and summarize each architecture in your own words. Give few examples of products/apps using following each architecture in use in industry today**.

1. Two Tier
2. Domain Driven
3. Service Oriented
4. Three Tier
5. Microservice Architecture
6. Onion
7. N-Tier
8. Aspect Oriented
9. Event based

https://cazton.com/consulting/enterprise/software-architecture

<https://drive.google.com/file/d/1b7GWLtb9O4dnYyogoFVplt11Z27RxNk1/view?usp=sharing>

