# The basics

**Scalability:**

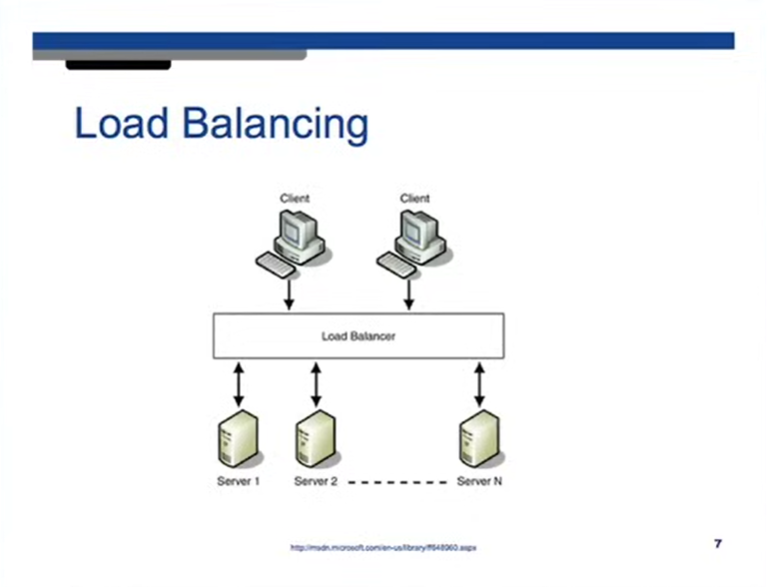
**A screen shot of a computer

Description automatically generated**

**Main Issues:** Hardware limit and single point of failure

**Horizontal Scaling:** Adding more machines.

**Main Issues:** Load balancing



In **vertical scaling**, when the client sends the request. **DNS** provides the IP address of the server to the client. In **horizontal scaling** we don’t have just one server, so, instead of the server, the **DNS** will provide the IP address of the **load balancer** which will then route the req to the server.

Another advantage of **horizontal scaling** is that we only need to make the IP address of the **load balancer** as public. The servers can have private IP addresses which is good for privacy and security.

We can also have a solution where the **DNS server** returns the IP address of one of the servers when user sends a request instead of sending the IP address of the load balancer.

How the **load balancer** distributes the load?

**Robin round scheduling:** The problem is if user sends a request to a webpage, he is served. If he sends the same request again, we should ideally route it to the same server because of cache. Another issue with this is that we lose sessions due to being routed to a different server. If you are logged in to a website, if we lose the session user will also lose its login and have to login again. So, using round robin is a bad idea.

We can store session details of the users in MySQL or other databases as well.

Or we can store the session details like which server you visited for a particular content in cookies. Cookie size is finite, so, you can store server id in the cookie and use that next time you want to visit the same site.

Load balancer can insert a cookie which has the server where we got the user data from in the browser of the user. This can be later used to go to the same server again and make use of the cache if possible.

**RAID: Redundant Array of Independent Disks:** RAID0, RAID1, RAID5, RAID10 etc.

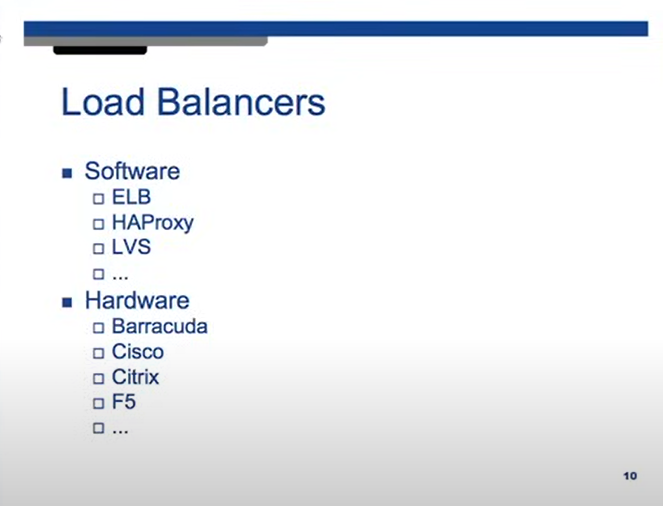
**RAID0:** If we have two hard disks, and the system wants to write a file, especially a large file, then the system can write some part of the file on disk1 and some part on disk2 and some part of disk1 and so on. This is because disk takes some time to write the data, in that time we write next data on the second machine and then back onto the first machine. This makes the speed of writing a lot faster. This is called **striping.**

**RAID1:** Whenever we write the file, we write it in both disks. There is performance overhead due to this. If any of the disk dies, data is not lost. This is **redundancy.**

**RAID10:** It is a combination of both RAID1 and RAID0. We use four drives. It has both **striping, striping.**

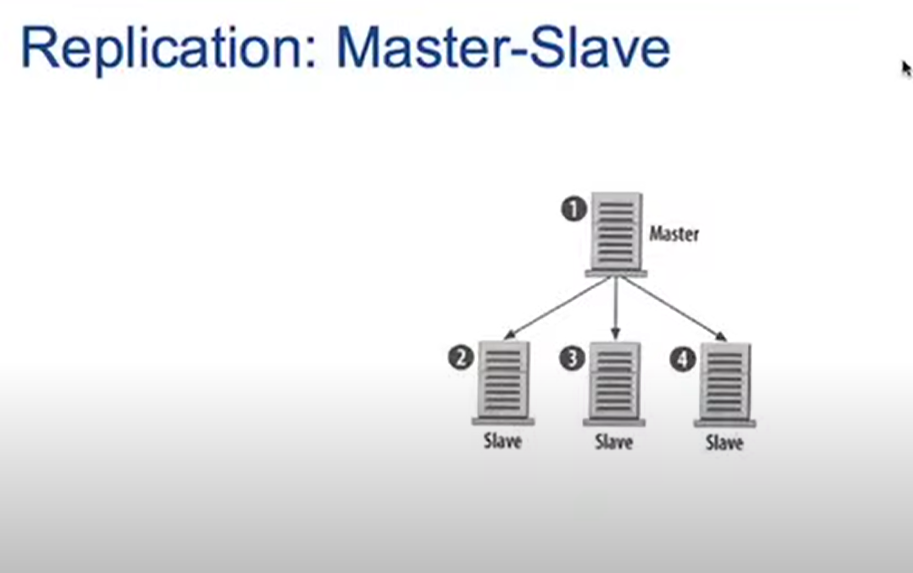
**RAID5, RAID6:** Variants of RAID1. If you have many disks, only one is used to keep redundant data.

All these techniques are used to reduce downtime.



**Cache: LRU (Least Recently Used):** Remove the data from the cache which was used in longest time. Cache will be used in the server.

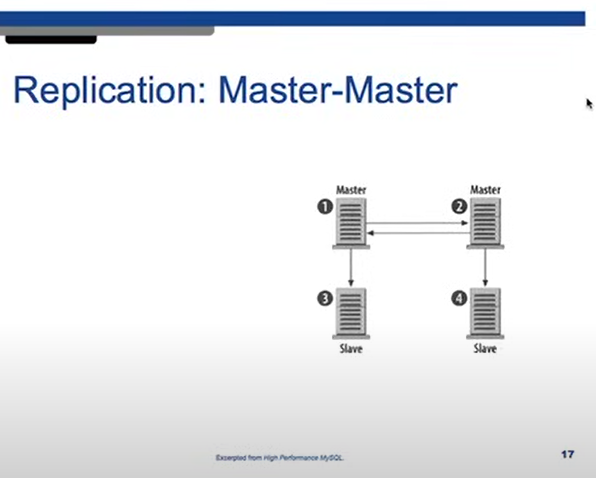
**Replication:**

****

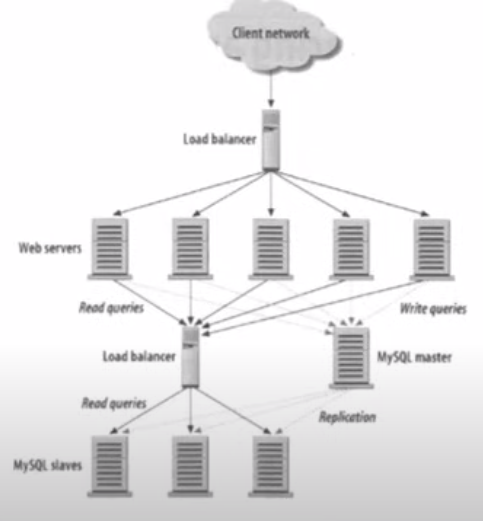
We read and write to **master database**. We copy this data to **slave databases**. Master and slaves are identical ideally. If one of the DB dies, we have backup. If master dies, we can make any of the slaves, master DB using some confuguration.

Above technique is much better for read heavy websites rather than write heavy websites. We can do reads from any of these servers. The reading becomes fast. If we want to write, we write to the master and then replicate that to other servers. This change has to propogate to the other DBs before we can do any reads. So, clearly, it is a better solution for read heavy systems.

The problem occures when our master dies. Since we can only write to the master. If the master dies, we can not do any reads until a slave is promoted to master. So, system becomes unusable until a master is chosen.



All servers are master. You can write to any and read from any.

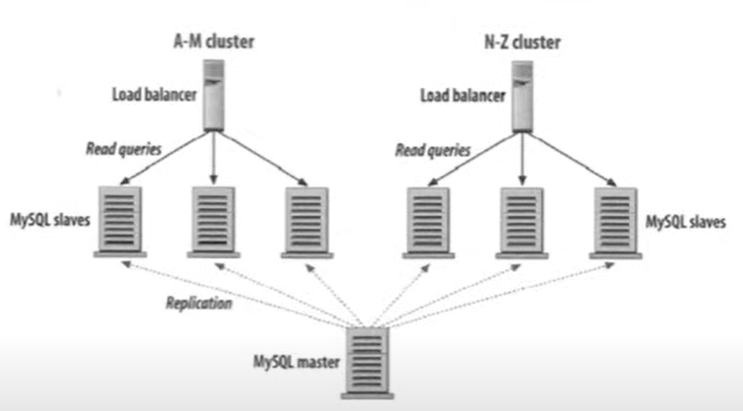


You can find the single point of failures by just looking at this picture and seeing any components which has a lot of incoming and outgoing and is bottlenecking.

Here, we see that load balancers are the bottleneck.

To avoid this, we usually also have a backup load balancer, the main load balancer sends a packet called **heartbeat,** to the backup load balancer. If the backup load balancer does not hear the heartbeat of the main load balancer, it is assumed to have died. This backup load balancer now becomes the main load balancer. Also, in load balancing terminology, we don’t call them master and slave, we call them **active and passive load balancers.**

How the passive balancer becomes active balancer. The backup one will just take the IP address of the active one and become active.



**A-M cluster** has the data for all the users with last name from a to m.

**N-Z cluster** has the data for all the users with last name from n to z.

This is called **partitioning.**

**High availability** is maintained using the heartbeat. Diagram of a server

Description automatically generated

Masters keep track of each other using the heartbeat.

Servers are usually in data centers ie some building in a geographical location. We also have replicas of the same servers in a different data center in some other grographical location. If one data center goes down, we need to send reuqest to the other data center. Moreover, requests that are closer to a certain geographic location must be routed to the closed server. This routing can be done through DNS. Different data centers with the same DB have different IP addresses. We can use DNS to give IP addresse of the closed data center to the user. This is called **global load balancing.**

**Scalability:**

[Scalability for Dummies Article](https://web.archive.org/web/20220530193926/https:/www.lecloud.net/tagged/scalability)

What is it that we really mean by scalability? **A service is said to be scalable if when we increase the resources in a system, it results in increased performance in a manner proportional to resources added. Increasing performance in general means serving more units of work, but it can also be to handle larger units of work, such as when datasets grow.**

In distributed systems there are other reasons for adding resources to a system; for example to improve the reliability of the offered service. Introducing redundancy is an important first line of defense against failures. An always-on service is said to be scalable if adding resources to facilitate redundancy does not result in a loss of performance.

Why is scalability so hard? Because scalability cannot be an after-thought. It requires applications and platforms to be designed with scaling in mind, such that adding resources actually results in improving the performance or that if redundancy is introduced the system performance is not adversely affected. Many algorithms that perform reasonably well under low load and small datasets can explode in cost if either requests rates increase, the dataset grows or the number of nodes in the distributed system increases.

A second problem area is that growing a system through scale-out generally results in a system that has to come to terms with heterogeneity. Resources in the system increase in diversity as next generations of hardware come on line, as bigger or more powerful resources become more cost-effective or when some resources are placed further apart. Heterogeneity means that some nodes will be able to process faster or store more data than other nodes in a system and algorithms that rely on uniformity either break down under these conditions or underutilize the newer resources.

Is achieving good scalability possible? Absolutely, but only if we architect and engineer our systems to take scalability into account. For the systems we build we must carefully inspect along which axis we expect the system to grow, where redundancy is required, and how one should handle heterogeneity in this system, and make sure that architects are aware of which tools they can use for under which conditions, and what the common pitfalls are.

Another way to look at **performance vs scalability:**

**If you have a performance problem, your system is slow for a single user.**

**If you have a scalability problem, your system is fast for a single user but slow under heavy load.**

**Latency is the time required to perform some action or to produce some result. Latency is measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods.**

**Throughput is the number of such actions executed or results produced per unit of time. This is measured in units of whatever is being produced (cars, motorcycles, I/O samples, memory words, iterations) per unit of time. The term "memory bandwidth" is sometimes used to specify the throughput of memory systems.**

**Generally, you should aim for maximal throughput with acceptable latency.**

**Availabilty vs Consistency:**

**CAP theorem:**

**The CAP Theorem states that, in a distributed system (a collection of interconnected nodes that share data.), you can only have two out of the following three guarantees across a write/read pair: Consistency, Availability, and Partition Tolerance - one of them must be sacrificed. However, as you will see below, you don't have as many options here as you might think.**

In a distributed computer system, you can only support two of the following guarantees:

**Consistency** - Every read receives the most recent write or an error. A read is guaranteed to return the most recent write for a given client.

**Availability** - Every request receives a response, without guarantee that it contains the most recent version of the information. A non-failing node will return a reasonable response within a reasonable amount of time (no error or timeout). The response may not have the latest updated data.

What does **available** mean?

A data store is available if and only if all get and set requests eventually return a response that's part of their specification. This does not permit error responses, since a system could be trivially available by always returning an error.

There is no requirement for a fixed time bound on the response, so the system can take as long as it likes to process a request. But the system must eventually respond.

Notice how this is both a strong and a weak requirement. It's strong because 100% of the requests must return a response (there's no 'degree of availability' here), but weak because the response can take an unbounded (but finite) amount of time.

**Partition** **Tolerance** - The system continues to operate despite arbitrary partitioning due to network failures. The system will continue to function when network partitions occur.

Networks aren't reliable, so you'll need to support partition tolerance. You'll need to make a software tradeoff between consistency and availability.

What is a **partition**?

A partition is when the network fails to deliver some messages to one or more nodes by losing them (not by delaying them - eventual delivery is not a partition).

The term is sometimes used to refer to a period during which no messages are delivered between two sets of nodes. This is a more restrictive failure model. We'll call these kinds of partitions total partitions.

The proof of CAP relied on a total partition. In practice, these are arguably the most likely since all messages may flow through one component; if that fails then message loss is usually total between two nodes.

CAP theorem is applied to asynchronous network. An **asynchronous network** is one in which there is no bound on how long messages may take to be delivered by the network or processed by a machine. The important consequence of this property is that there's no way to distinguish between a machine that has failed, and one whose messages are getting delayed.

**Why is CAP true?**

The basic idea is that if a client writes to one side of a partition, any reads that go to the other side of that partition can't possibly know about the most recent write. Now you're faced with a choice: do you respond to the reads with potentially stale information, or do you wait (potentially forever) to hear from the other side of the partition and compromise availability?

You have many servers and they are connected via networks. Networks are not reliable. When networks go down, According to the CAP theorem we can only have either **consistency or availability.**

CAP is better understood as describing the tradeoffs you have to make when you are building a system that may suffer partitions. In practice, this is every distributed system: there is no 100% reliable network. So (at least in the distributed context) there is no realistic CA system. You will potentially suffer partitions, therefore you must at some point compromise C or A.

There are some systems that won't experience partitions - single-site databases, for example. These systems aren't generally relevant to the contexts in which CAP is most useful. If you describe your distributed database as 'CA', you are misunderstanding something.

**CP - Consistency/Partition Tolerance** - Wait for a response from the partitioned node which could result in a timeout error. The system can also choose to return an error, depending on the scenario you desire. You will not return stale data to maintain consistency. If you can’t get the data due to a dead node, you will send an error instead of sending older data from some other node. Choose **Consistency over Availability** when your **business requirements dictate atomic reads and writes.**

**AP - Availability/Partition Tolerance** - Return the most recent version of the data you have, which could be stale. This system state will also accept writes that can be processed later when the partition is resolved. **Choose Availability over Consistency when your business requirements allow for some flexibility around when the data in the system synchronizes.** Availability is also a compelling option when the system needs to continue to function in spite of external errors (shopping carts, etc.) Or when the system allows for **eventual consistency.**

The decision between Consistency and Availability is a software trade off. You can choose what to do in the face of a network partition - the control is in your hands. Network outages, both temporary and permanent, are a fact of life and occur whether you want them to or not - this exists outside of your software.

Building distributed systems provide many advantages, but also adds complexity. Understanding the trade-offs available to you in the face of network errors, and choosing the right path is vital to the success of your application. Failing to get this right from the beginning could doom your application to failure before your first deployment.

**Consistency patterns**

With multiple copies of the same data, we are faced with options on how to synchronize them so clients have a consistent view of the data. Recall the definition of consistency from the CAP theorem - Every read receives the most recent write or an error.

**Weak consistency**

After a write, reads may or may not see it. A best effort approach is taken.

This approach is seen in systems such as **memcached**. **Weak consistency works well in real time use cases such as VoIP, video chat, and realtime multiplayer games**. For example, if you are on a phone call and lose reception for a few seconds, when you regain connection you do not hear what was spoken during connection loss.

**Eventual consistency**

After a write, reads will eventually see it (typically within milliseconds). Data is replicated **asynchronously**.

**This approach is seen in systems such as DNS and email. Eventual consistency works well in highly available systems.**

**Strong consistency**

After a write, reads will see it. Data is replicated **synchronously**.

**This approach is seen in file systems and RDBMSes. Strong consistency works well in systems that need transactions**

**Availability patterns**

There are two complementary patterns to support high availability: fail-over and replication.

**Fail-over**

**Active-passive**

With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service.

The length of downtime is determined by whether the passive server is already running **in 'hot' standby** or whether it needs to start up from **'cold' standby.** Only the active server handles traffic.

Active-passive failover can also be referred to as **master-slave failover.**

**Active**-**active**

In active-active, both servers are managing traffic, spreading the load between them.

If the servers are public-facing, the DNS would need to know about the public IPs of both servers. If the servers are internal-facing, application logic would need to know about both servers.

Active-active failover can also be referred to as **master-master failover**.

Disadvantage(s): failover

* Fail-over adds more hardware and additional complexity.
* There is a potential for loss of data if the active system fails before any newly written data can be replicated to the passive.

**Replication**

Master-slave and master-master

This topic is further discussed in the Database section:

**Master-slave replication**

**Master-master replication**

**Availability in numbers**

Availability is often quantified by uptime (or downtime) as a percentage of time the service is available. Availability is generally measured in number of 9s--a service with 99.99% availability is described as having four 9s.

**99.9% availability - three 9s**

Duration Acceptable downtime

Downtime per year 8h 45min 57s

Downtime per month 43m 49.7s

Downtime per week 10m 4.8s

Downtime per day 1m 26.4s

**99.99% availability - four 9s**

Duration Acceptable downtime

Downtime per year 52min 35.7s

Downtime per month 4m 23s

Downtime per week 1m 5s

Downtime per day 8.6s

**Availability in parallel vs in sequence**

If a service consists of multiple components prone to failure, the service's overall availability depends on whether the components are in sequence or in parallel.

**In sequence**

Overall availability decreases when two components with availability < 100% are in sequence:

**Availability (Total) = Availability (Foo) \* Availability (Bar)**

If both Foo and Bar each had 99.9% availability, their total availability in sequence would be 99.8%.

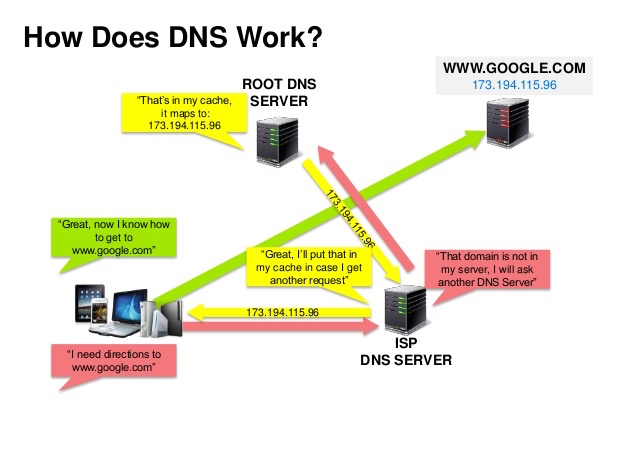
**In parallel**

Overall availability increases when two components with availability < 100% are in parallel:

**Availability (Total) = 1 - (1 - Availability (Foo)) \* (1 - Availability (Bar))**

If both Foo and Bar each had 99.9% availability, their total availability in parallel would be 99.9999%.

**DNS Server:**



A **Domain Name System (DNS)** translates a domain name such as www.example.com to an IP address.

The [Domain Name System (DNS)](https://www.oracle.com/cloud/networking/dns/what-is-dns/) is a distributed internet system that maps human-readable names (such as Oracle.com) to IP addresses and serves as the first link in the customers’ digital supply chain.

**DNS** is hierarchical, with a few authoritative servers at the top level. Your router or ISP provides information about which DNS server(s) to contact when doing a lookup. Lower level DNS servers cache mappings, which could become stale due to DNS propagation delays. DNS results can also be cached by your browser or OS for a certain period of time, determined by the time to live [time to live (TTL)](https://en.wikipedia.org/wiki/Time_to_live).

**NS record (name server)** - Specifies the DNS servers for your domain/subdomain.

**MX record (mail exchange)** - Specifies the mail servers for accepting messages.

**A record (address)** - Points a name to an IP address.

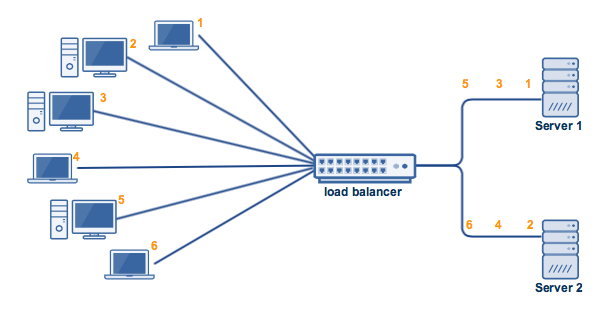
**CNAME** (**canonical**) - Points a name to another name or CNAME (example.com to [www.example.com](http://www.example.com/)) or to an A record.

Services such as [CloudFlare](https://www.cloudflare.com/dns/) and [Route 53](https://aws.amazon.com/route53/) provide managed DNS services. Some DNS services can route traffic through various methods:

**Load Balancing Algorithms:**

**Robin Round:** Round Robin is undoubtedly the most widely used algorithm. It's easy to implement and easy to understand. Here's how it works. Let's say you have 2 servers waiting for requests behind your load balancer. Once the first request arrives, the load balancer will forward that request to the 1st server. When the 2nd request arrives (presumably from a different client), that request will then be forwarded to the 2nd server.

Because the 2nd server is the last in this cluster, the next request (i.e., the 3rd) will be forwarded back to the 1st server, the 4th request back to the 2nd server, and so on, in a cyclical fashion.



As you can see, the method is very simple. However, it won't do well in certain scenarios.

For example, what if Server 1 had more CPU, RAM, and other specs compared to Server 2? Server 1 should be able to handle a higher workload than Server 2, right?

Unfortunately, a load balancer running on a round robin algorithm won't be able to treat the two servers accordingly. In spite of the two servers' disproportionate capacities, the load balancer will still distribute requests equally. As a result, Server 2 can get overloaded faster and probably even go down. You wouldn't want that to happen.

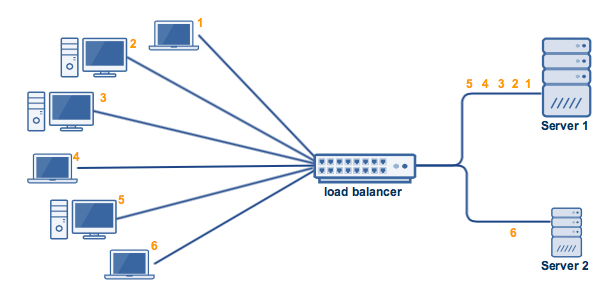
The Round Robin algorithm is best for clusters consisting of servers with identical specs. For other situations, you might want to look at other algorithms, like the ones below.

**Weighted Round Robin:** For the 2nd scenario mentioned above, i.e., Server 1 having higher specs than Server 2, you might prefer an algorithm that assigns more requests to the server with a higher capability of handling greater load. One such algorithm is the Weighted Round Robin.

The Weighted Round Robin is similar to the Round Robin in a sense that the manner by which requests are assigned to the nodes is still cyclical, albeit with a twist. The node with the higher specs will be apportioned a greater number of requests.

But how would the load balancer know which node has a higher capacity? Simple. You tell it beforehand. Basically, when you set up the load balancer, you assign "weights" to each node. The node with the higher specs should of course be given the higher weight.

You usually specify weights in proportion to actual capacities. So, for example, if Server 1's capacity is 5x more than Server 2's, then you can assign it a weight of 5 and Server 2 a weight of 1.



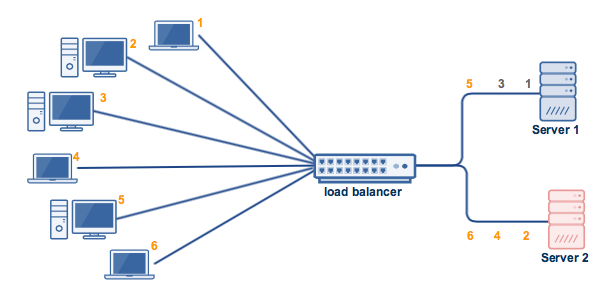
So when clients start coming in, the first 5 will be assigned to node 1 and the 6th to node 2. If more clients come in, the same sequence will be followed. That is, the 7th, 8th, 9th, 10, and 11th will all go to Server1, and the 12th to Server 2, and so on.

Capacity isn't the only basis for choosing the Weighted Round Robin (WRR) algorithm. Sometimes, you'll want to use it if say you want one server to get a substantially lower number of connections than an equally capable server for the reason that the first server is running business-critical applications and you don't want it to be easily overloaded.

**Least Connection:**

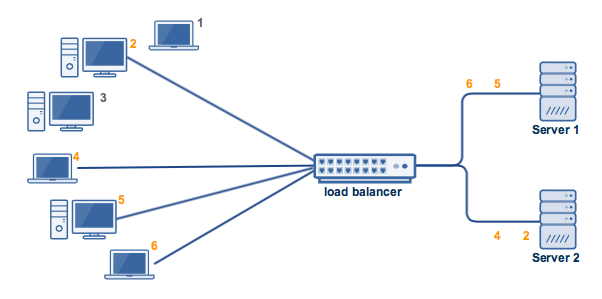
There can be instances when, even if two servers in a cluster have exactly the same specs (see first example/figure), one server can still get overloaded considerably faster than the other. One possible reason would be because clients connecting to Server 2 stay connected much longer than those connecting to Server 1.

This can cause the total current connections in Server 2 to pile up, while those of Server 1 (with clients connecting and disconnecting over shorter times) would virtually remain the same. As a result, Server 2's resources can run out faster. This is depicted below, wherein clients 1 and 3 already disconnect, while 2, 4, 5, and 6 are still connected.



In situations like this, the Least Connections algorithm would be a better fit. This algorithm takes into consideration the number of current connections each server has. When a client attempts to connect, the load balancer will try to determine which server has the least number of connections and then assign the new connection to that server.

So if say (continuing our last example), client 6 attempts to connect after 1 and 3 have already disconnected but 2 and 4 are still connected, the load balancer will assign client 6 to Server 1 instead of Server 2.



**Weighted Least Connections**

The Weighted Least Connections algorithm does to Least Connections what Weighted Round Robin does to Round Robin. That is, it introduces a "weight" component based on the respective capacities of each server. Just like in the Weighted Round Robin, you'll have to specify each server's "weight" beforehand.

A load balancer that implements the Weighted Least Connections algorithm now takes into consideration two things: the weights/capacities of each server AND the current number of clients currently connected to each server.

**Random**

As its name implies, this algorithm matches clients and servers by random, i.e. using an underlying random number generator. In cases wherein the load balancer receives a large number of requests, a Random algorithm will be able to distribute the requests evenly to the nodes. So like Round Robin, the Random algorithm is sufficient for clusters consisting of nodes with similar configurations (CPU, RAM, etc).

There are some polices which you can use with AWS Route 53:

**Simple routing policy** – Use for a single resource that performs a given function for your domain, for example, a web server that serves content for the example.com website. You can use simple routing to create records in a private hosted zone.

**Failover routing policy** – Use when you want to configure active-passive failover. You can use failover routing to create records in a private hosted zone.

**Geolocation routing policy** – Use when you want to route traffic based on the location of your users. You can use geolocation routing to create records in a private hosted zone.

**Geoproximity routing policy** – Use when you want to route traffic based on the location of your resources and, optionally, shift traffic from resources in one location to resources in another location. You can use geoproximity routing to create records in a private hosted zone.

**Latency routing policy** – Use when you have resources in multiple AWS Regions and you want to route traffic to the Region that provides the best latency. You can use latency routing to create records in a private hosted zone.

**IP-based routing policy** – Use when you want to route traffic based on the location of your users, and have the IP addresses that the traffic originates from.

**Multivalue answer routing policy** – Use when you want Route 53 to respond to DNS queries with up to eight healthy records selected at random. You can use multivalue answer routing to create records in a private hosted zone.

**Weighted routing policy** – Use to route traffic to multiple resources in proportions that you specify. You can use weighted routing to create records in a private hosted zone.

You your website has multiple servers and hence multiple IP addresses which the domain name can map to, then you can use any of the above routing techniques.

Disadvantage(s):

1. Accessing a DNS server introduces a slight delay, although mitigated by caching described above.
2. DNS server management could be complex and is generally managed by [governments, ISPs, and large companies](http://superuser.com/questions/472695/who-controls-the-dns-servers/472729).
3. DNS services have recently come under [DDoS attack](http://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/), preventing users from accessing websites such as Twitter without knowing Twitter's IP address(es).

**Content Delivery Network (CDN):**



A **content delivery network (CDN)** is a globally distributed network of proxy servers, serving content from locations closer to the user. Generally, static files such as HTML/CSS/JS, photos, and videos are served from CDN, although some CDNs such as Amazon's CloudFront support dynamic content. The site's DNS resolution will tell clients which server to contact.

You store your static files in the CDN and when users request these files, they are fetched from the CDN instead of the server. This reduces the latency and reduces the load on your servers.

Serving content from CDNs can significantly improve performance in two ways:

* Users receive content from data centers close to them
* Your servers do not have to serve requests that the CDN fulfills

**Push CDNs:** You push data onto the CDNs yourself. Once pushed, the data is available and will be used by requests asking for this data. Of course, you have to rewrite the URLs of such data.

Push CDNs receive new content whenever changes occur on your server. You take full responsibility for providing content, uploading directly to the CDN and rewriting URLs to point to the CDN. You can configure when content expires and when it is updated. Content is uploaded only when it is new or changed, minimizing traffic, but maximizing storage.

Sites with a small amount of traffic or sites with content that isn't often updated work well with push CDNs. Content is placed on the CDNs once, instead of being re-pulled at regular intervals.

If you don’t make much changes to your data, you can put your data onto push CDNs.

 a push CDN can put added strain on your server if it’s underpowered for your traffic, or you have lots of changing content in a given day. The reason being, pushing all of your data and any changes as they happen to the CDN takes work on your server’s part. If your server is already struggling under heavy load ([here are a few tips to optimize your site](http://travelblogadvice.com/technical/basic-ways-to-reduce-your-travel-blogs-loading-time/)) or has new content several times a day, all of them syncing between your server and the CDN might do more harm than good.

**Pull CDNs:** You do not push the data onto the CDN. Whenever a request comes to the content configured to be on the CND, the data is fetched from the server and pulled onto the CDN. So, the first request will not see any latency difference. After the first request the data will be in the CDN with a TTL (Time to Live). After that if any request needs this data, they will get it from the CDN.

A [time-to-live (TTL)](https://en.wikipedia.org/wiki/Time_to_live) determines how long content is cached. Pull CDNs minimize storage space on the CDN, but can create redundant traffic if files expire and are pulled before they have actually changed.

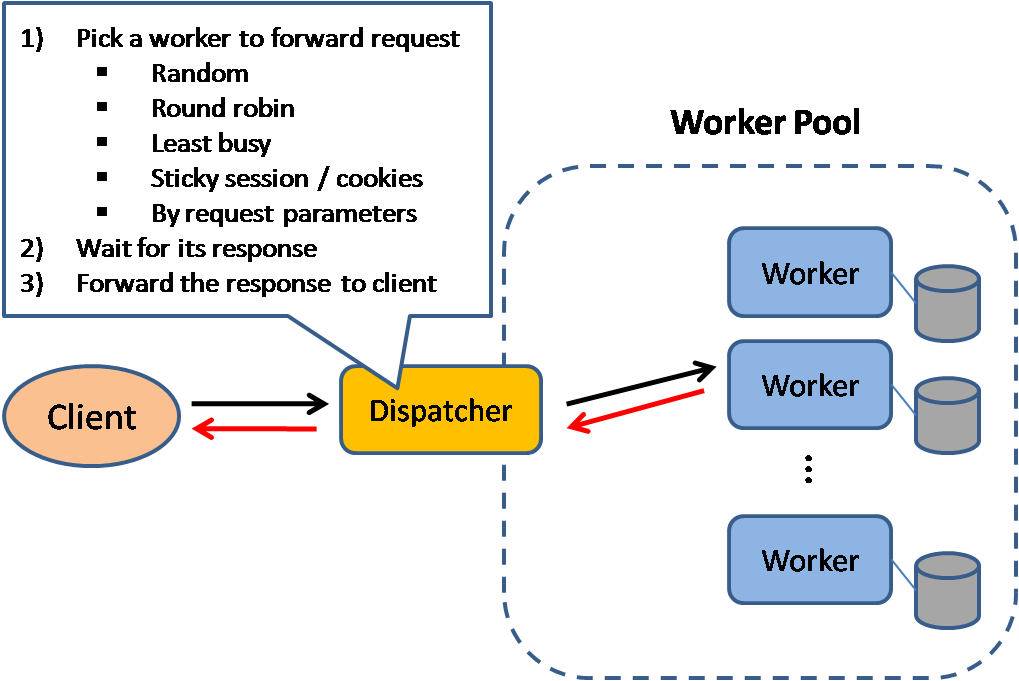
 A pull CDN is much easier to configure than a push CDN. Once initially configured, a pull CDN rather seamlessly stores and updates content on its servers as its requested. The data usually stays there for 24 hours or longer if the CDN doesn’t detect that a file has been modified. For low traffic sites or those that are sufficiently optimized with caching, good code, and more, a pull CDN provides speed without asking much of your server. Once your content is pulled (give it 48 hours to get enough data to make it a noticeable difference) the maintenance required is low.

So, what makes a pull CDN so easy can also be a pain, especially when you’re making changes to your travel blog. Typically, you don’t have control over how long the pull CDN cache lasts, so if you update a photo or theme, it might take up to 24 hours for all of your readers (and you) to see it. You lose control for ease so when it comes to making widespread changes like updating your theme, you often have to shut off the CDN during the process.

Disadvantage(s): CDN

* CDN costs could be significant depending on traffic, although this should be weighed with additional costs you would incur not using a CDN.
* Content might be stale if it is updated before the TTL expires it.
* CDNs require changing URLs for static content to point to the CDN.

## Load Balancing:



**Load balancers** distribute incoming client requests to computing resources such as **application servers and databases**. In each case, the load balancer returns the response from the computing resource to the appropriate client. Load balancers are effective at:

* Preventing requests from going to unhealthy servers
* Preventing overloading resources
* Helping to eliminate a single point of failure

**Load balancers can be implemented with hardware (expensive) or with software such as HAProxy.**

**Additional benefits include:**

* **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations. Removes the need to install [X.509 certificates](https://en.wikipedia.org/wiki/X.509) on each server
* **Session persistence** - Issue cookies and route a specific client's requests to same instance if the web apps do not keep track of sessions

To protect against failures, it's common to set up multiple load balancers, either in [active-passive](https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#active-passive) or [active-active](https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#active-active) mode.

**Fail-over**

**Active-passive**

With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service.

The length of downtime is determined by whether the passive server is already running **in 'hot' standby** or whether it needs to start up from **'cold' standby.** Only the active server handles traffic.

Active-passive failover can also be referred to as **master-slave failover.**

**Active**-**active**

In active-active, both servers are managing traffic, spreading the load between them.

If the servers are public-facing, the DNS would need to know about the public IPs of both servers. If the servers are internal-facing, application logic would need to know about both servers.

Active-active failover can also be referred to as **master-master failover**.

Load balancers can route traffic based on various metrics, including:

* Random
* Least loaded
* Session/cookies
* [Round robin or weighted round robin](https://www.g33kinfo.com/info/round-robin-vs-weighted-round-robin-lb)
* [Layer 4](https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#layer-4-load-balancing)
* [Layer 7](https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#layer-7-load-balancing)

**Layer 4 load balancing**

Layer 4 load balancers look at info at the [transport layer](https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#communication) to decide how to distribute requests. Generally, this involves the source, destination IP addresses, and ports in the header, but not the contents of the packet. Layer 4 load balancers forward network packets to and from the upstream server, performing [Network Address Translation (NAT)](https://www.nginx.com/resources/glossary/layer-4-load-balancing/).

When the Layer 4 load balancer receives a request and makes the load balancing decision, it also performs **Network Address Translation (NAT)** on the request packet, changing the recorded destination IP address from its own to that of the content server it has chosen on the internal network. Similarly, before forwarding server responses to clients, the load balancer changes the source address recorded in the packet header from the server’s IP address to its own. (The destination and source TCP port numbers recorded in the packets are sometimes also changed in a similar way.)

Layer 4 load balancers make their routing **decisions based on address information** extracted from the **first few packets in the TCP stream**, and do not inspect packet content. A Layer 4 load balancer is often a **dedicated hardware device** supplied by a vendor and runs proprietary load-balancing software, and the NAT operations might be performed by specialized chips rather than in software.

Layer 4 load balancing was a popular architectural approach to traffic handling when commodity hardware was not as powerful as it is now, and the interaction between clients and application servers was much less complex. It requires less computation than more sophisticated load balancing methods (such as Layer 7), but CPU and memory are now sufficiently fast and cheap that the performance advantage for Layer 4 load balancing has become negligible or irrelevant in most situations.

**Layer 7 load balancing**

Layer 7 load balancers look at the [**application layer**](https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#communication) to decide how to distribute requests. This can involve **contents of the header, message, and cookies.** Layer 7 load balancers terminate network traffic, reads the message, makes a load-balancing decision, then opens a connection to the selected server. For example, a layer 7 load balancer can direct video traffic to servers that host videos while directing more sensitive user billing traffic to security-hardened servers.

At the cost of flexibility, layer 4 load balancing requires less time and computing resources than Layer 7, **although the performance impact can be minimal on modern commodity hardware.**

Layer 7 load balancing operates at the **high‑level application layer**, which deals with the actual content of each message. HTTP is the predominant Layer 7 protocol for website traffic on the Internet. Layer 7 load balancers route network traffic in a much more sophisticated way than Layer 4 load balancers, particularly applicable to TCP‑based traffic such as HTTP. A Layer 7 load balancer terminates the network traffic and reads the message within. It can make a load‑balancing decision based on the content of the message (the URL or cookie, for example). It then makes a new TCP connection to the selected upstream server (or reuses an existing one, by means of [HTTP keepalives](https://www.nginx.com/blog/http-keepalives-and-web-performance/)) and writes the request to the server.

**Horizontal scaling**

Load balancers can also help with horizontal scaling, improving performance and availability. Scaling out using commodity machines is more cost efficient and results in higher availability than scaling up a single server on more expensive hardware, called Vertical Scaling. It is also easier to hire for talent working on commodity hardware than it is for specialized enterprise systems.

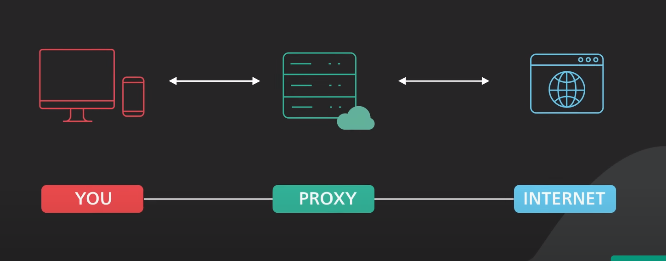
**Disadvantage(s): horizontal scaling**

* Scaling horizontally introduces complexity and involves cloning servers
* Servers should be **stateless**: they should not contain any user-related data like sessions or profile pictures. Sessions can be stored in a centralized data store such as a [database](https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#database) (SQL, NoSQL) or a persistent [cache](https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#cache) (Redis, Memcached)
* Downstream servers such as caches and databases need to handle more simultaneous connections as upstream servers scale out.

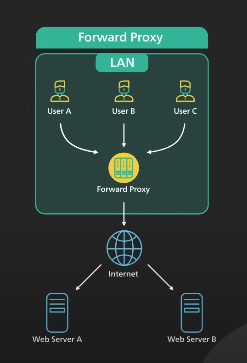
**Disadvantage(s): load balancer**

* The load balancer can become a performance bottleneck if it does not have enough resources or if it is not configured properly.
* Introducing a load balancer to help eliminate a single point of failure results in increased complexity.
* A single load balancer is a single point of failure, configuring multiple load balancers further increases complexity.

## Proxy:

****

**Forward proxy:** It is a server that sits between client machines and the internet. When a request is made by a client, the forward proxy intercepts those requests and talks to web servers on behalf of those client machines.



Why use it?

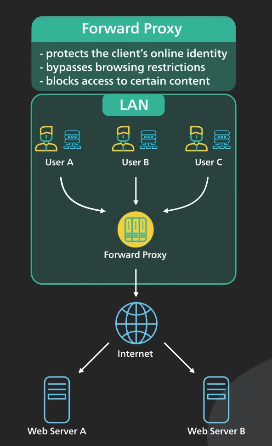
It hides the IP address of the client. Server will see the IP address of the proxy server. It is harder to trace the client.

It is also helpful to bypass browsing restrictions. Governments, schools, organization may use firewall to restrict access to the internet. By connecting to the forward proxy outside the firewall the client machine can get away with the restrictions. But it may not always work because firewall may even block these proxies.

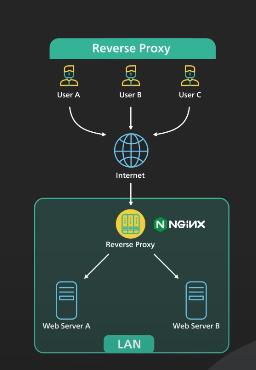
It can use used to block access to certain websites. Schools, govt, organizations may configure for the internet to be accessed through the proxy only. With this, they may block certain sites.

Usually, a forward proxy needs to be configured by client machine.

**Forward proxy acts as the client.**



**Reverse Proxy:** Reverse proxy sits between the internet and the web servers. It intercepts the requests from the client and talks to the web server on behalf of the client.



Why would a website use a reverse proxy?

It could be used to protect a website. Websites IP addresses such as server’s web address are hidden behind the reverse proxy and are not revealed to the clients. This makes it much harder to target a **DDoS attack** again the website. Hide information about backend servers, blacklist IPs, limit number of connections per client.

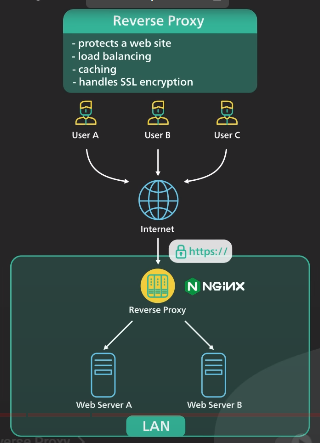
A Distributed Denial of Service (DDoS) attack is designed to force a website, computer, or online service offline. This is accomplished by flooding the target with many requests, consuming its capacity and rendering it unable to respond to legitimate requests.

**Increased scalability and flexibility** - Clients only see the reverse proxy's IP, allowing you to scale servers or change their configuration.

Reverse proxy is used for **load balancing.** It will distribute the load among many web servers. Services like **Cloudflare** put reverse proxy servers in hundreds of locations around the world.

Reverse proxy can be used to serve static content. A piece of content can be cached on the reverse proxy for a period of time. This makes the website faster. You can put static content such as pictures, videos, HTML, CSS, JS content etc. on reverse proxy. This is similar to CDN. This reduces the load on the servers.

A reverse proxy can handle **SSL encryption.** **SSL handshakes** can be handled by the reverse proxy. This frees up the servers. Decrypt incoming requests and encrypt server responses, so, backend servers do not have to perform these potentially expensive operations. This Removes the need to install **X.509 certificates** on each server.



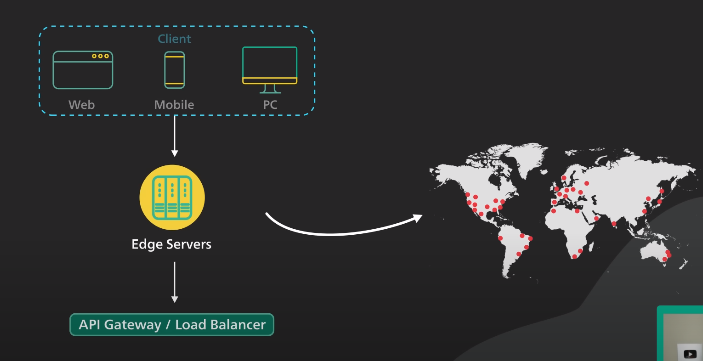
**Cloudflare is an example of reverse proxy.**

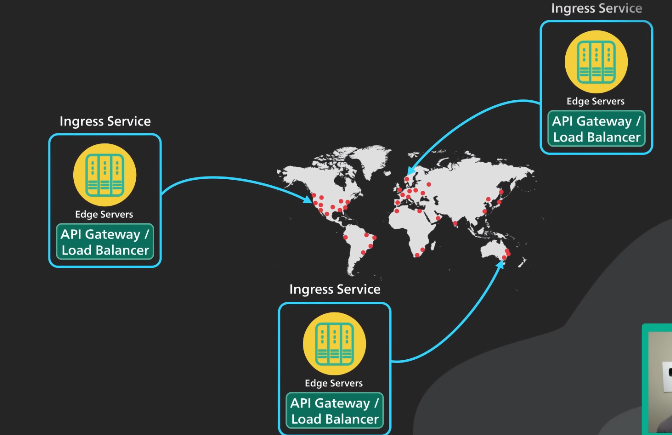


All these sites use revere proxy.

We can have many layers of reverse proxy. First layer can be an **edge servers** such as **Cloudflare.** Reverse proxies are deployed to hundreds of locations around the world closer to the user. Second layer can be a **load balancer/API gateway.** Many cloud providers combine these two together into a single service called **Ingress Service.**

When user sends a request, it will go to the closest **edge server** closest to him. From the edge server, the reverse proxy connects to the **load balancer/API gateway** over a fast fiber network. Load balancer then evenly distributes the traffic among servers.



****

**Load balancer vs reverse proxy**

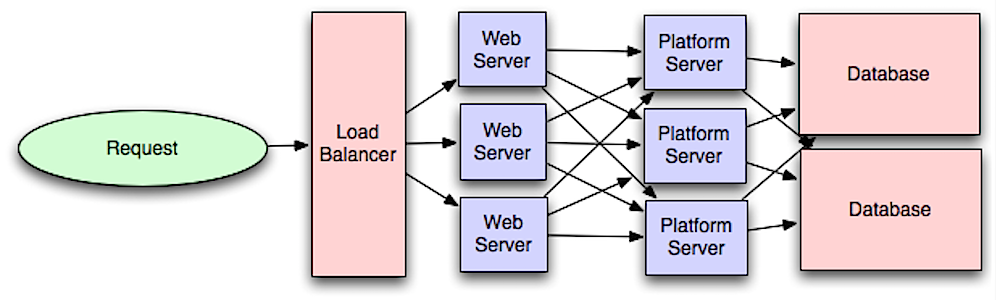
* Deploying a load balancer is useful when you have multiple servers. Often, load balancers route traffic to a set of servers serving the same function.
* Reverse proxies can be useful even with just one web server or application server, opening up the benefits described in the previous section.

**Solutions such as NGINX and HAProxy can support both layer 7 reverse proxying and load balancing.**

Disadvantage(s): reverse proxy

* Introducing a reverse proxy result in increased complexity.
* A single reverse proxy is a single point of failure, configuring multiple reverse proxies (ie a failover) further increases complexity.

## Application Layer:



Separating out the web layer from the **application layer (also known as platform layer)** allows you to scale and configure both layers independently. Adding a new API result in adding application servers without necessarily adding additional web servers. The **single responsibility principal** advocates for small and autonomous services that work together. Small teams with small services can plan more aggressively for rapid growth.

Workers in the application layer also help enable **asynchronism**.

**What are Microservices?**

Microservice architecture is a distinctive method of developing software systems that tries to focus on building **single-function modules** with well-defined interfaces and operations.

The trend has grown in recent years as enterprises look to become more Agile and move towards a DevOps and continuous testing.

Microservices have many benefits for Agile and DevOps teams - as Martin Fowler points out, **Netflix, eBay, Amazon, Twitter, PayPal**, and other tech stars have all evolved from monolithic to microservices architecture.

**The distinction**

Unlike **microservices**, a **monolith application** is built as a **single**, **autonomous** unit. So, any changes to the application are slow, as it affects the entire system. A modification made to a small section of code might require building and deploying an entirely new version of software. So, to scale a specific function of an application also means you have to scale the entire application.

**Modularity is the key**

**Microservices** solve the challenges of **monolithic** systems by being as **modular** as possible. In the simplest form, they help build an application as a suite of small services, each running in its own process and being independently deployable. These services may be written in different programming languages and may even use different data storage techniques.

While this makes things more scalable and flexible, it needs a dynamic makeover. **Microservices** are often connected via **APIs** and can leverage many of the same tools and solutions that have grown in the **RESTful** and **web service ecosystem**. Testing these APIs is now non-negotiable to ensure quality software deployments. It works by validating the communication paths and flow of data throughout your microservice deployment.

**Benefits Of Microservices:**

**Simpler to deploy:** Deploy in literal pieces without affecting other services.

**Simpler to understand:** Code is easier to follow since the function is isolated and less dependent.

**Reusable across business:** Share small services like payment or login systems across the business.

**Faster defect isolation:** When a test fails or service goes down, isolate it quickly.

**Minimized risk from change:** Avoid locking in technologies or languages - change on the fly with minimal risk.

**The Six Characteristics of Microservices:**

1. **Multiple Components:** It can be broken down into multiple component services. Why? So that each service can be deployed, tweaked, and redeployed independently without compromising the integrity of an application. This way, you might only need to change one distinct service instead of redeploying entire applications. But this approach has its downsides, including expensive remote calls (instead of in-process calls), coarser-grained remote APIs, and more complexity when redistributing responsibilities between components.
2. **Built For Business:** The microservices style is usually organized around business capabilities and priorities. Unlike a traditional monolithic development approach – where different teams each have a specific focus on, say, UIs, databases, technology layers, or server-side logic – microservice architecture utilizes cross-functional teams. Each team is responsible to make specific products based on individual services communicating via message bus. In microservices, a team owns the product for its lifetime, as in Amazon’s oft-quoted maxim You build it, you run it.
3. **Simple Routing:** Microservices act somewhat like the classical UNIX system: they receive requests, process them, and generate a response accordingly. This is the opposite to how many other products such, as ESBs (Enterprise Service Buses) work. That’s where high-tech systems for message routing, choreography, and applying business rules are utilized. You could say that microservices have smart endpoints that process information and apply logic, and “dumb pipes” through which the info flows.
4. **Decentralized:** Since microservices involve a variety of technologies, old-school methods of centralized governance aren’t optimal. The microservices community favors decentralized governance so its developers can produce tools that can be used by others to solve the same problems. Just like decentralized governance, microservice architecture favors decentralized data management. Monolithic systems use a single logical database across different applications. In a microservice application, each service usually manages its unique database.
5. **Failure Resistant:** Like a well-rounded child, microservices are designed to cope with failure. Since several diverse services communicate, it’s quite possible that a service could fail (e.g., when the supplier isn’t available). In these instances, the client should allow its neighboring services to function while it gracefully bows out. However, monitoring microservices can help prevent the risk of a failure. For obvious reasons, this requirement adds more complexity to microservices as compared to monolithic systems architecture.
6. **Evolutionary:** It’s an evolutionary design and, again, is ideal for evolutionary systems where you can’t fully anticipate what future devices will access your application. Many applications start based on monolithic architecture, but as several unforeseen requirements surfaced, can be slowly revamped to microservices that interact over an older monolithic architecture through APIs.

**Examples of Microservices**

**Netflix** has a widespread architecture that has evolved from monolithic to SOA. It receives more than one billion calls every day, from more than 800 different types of devices, to its streaming-video API. Each API call then prompts around five additional calls to the backend service.

**Amazon** has also migrated to microservices. They get countless calls from a variety of applications – including applications that manage the web service API as well as the website itself – which would have been simply impossible for their old, two-tiered architecture to handle.

**Pros**

Gives developers the freedom to independently develop and deploy services

Can be developed by a fairly small team

Code for different services can be written in different languages (though many practitioners discourage it)

Easy integration and automatic deployment (using open-source continuous integration tools such as Jenkins, Hudson, etc.)

Easy to understand and modify for developers, thus can help a new team member become productive quickly

Developers can make use of the latest technologies

The code is organized around business capabilities

Starts the web container more quickly, so the deployment is also faster

When change is required in a certain part of the application, only the related service can be modified and redeployed – no need to modify and redeploy the entire application

Better fault isolation: if one microservice fails, the other will continue to work (although one problematic area of a monolith application can jeopardize the entire system)

Easy to scale and integrate with third-party services

No long-term commitment to technology stack

**Cons**

Due to distributed deployment, testing can become complicated and tedious – and often hinders some of the scaling benefits of microservices

Increasing number of services can result in information barriers

Additional complexity as the developers mitigates fault tolerance, network latency, and a variety of message formats as well as load balancing

Being a distributed system, it can result in duplication of effort

When number of services increases, integration and managing whole products can become complicated

Developers have to deal with the additional complexity of a distributed system

Developers have to implement a mechanism of communication between the services

Handling use cases that span more than one service without using distributed transactions is not only tough, it requires cooperation between different teams.

**SOA vs. Microservices:** “Wait a minute,” you may be said, “Isn’t this just another name for SOA?” Service-Oriented Architecture sprung up during the first few years of this century, and microservice architecture (abbreviated by some as MSA) bears a number of similarities.

Traditional SOA, however, is a broader framework and can mean a variety of things. Some microservices advocates reject the SOA tag altogether, while others consider microservices as an ideal, refined form of SOA.

The typical SOA model, for example, usually has more dependent ESBs, with microservices using faster messaging mechanisms. SOA also focuses on imperative programming, whereas microservices architecture focuses on a responsive-actor programming style.

Moreover, SOA models tend to have an outsized relational database, while microservices frequently use NoSQL or micro-SQL databases (which can be connected to conventional databases). But the real difference has to do with the architecture methods used to arrive at an integrated set of services in the first place.

# Introduction to architecting systems for scale:

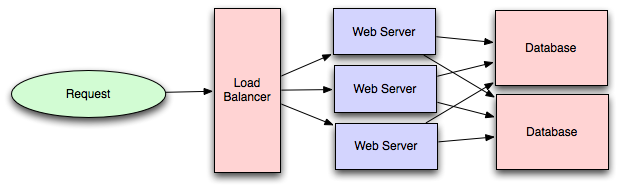


## Load balancing:

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.



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.

## Caching:

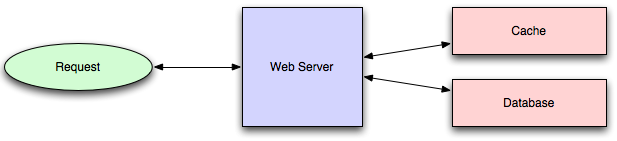
Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible.

Caching consists of: pre-calculating 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.)

## 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):



Code may look like this:

*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:*

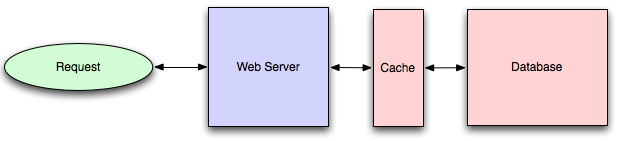
*memcache.set(key, json.dumps(user))*

*return user*

*else:*

*return json.loads(user\_blob)*

**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.

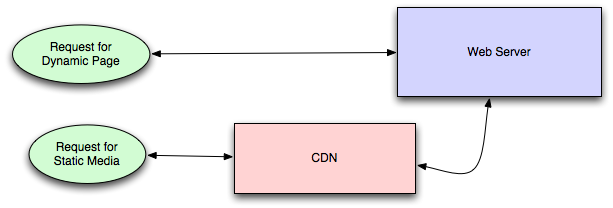
## In-memory caches:

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).

## 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 application 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).\

## Off-line processing:

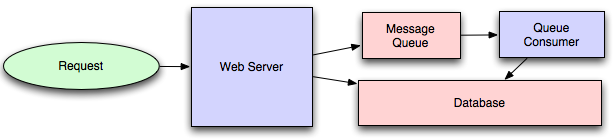
As a system grows more complex, it is almost always necessary to perform processing which can’t be performed in-line with a client’s request either because it creates unacceptable latency (e.g. you want to want to propagate a user’s action across a social graph) or because it needs to occur periodically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

**Message queues:**

For processing you’d like to perform in-line 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).



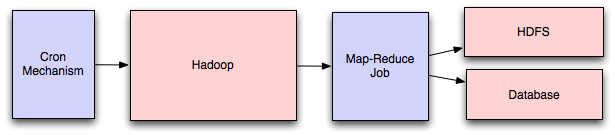
Message queues have another benefit, which is that they allow you to create a separate machine pool for performing off-line processing rather than burdening your web application servers. This allows you to target increases in resources to your current performance or throughput bottleneck rather than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

## Map-reduce:

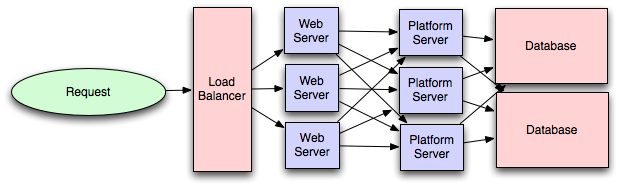
If your large-scale application is dealing with a large quantity of data, at some point you’re likely to add support for map-reduce, probably using Hadoop, and maybe Hive or HBase.



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.

## 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.

# Service-Oriented Architecture (SOA):

In software engineering, service-oriented architecture (SOA) is an architectural style that focuses on discrete services instead of a monolithic design. By consequence, it is also applied in the field of software design where services are provided to the other components by application components, through a communication protocol over a network. A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as retrieving a credit card statement online. SOA is also intended to be independent of vendors, products and technologies.

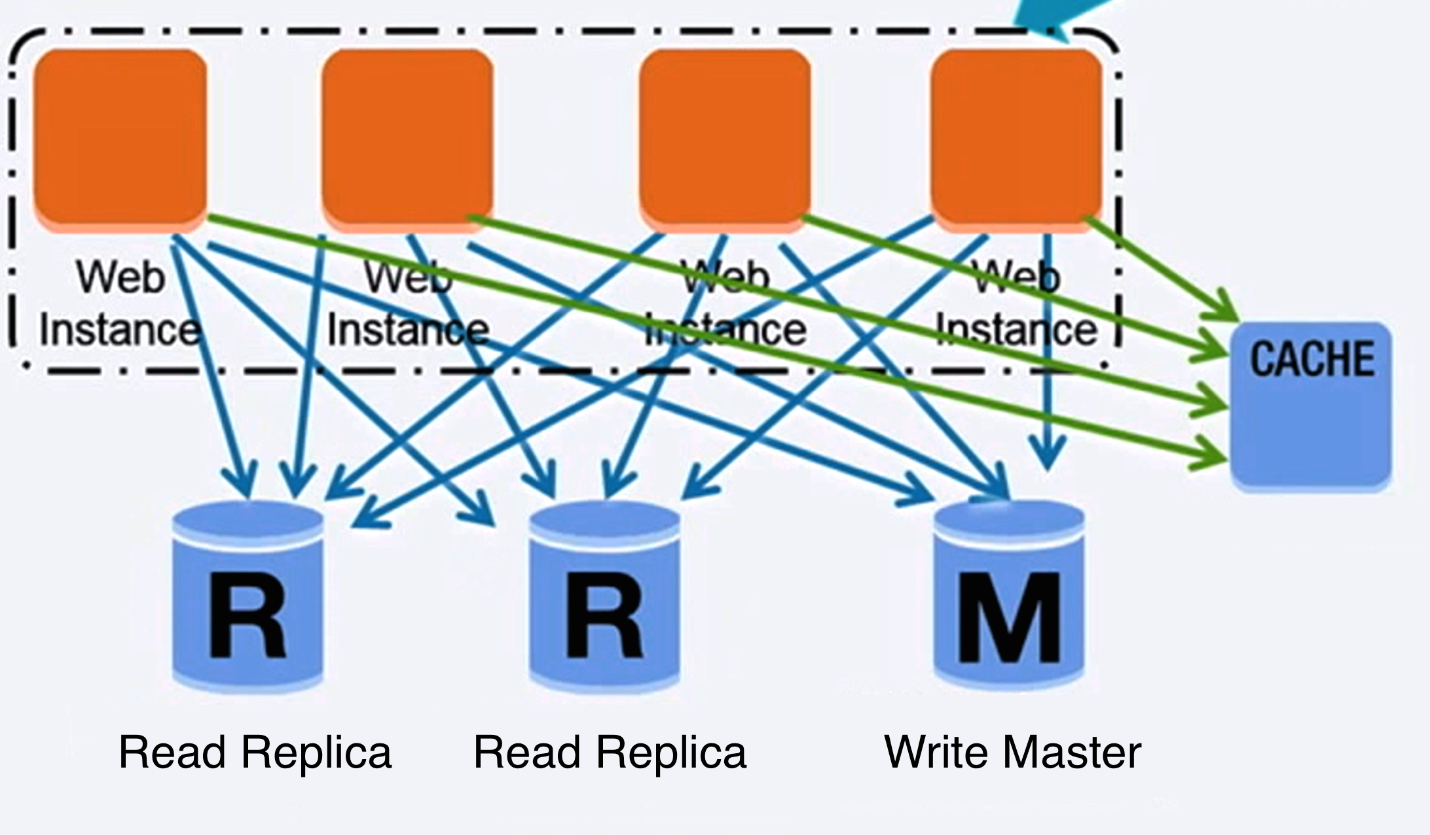
Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

A service has four properties according to one of many definitions of SOA:

* It logically represents a repeatable business activity with a specified outcome.
* It is self-contained.
* It is a black box for its consumers, meaning the consumer does not have to be aware of the service's inner workings.
* It may be composed of other services.

Different services can be used in conjunction as a service mesh to provide the functionality of a large software application, a principle, SOA shares with modular programming. Service-oriented architecture integrates distributed, separately maintained and deployed software components. It is enabled by technologies and standards that facilitate components' communication and cooperation over a network, especially over an IP network.

# Database:



**Relational database management system (RDBMS):**

A relational database like SQL is a collection of data items organized in tables.

**ACID** is a set of properties of relational database transactions.

**Atomicity** - Each transaction is all or nothing

**Consistency** - Any transaction will bring the database from one valid state to another

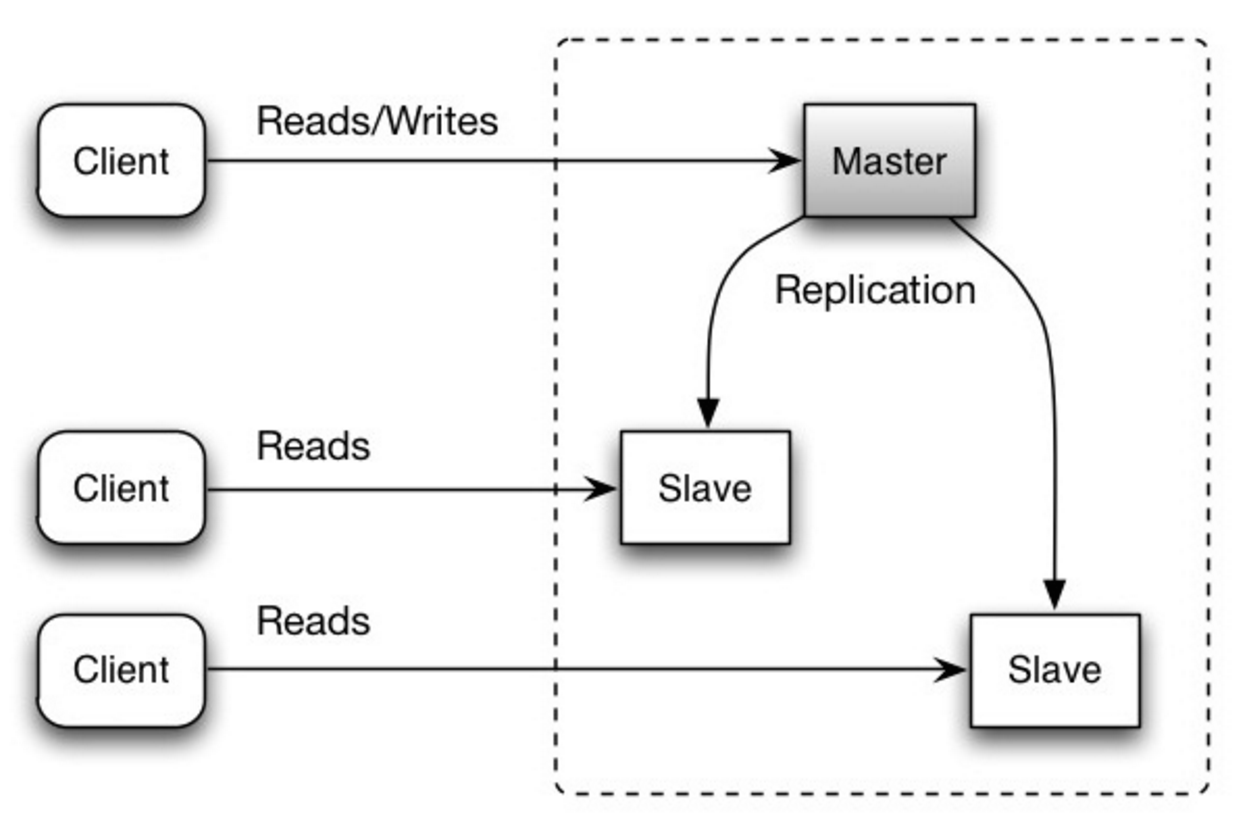
**Isolation** - Executing transactions concurrently has the same results as if the transactions were executed serially

**Durability** - Once a transaction has been committed, it will remain so

There are many techniques to scale a relational database: master-slave replication, master-master replication, federation, sharding, denormalization, and SQL tuning.

**Master-slave replication:**

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate to additional slaves in a tree-like fashion. If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.

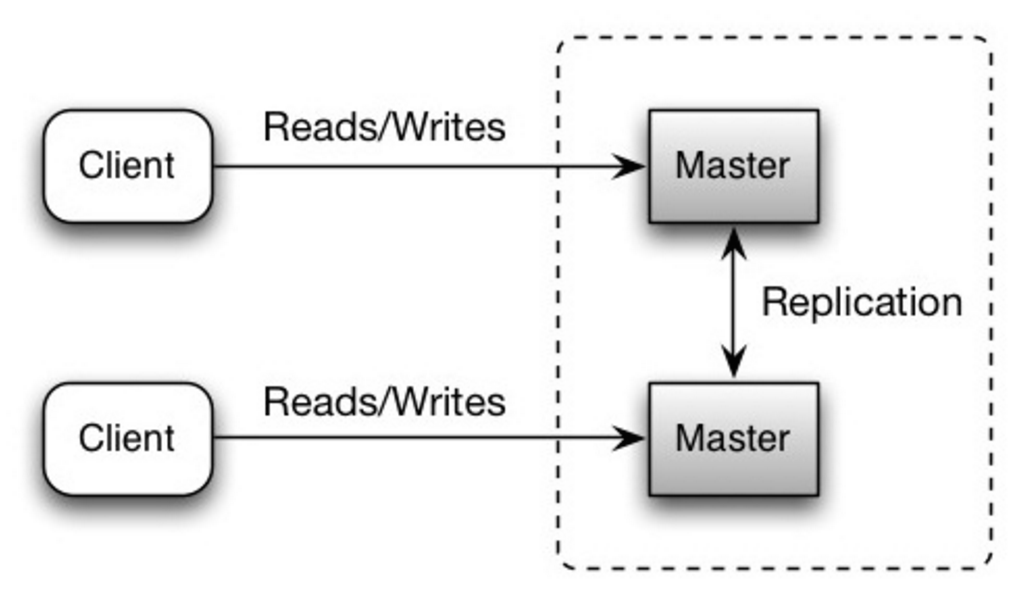


**Disadvantage(s): master-slave replication**

* Additional logic is needed to promote a slave to a master.
* See Disadvantage(s): replication for points related to both master-slave and master-master.

**Master-master replication:**

Both masters serve reads and writes and coordinate with each other on writes. If either master goes down, the system can continue to operate with both reads and writes.



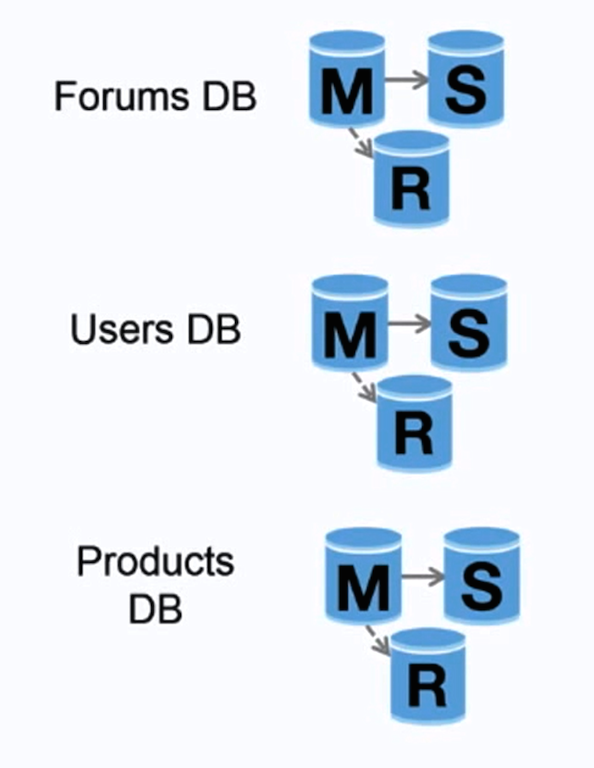
**Disadvantage(s): master-master replication**

* You'll need a load balancer or you'll need to make changes to your application logic to determine where to write.
* Most master-master systems are either loosely consistent (violating ACID) or have increased write latency due to synchronization.
* Conflict resolution comes more into play as more write nodes are added and as latency increases.
* See Disadvantage(s): replication for points related to both master-slave and master-master.

**Disadvantage(s): replication**

* There is a potential for loss of data if the master fails before any newly written data can be replicated to other nodes.
* Writes are replayed to the read replicas. If there are a lot of writes, the read replicas can get bogged down with replaying writes and can't do as many reads.
* The more read slaves, the more you have to replicate, which leads to greater replication lag.
* On some systems, writing to the master can spawn multiple threads to write in parallel, whereas read replicas only support writing sequentially with a single thread.
* Replication adds more hardware and additional complexity.

**Federation:**

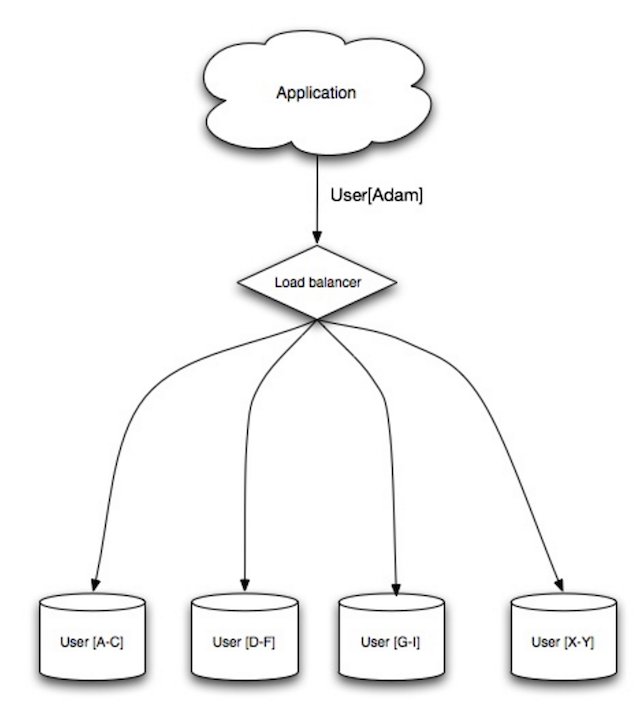


Federation (or functional partitioning) splits up databases by function. For example, instead of a single, monolithic database, you could have three databases: forums, users, and products, resulting in less read and write traffic to each database and therefore less replication lag. Smaller databases result in more data that can fit in memory, which in turn results in more cache hits due to improved cache locality. With no single central master serializing writes you can write in parallel, increasing throughput.

**Disadvantage(s): federation**

* Federation is not effective if your schema requires huge functions or tables.
* You'll need to update your application logic to determine which database to read and write.
* Joining data from two databases is more complex with a server link.
* Federation adds more hardware and additional complexity.

**Sharding:**



Sharding distributes data across different databases such that each database can only manage a subset of the data. Taking a user’s database as an example, as the number of users increases, more shards are added to the cluster.

Similar to the advantages of federation, sharding results in less read and write traffic, less replication, and more cache hits. Index size is also reduced, which generally improves performance with faster queries. If one shard goes down, the other shards are still operational, although you'll want to add some form of replication to avoid data loss. Like federation, there is no single central master serializing writes, allowing you to write in parallel with increased throughput.

Common ways to shard a table of users is either through the user's last name initial or the user's geographic location.

**Disadvantage(s): sharding**

* You'll need to update your application logic to work with shards, which could result in complex SQL queries.
* Data distribution can become lopsided in a shard. For example, a set of power users on a shard could result in increased load to that shard compared to others.
* Rebalancing adds additional complexity. A sharding function based on consistent hashing can reduce the amount of transferred data.
* Joining data from multiple shards is more complex.
* Sharding adds more hardware and additional complexity.

**Denormalization:**

Denormalization attempts to improve read performance at the expense of some write performance. Redundant copies of the data are written in multiple tables to avoid expensive joins. Some RDBMS such as PostgreSQL and Oracle support materialized views which handle the work of storing redundant information and keeping redundant copies consistent.

Once data becomes distributed with techniques such as federation and sharding, managing joins across data centers further increases complexity. Denormalization might circumvent the need for such complex joins.

In most systems, reads can heavily outnumber writes 100:1 or even 1000:1. A read resulting in a complex database join can be very expensive, spending a significant amount of time on disk operations.

**Disadvantage(s): denormalization**

* Data is duplicated.
* Constraints can help redundant copies of information stay in sync, which increases complexity of the database design.
* A denormalized database under heavy write load might perform worse than its normalized counterpart.

**SQL tuning**

SQL tuning is a broad topic and many books have been written as reference.

It's important to benchmark and profile to simulate and uncover bottlenecks.

**Benchmark** - Simulate high-load situations with tools such as ab.

**Profile** - Enable tools such as the slow query log to help track performance issues.

Benchmarking and profiling might point you to the following optimizations.

**Tighten up the schema:**

* MySQL dumps to disk in contiguous blocks for fast access.
* Use CHAR instead of VARCHAR for fixed-length fields.CHAR effectively allows for fast, random access, whereas with VARCHAR, you must find the end of a string before moving onto the next one.
* Use TEXT for large blocks of text such as blog posts. TEXT also allows for boolean searches. Using a TEXT field results in storing a pointer on disk that is used to locate the text block.
* Use INT for larger numbers up to 2^32 or 4 billion.
* Use DECIMAL for currency to avoid floating point representation errors.
* Avoid storing large BLOBS, store the location of where to get the object instead.
* VARCHAR(255) is the largest number of characters that can be counted in an 8 bit number, often maximizing the use of a byte in some RDBMS.
* Set the NOT NULL constraint where applicable to improve search performance.

**Use good indices**

* Columns that you are querying (SELECT, GROUP BY, ORDER BY, JOIN) could be faster with indices.
* Indices are usually represented as self-balancing B-tree that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
* Placing an index can keep the data in memory, requiring more space.
* Writes could also be slower since the index also needs to be updated.
* When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

**Avoid expensive joins**

* Denormalize where performance demands it.

**Partition tables**

* Break up a table by putting hot spots in a separate table to help keep it in memory.

**Tune the query cache**

In some cases, the query cache could lead to performance issues.

# NoSQL

NoSQL is a collection of data items represented in a key-value store, document store, wide column store, or a graph database. Data is denormalized, and joins are generally done in the application code. Most NoSQL stores lack true ACID transactions and favor eventual consistency.

BASE is often used to describe the properties of NoSQL databases. In comparison with the CAP Theorem, BASE chooses availability over consistency.

**Basically available** - the system guarantees availability.

**Soft state** - the state of the system may change over time, even without input.

**Eventual consistency** - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

In addition to choosing between SQL or NoSQL, it is helpful to understand which type of NoSQL database best fits your use case(s). We'll review key-value stores, document stores, wide column stores, and graph databases in the next section.

**Key-value store:**

**Abstraction: hash table**

A key-value store generally allows for O(1) reads and writes and is often backed by memory or SSD. Data stores can maintain keys in lexicographic order, allowing efficient retrieval of key ranges. Key-value stores can allow for storing of metadata with a value.

Key-value stores provide high performance and are often used for simple data models or for rapidly-changing data, such as an in-memory cache layer. Since they offer only a limited set of operations, complexity is shifted to the application layer if additional operations are needed.

A key-value store is the basis for more complex systems such as a document store, and in some cases, a graph database.

**Examples: Redis, Memcached**

**Document store**

**Abstraction: key-value store with documents stored as values**

A document store is centered around documents (XML, JSON, binary, etc), where a document stores all information for a given object. Document stores provide APIs or a query language to query based on the internal structure of the document itself. Note, many key-value stores include features for working with a value's metadata, blurring the lines between these two storage types.

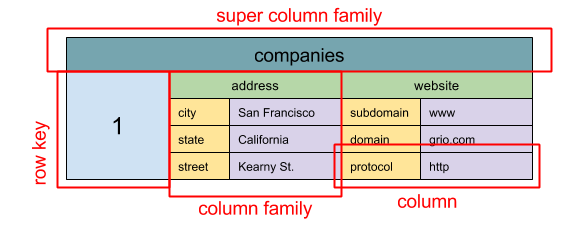
Based on the underlying implementation, documents are organized by collections, tags, metadata, or directories. Although documents can be organized or grouped together, documents may have fields that are completely different from each other.

Some document stores like **MongoDB and CouchDB** also provide a SQL-like language to perform complex queries. DynamoDB supports both key-values and documents.

Document stores provide high flexibility and are often used for working with occasionally changing data.

**Examples: MongoDB and CouchDB**

**Wide column store:**



**Abstraction: nested map ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>**

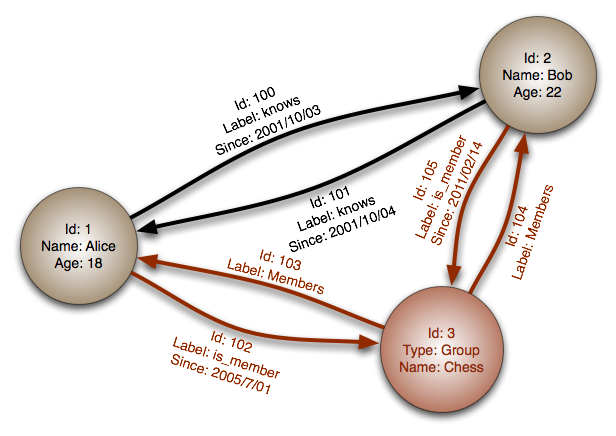
A wide column store's basic unit of data is a column (name/value pair). A column can be grouped in column families (analogous to a SQL table). Super column families further group column families. You can access each column independently with a row key, and columns with the same row key form a row. Each value contains a timestamp for versioning and for conflict resolution.

Google introduced **Bigtable** as the first wide column store, which influenced the open-source **HBase** often-used in the Hadoop ecosystem, and **Cassandra** from Facebook. Stores such as BigTable, HBase, and Cassandra maintain keys in lexicographic order, allowing efficient retrieval of selective key ranges.

Wide column stores offer high availability and high scalability. They are often used for very large data sets.

**Examples: Bigtable, Bigtable, Bigtable**

**Graph database:**



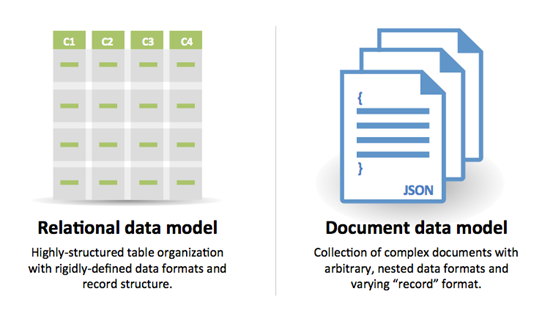
**Abstraction: graph**

In a graph database, each node is a record and each arc is a relationship between two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships.

Graphs databases offer high performance for data models with complex relationships, such as a social network. They are relatively new and are not yet widely-used; it might be more difficult to find development tools and resources. Many graphs can only be accessed with REST APIs.

**Examples: Neo4j, FlockDB**

# SQL or NoSQL:



**Reasons for SQL:**

* Structured data
* Strict schema
* Relational data
* Need for complex joins
* Transactions
* Clear patterns for scaling
* More established: developers, community, code, tools, etc
* Lookups by index are very fast

**Reasons for NoSQL:**

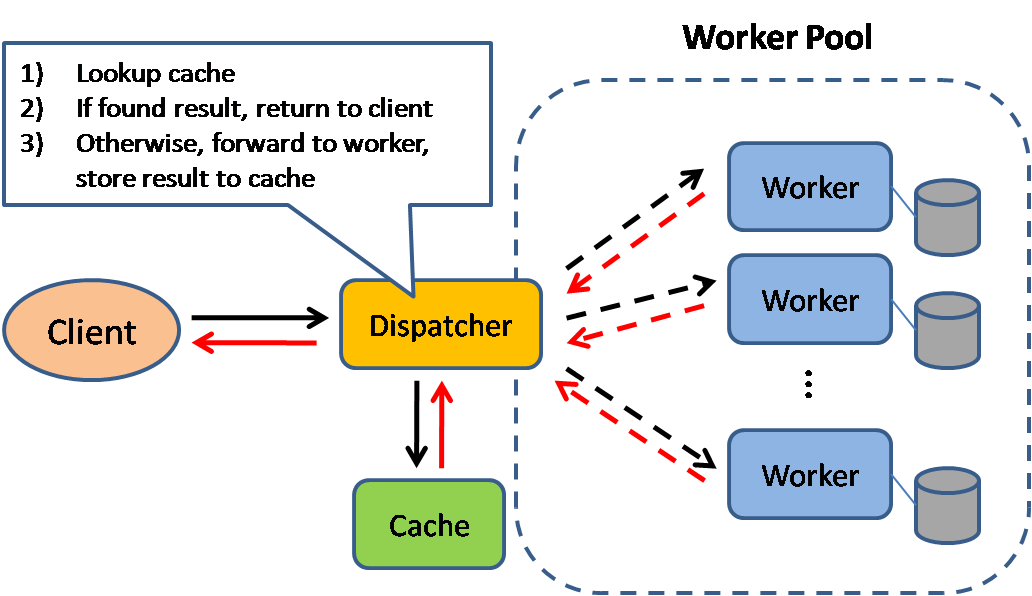
* Semi-structured data
* Dynamic or flexible schema
* Non-relational data
* No need for complex joins
* Store many TB (or PB) of data
* Very data intensive workload
* Very high throughput for IOPS

**Sample data well-suited for NoSQL:**

* Rapid ingest of clickstream and log data
* Leaderboard or scoring data
* Temporary data, such as a shopping cart
* Frequently accessed ('hot') tables
* Metadata/lookup tables

**Must Read:** [**SQL vs NoSQL**](https://www.sitepoint.com/sql-vs-nosql-differences/)

# Cache:



Caching improves page load times and can reduce the load on your servers and databases. In this model, the dispatcher will first lookup if the request has been made before and try to find the previous result to return, in order to save the actual execution.

Databases often benefit from a uniform distribution of reads and writes across its partitions. Popular items can skew the distribution, causing bottlenecks. Putting a cache in front of a database can help absorb uneven loads and spikes in traffic.

**Client caching**

Caches can be located on the client side (OS or browser), server side, or in a distinct cache layer.

**CDN caching**

CDNs are considered a type of cache.

**Web server caching**

Reverse proxies and caches such as Varnish can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers.

**Database caching**

Your database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance.

**Application caching**

In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation algorithms such as least recently used (LRU) can help invalidate 'cold' entries and keep 'hot' data in RAM.

Redis has the following additional features:

* Persistence option
* Built-in data structures such as sorted sets and lists

There are multiple levels you can cache that fall into two general categories: database queries and objects:

* Row level
* Query-level
* Fully-formed serializable objects
* Fully-rendered HTML

Generally, you should try to avoid file-based caching, as it makes cloning and auto-scaling more difficult.

**Caching at the database query level**

Whenever you query the database, hash the query as a key and store the result to the cache. This approach suffers from expiration issues:

* Hard to delete a cached result with complex queries
* If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell

**Caching at the object level**

See your data as an object, similar to what you do with your application code. Have your application assemble the dataset from the database into a class instance or a data structure(s):

* Remove the object from cache if its underlying data has changed
* Allows for asynchronous processing: workers assemble objects by consuming the latest cached object

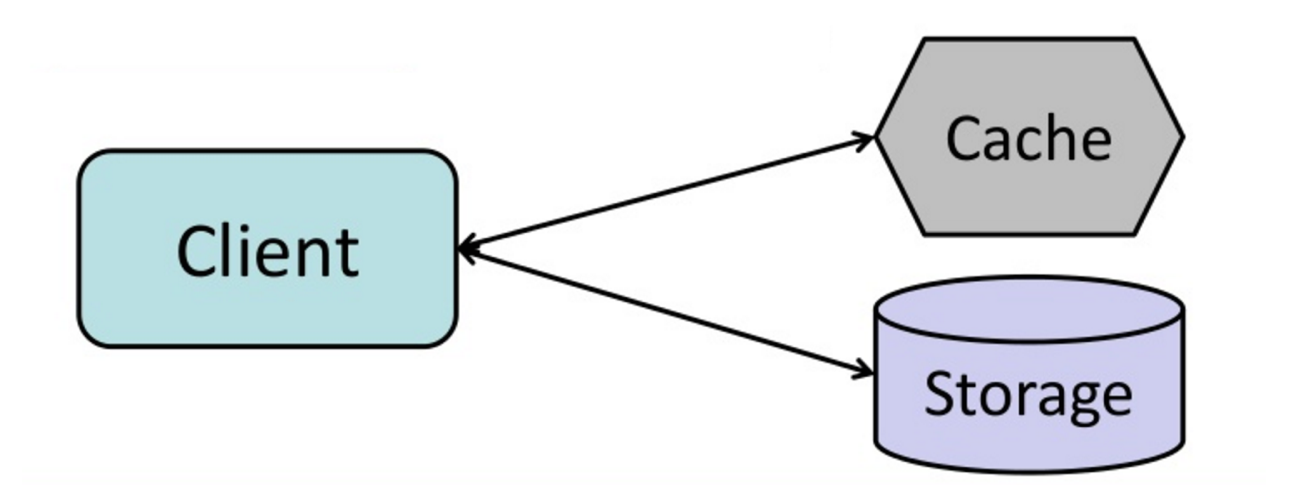
**Suggestions of what to cache:**

* User sessions
* Fully rendered web pages
* Activity streams
* User graph data

## When to update the cache:

Since you can only store a limited amount of data in cache, you'll need to determine which cache update strategy works best for your use case.

**Cache-aside:**



The application is responsible for reading and writing from storage**. The cache does not interact with storage directly.** The application does the following:

1. Look for entry in cache, resulting in a cache miss
2. Load entry from the database
3. Add entry to cache
4. Return entry

*def get\_user(self, user\_id):*

*user = cache.get("user.{0}", user\_id)*

*if user is None:*

*user = db.query("SELECT \* FROM users WHERE user\_id = {0}", user\_id)*

*if user is not None:*

*key = "user.{0}".format(user\_id)*

*cache.set(key, json.dumps(user))*

*return user*

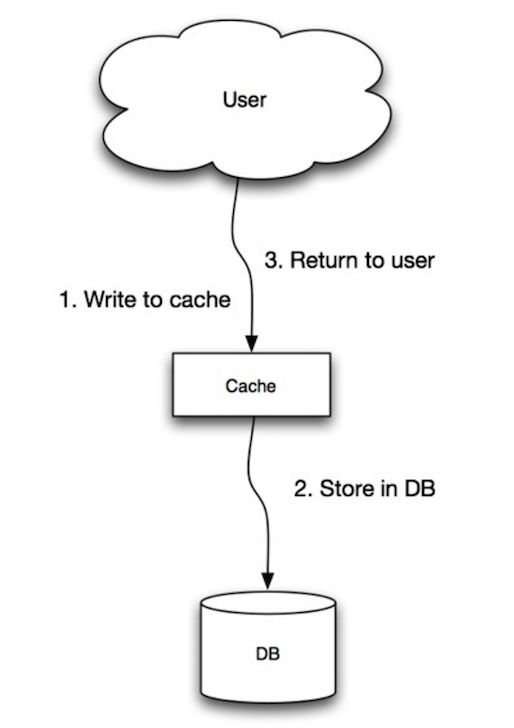
**Memcached is generally used in this manner.**

Subsequent reads of data added to cache are fast. Cache-aside is also referred to as lazy loading. Only requested data is cached, which avoids filling up the cache with data that isn't requested.

**Disadvantage**(s): cache-aside

* Each cache miss results in three trips, which can cause a noticeable delay.
* Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
* When a node fails, it is replaced by a new, empty node, increasing latency.

**Write-through:**



The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

* Application adds/updates entry in cache
* Cache synchronously writes entry to data store
* Return

**Application code:**

*set\_user(12345, {"foo":"bar"})*

**Cache code:**

*def set\_user(user\_id, values):*

*user = db.query("UPDATE Users WHERE id = {0}", user\_id, values)*

*cache.set(user\_id, user)*

Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast. Users are generally more tolerant of latency when updating data than reading data. Data in the cache is not stale.

**Disadvantage**(s): write through

* When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
* Most data written might never be read, which can be minimized with a TTL.

**Write-behind (write-back):**



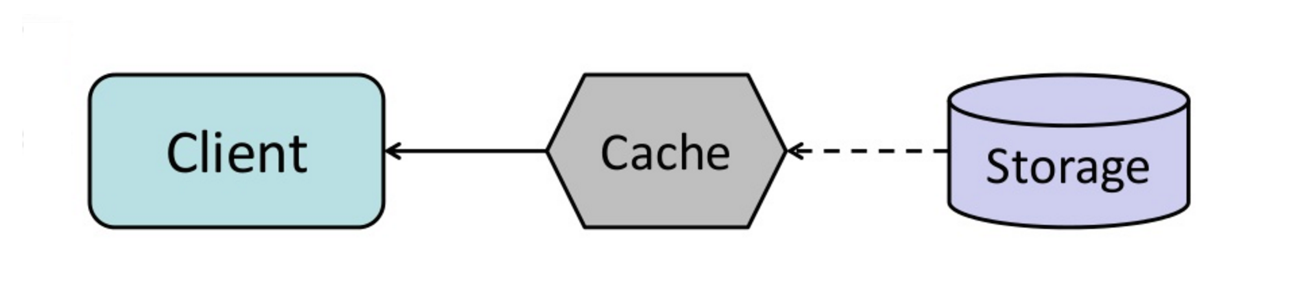
In write-behind, the application does the following:

* Add/update entry in cache
* Asynchronously write entry to the data store, improving write performance

**Disadvantage**(s): write-behind

* There could be data loss if the cache goes down prior to its contents hitting the data store.
* It is more complex to implement write-behind than it is to implement cache-aside or write-through.

**Refresh-ahead:**



You can configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.

Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future.

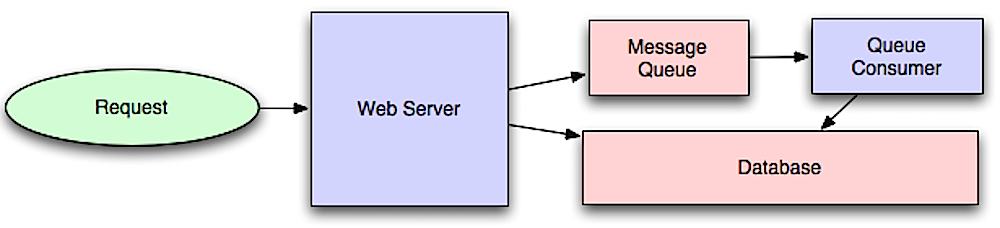
**Disadvantage**(s): refresh-ahead

* Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.

**Disadvantage(s): cache**

* Need to maintain consistency between caches and the source of truth such as the database through cache invalidation.
* Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
* Need to make application changes such as adding Redis or memcached.

# Asynchronism:



**Asynchronous** workflows help reduce request times for expensive operations that would otherwise be performed in-line. They can also help by doing time-consuming work in advance, such as periodic aggregation of data.

## Message queues

Message queues receive, hold, and deliver messages. If an operation is too slow to perform inline, you can use a message queue with the following workflow:

* An application publishes a job to the queue, then notifies the user of job status.
* A worker picks up the job from the queue, processes it, then signals the job is complete

The user is not blocked and the job is processed in the background. During this time, the client might optionally do a small amount of processing to make it seem like the task has completed. For example, if posting a tweet, the tweet could be instantly posted to your timeline, but it could take some time before your tweet is actually delivered to all of your followers.

**Redis** is useful as a simple message broker but messages can be lost.

**RabbitMQ** is popular but requires you to adapt to the 'AMQP' protocol and manage your own nodes.

**Amazon** **SQS** is hosted but can have high latency and has the possibility of messages being delivered twice.

## Task queues

Tasks queues receive tasks and their related data, runs them, then delivers their results. They can support scheduling and can be used to run computationally-intensive jobs in the background.

Celery has support for scheduling and primarily has python support.

## Back pressure

If queues start to grow significantly, the queue size can become larger than memory, resulting in cache misses, disk reads, and even slower performance. Back pressure can help by limiting the queue size, thereby maintaining a high throughput rate and good response times for jobs already in the queue. Once the queue fills up, clients get a server busy or HTTP 503 status code to try again later. Clients can retry the request at a later time, perhaps with exponential backoff.

**Explanation:**

To put it simply: Task or message, they can be thought of or used interchangeably. It's the asynchronous operation that matters. (Repeat that last line to yourself :)) The point of having a queue is that one guy can ask to do something or say something and forget about it, and another guy can follow up on it.

**A task/message queue**: Think of this like a passive data structure which acts like a queue, it 'stores' tasks and keeps them for processing later on. True, there is a process who manages this queue, but it doesn't do more, it just stores things and gives them when asked in an orderly way.

Righto, great. But who processes it? The **worker threads.**

Okay. So now, we have got 3 things.

**A task/message.**

A process who adds the tasks in to the **task queue (data structure),** and maintains this task queue.

**Worker processes** who take the tasks/messages from this queue and execute them.

So, you mean workers can read this queue and remove tasks from them? Yes they can, but this leads to problems (race conditions and such).

Enter the **'message broker'.**

The **'message broker'** is the one who takes the tasks from the task queue and distributes them to worker processes (even if they are on different machines) and manages the integrity of task completion, retrying tasks if workers fail, giving faster workers more tasks, and some other housekeeping for distributed processing.

Aha! So that's what the message broker does! Wait, why can't the 'Task/Message queue' manage the giving away of tasks to these workers, instead of just managing the queue?

They can and do so, this is why there is confusion sometimes. **Redis**, primarily, a high-performance passive data store, also does message broking/scheduling.

Some message brokers (like RabbitMQ) use their own task queues to store and schedule them.

The roles/functions may overlap, but the concept is essentially the same.

**Disadvantage**(s): **asynchronism**

* Use cases such as inexpensive calculations and real-time workflows might be better suited for synchronous operations, as introducing queues can add delays and complexity.