Some videos to get an overall idea:

https://github.com/yuchenwang2011/Hello\_World/blob/master/SystemDesign/Day1.txt

**Scalability for Dummies: (1) Clones**

Public servers of a scalable web service are hidden behind a load balancer. This load balancer evenly distributes load (requests from your users) onto your group/cluster of application servers. That means that if, for example, user Steve interacts with your service, he may be served at his first request by server 2, then with his second request by server 9 and then maybe again by server 2 on his third request.

Steve should always get the same results of his request back, independent what server he “landed on”. That leads to the first golden rule for scalability: every server contains exactly the same codebase and does not store any user-related data, like sessions or profile pictures, on local disc or memory.

Sessions need to be stored in a centralized data store which is accessible to all your application servers. It can be an external database or an external persistent cache, like Redis. An external persistent cache will have better performance than an external database. By external I mean that the data store does not reside on the application servers. Instead, it is somewhere in or near the data center of your application servers.

But what about deployment? How can you make sure that a code change is sent to all your servers without one server still serving old code? This tricky problem is fortunately already solved by the great tool Capistrano. It requires some learning, especially if you are not into Ruby on Rails, but it is definitely both the effort.

After “outsourcing” your sessions and serving the same codebase from all your servers, you can now create an image file from one of these servers (AWS calls this AMI - Amazon Machine Image.) Use this AMI as a “super-clone” that all your new instances are based upon. Whenever you start a new instance/clone, just do an initial deployment of your latest code and you are ready!

**Scalability for Dummies: (2) Database**

To break bottleneck of the database if you are using MySQL

**Path #1** is to stick with MySQL and keep the “beast” running. Hire a database administrator (DBA,) tell him to do master-slave replication (read from slaves, write to master) and upgrade your master server by adding RAM, RAM and more RAM. In some months, your DBA will come up with words like “sharding”, “denormalization” and “SQL tuning” and will look worried about the necessary overtime during the next weeks. At that point every new action to keep your database running will be more expensive and time consuming than the previous one. You might have been better off if you had chosen Path #2 while your dataset was still small and easy to migrate.

**Path #2** means to denormalize right from the beginning and include no more Joins in any database query. You can stay with MySQL, and use it like a NoSQL database, or you can switch to a better and easier to scale NoSQL database like MongoDB or CouchDB. Joins will now need to be done in your application code. The sooner you do this step the less code you will have to change in the future. But even if you successfully switch to the latest and greatest NoSQL database and let your app do the dataset-joins, soon your database requests will again be slower and slower. You will need to introduce a cache.

**Scalability for Dummies: (3) Cache**

Your users still have to suffer slow page requests when a lot of data is fetched from the database. The solution is the implementation of a cache.

With “cache” I always mean in-memory caches like Memcached or Redis. Please never do file-based caching, it makes cloning and auto-scaling of your servers just a pain.

But back to in-memory caches. A cache is a simple key-value store and it should reside as a buffering layer between your application and your data storage. Whenever your application has to read data it should at first try to retrieve the data from your cache. Only if it’s not in the cache should it then try to get the data from the main data source. Why should you do that? Because a cache is lightning-fast. It holds every dataset in RAM and requests are handled as fast as technically possible. For example, Redis can do several hundreds of thousands of read operations per second when being hosted on a standard server. Also writes, especially increments, are very, very fast. Try that with a database!

There are 2 patterns of caching your data. An old one and a new one:

**#1 - Cached Database Queries**

That’s still the most commonly used caching pattern. Whenever you do a query to your database, you store the result dataset in cache. A hashed version of your query is the cache key. The next time you run the query, you first check if it is already in the cache. The next time you run the query, you check at first the cache if there is already a result. This pattern has several issues. The main issue is the expiration. It is hard to delete a cached result when you cache a complex query (who has not?). When one piece of data changes (for example a table cell) you need to delete all cached queries who may include that table cell. You get the point?

**#2 - Cached Objects**

That’s my strong recommendation and I always prefer this pattern. In general, see your data as an object like you already do in your code (classes, instances, etc.). Let your class assemble a dataset from your database and then store the complete instance of the class or the assembed dataset in the cache. Sounds theoretical, I know, but just look how you normally code. You have, for example, a class called “Product” which has a property called “data”. It is an array containing prices, texts, pictures, and customer reviews of your product. The property “data” is filled by several methods in the class doing several database requests which are hard to cache, since many things relate to each other. Now, do the following: when your class has finished the “assembling” of the data array, directly store the data array, or better yet the complete instance of the class, in the cache! This allows you to easily get rid of the object whenever something did change and makes the overall operation of your code faster and more logical.

And the best part: it makes asynchronous processing possible! Just imagine an army of worker servers who assemble your objects for you! The application just consumes the latest cached object and nearly never touches the databases anymore!

**Scalability for Dummies: (4) Asynchronism**

please imagine that you want to buy bread at your favorite bakery. So you go into the bakery, ask for a loaf of bread, but there is no bread there! Instead, you are asked to come back in 2 hours when your ordered bread is ready. That’s annoying, isn’t it?

To avoid such a “please wait a while” - situation, asynchronism needs to be done. And what’s good for a bakery, is maybe also good for your web service or web app.

In general, there are two ways / paradigms asynchronism can be done.

**Async #1**

Let’s stay in the former bakery picture. The first way of async processing is the “bake the breads at night and sell them in the morning” way. No waiting time at the cash register and a happy customer. Referring to a web app this means doing the time-consuming work in advance and serving the finished work with a low request time.

Very often this paradigm is used to turn dynamic content into static content. Pages of a website, maybe built with a massive framework or CMS, are pre-rendered and locally stored as static HTML files on every change. Often these computing tasks are done on a regular basis, maybe by a script which is called every hour by a cronjob. This pre-computing of overall general data can extremely improve websites and web apps and makes them very scalable and performant. Just imagine the scalability of your website if the script would upload these pre-rendered HTML pages to AWS S3 or Cloudfront or another Content Delivery Network! Your website would be super responsive and could handle millions of visitors per hour!

**Async #2**

Back to the bakery. Unfortunately, sometimes customers has special requests like a birthday cake with “Happy Birthday, Steve!” on it. The bakery can not foresee these kind of customer wishes, so it must start the task when the customer is in the bakery and tell him to come back at the next day. Refering to a web service that means to handle tasks asynchronously.

Here is a typical workflow: A user comes to your website and starts a very computing intensive task which would take several minutes to finish. So the frontend of your website sends a job onto a job queue and immediately signals back to the user: your job is in work, please continue to the browse the page. The job queue is constantly checked by a bunch of workers for new jobs. If there is a new job then the worker does the job and after some minutes sends a signal that the job was done. The frontend, which constantly checks for new “job is done” - signals, sees that the job was done and informs the user about it. I know, that was a very simplified example.

If you now want to dive more into the details and actual technical design, I recommend you take a look at the first 3 tutorials on the RabbitMQ website. RabbitMQ is one of many systems which help to implement async processing. You could also use ActiveMQ or a simple Redis list. The basic idea is to have a queue of tasks or jobs that a worker can process. Asynchronism seems complicated, but it is definitely worth your time to learn about it and implement it yourself. Backends become nearly infinitely scalable and frontends become snappy which is good for the overall user experience.

If you do something time-consuming, try to do it always asynchronously.

**Performance vs scalability**

A service is scalable if it results in increased performance in a manner proportional to resources added. Generally, increasing performance means serving more units of work, but it can also be to handle larger units of work, such as when datasets grow.

Another way to look at performance vs scalability:

1If you have a performance problem, your system is slow for a single user.

2If you have a scalability problem, your system is fast for a single user but slow under heavy load.

**Latency vs throughput**

Latency is the time to perform some action or to produce some result.

Throughput is the number of such actions or results per unit of time.

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

**CAP theorem**

**Consistency**: all nodes see the same data at the same time

**Availability**: a guarantee that every request receives a response about whether it succeeded or failed

**Partition tolerance**: system continues to operate despite arbitrary message loss or failure of part of the system

In a distributed context, the choice is between CP and AP. Unfortunately, CA is just a joke, because single point of failure is a red flag in the real distributed systems world.

**Two main patterns to support high availability: Fail-over and Replication**

(personal understanding: failover is to prepare failure, replication is master and slave is already replicated and in use)

**Fail-over**

**Active-passive (Master-Slave)**

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-active(master-master failover)**

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.

**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.

**DNS:**

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

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 (TTL).

**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.

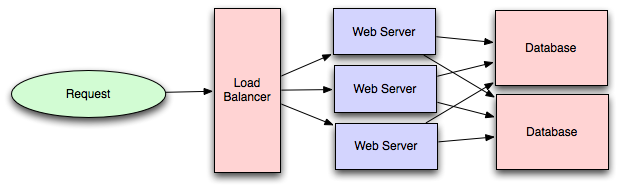
**Advantages:**

1users receive content at data centers close to them

2your servers do not have to serve requests that the CDN fulfills

**Load Balancer:**

Load balancers distribute incoming client requests to computing resources such as application servers and databases. In each case by Random/Least loaded/Session/cookies/Round robin or weighted round robin/Layer 4/Layer 7



1Preventing requests from going to unhealthy servers

2Preventing overloading resources

3Helping eliminate single points of failure

4SSL termination - Decrypt incoming requests and encrypt server responses so backend servers don’t need to

5Session persistence - Issue cookies and route a specific client's requests to same instance if the web apps do not keep track of sessions

**Layer 4 load balancers** look at info at the transport layer

**Layer 7 load balancers** look at the application layer

layer 4 load balancing requires less time and computing resources than Layer 7

Sessions can be stored in a centralized data store such as a database (SQL, NoSQL) or a persistent cache (Redis, Memcached)

**Reverse proxy**

A reverse proxy is a web server that centralizes internal services and provides unified interfaces to the public.

1Increased security - Hide information about backend servers, blacklist IPs, limit number of connections per client

2Increased scalability and flexibility - Clients only see the reverse proxy's IP, allowing you to scale servers behind

3SSL termination - Decrypt incoming requests and encrypt server responses so backend servers don’t need to

4Caching - Return the response for cached requests

5Static content - Serve static content directly

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

**Introduction to Architecting System for scale (Another archicle covers previous topic)**

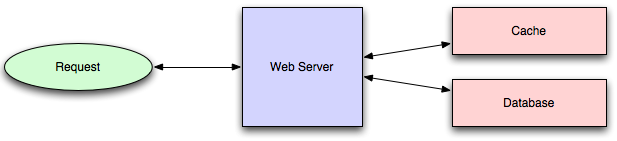
**Load balancing**

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.

**Caching**

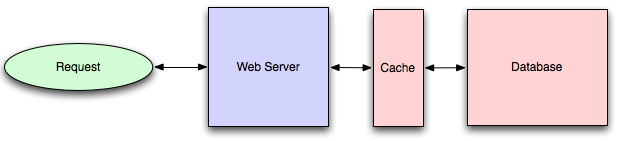
Caching consists of: pre-calculating results (e.g. the number of visits from each referring domain for the previous day), pre-generating expensive indexes (e.g. suggested stories based on a user's click history), and storing copies of frequently accessed data in a faster backend (e.g. Memcache).

**Application caching**



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

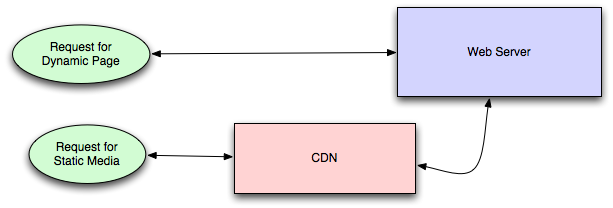
**Database Caching**



**In-memory caches**

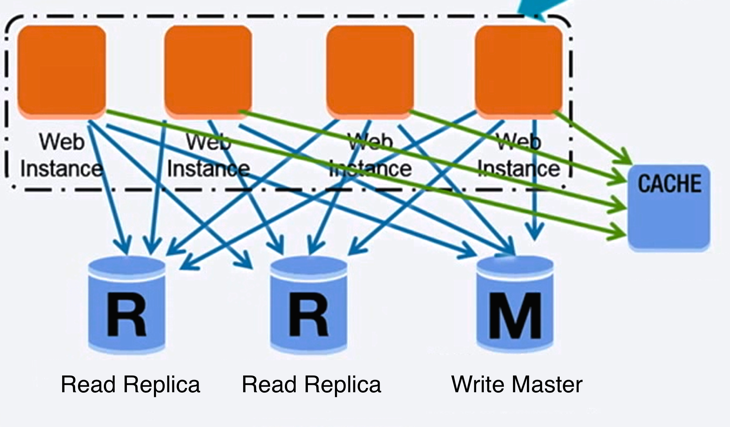
Stores entire set of data in memory. Memcached and Redis are both examples of in-memory caches This is because accesses to RAM are orders of magnitude faster than those to disk. It only keeps least recently used (LRU) data

**Content distribution networks**



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).

**Database**



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

**ACID**

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

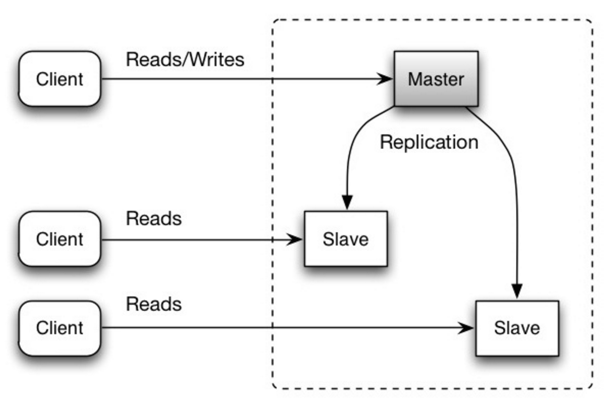
1 master-slave replication 2 master-master replication

3 federation 4 sharding

5 denormalization 6 SQL tuning

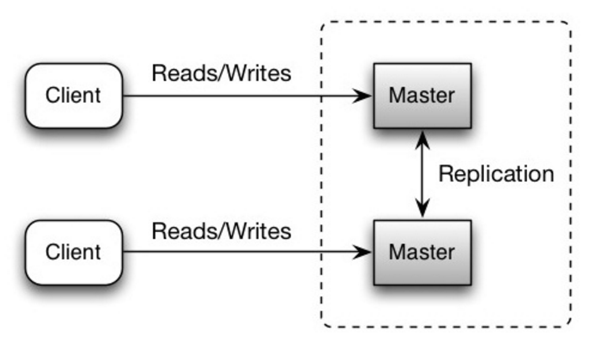
**Master-slave replication**

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.



**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. You need a load balancer to determine which server to write. Latency increases and could be more conflict.



**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.

**Disadvantage**(s): federation

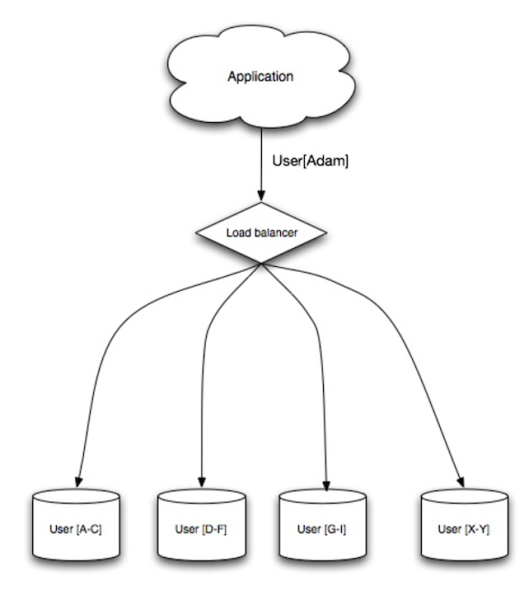
1Federation is not effective if your schema requires huge functions or tables.

2You'll need to update your application logic to determine which database to read and write.

3Joining data from two databases is more complex with a server link.

4Federation 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.

**Disadvantage**(s): sharding

1You'll need to update your application logic to work with shards, which could result in complex SQL queries.

2.Data distribution can become lopsided in a shard.

3.Rebalancing adds additional complexity. A sharding function based on **consistent hashing** can reduce the amount of transferred data.

4.Joining data from multiple shards is more complex.

5.Sharding adds more hardware and additional complexity.

**Denomalization**

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. 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 (not only 3 advices)**

1MySQL dumps to disk in contiguous blocks for fast access.

2Use CHAR instead of VARCHAR for fixed-length fields.

3Use TEXT for large blocks of text such as blog posts

**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 , availability over consistency**

1Basically available - the system guarantees availability.

2Soft state - the state of the system may change over time, even without input.

3Eventual consistency - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

**SQL vs NoSQL**

**SQL**:

1Structured data

4Need for complex joins

5Clear patterns for scaling

7Lookups by index are very fast

**NoSQL**:

1Dynamic or flexible schema

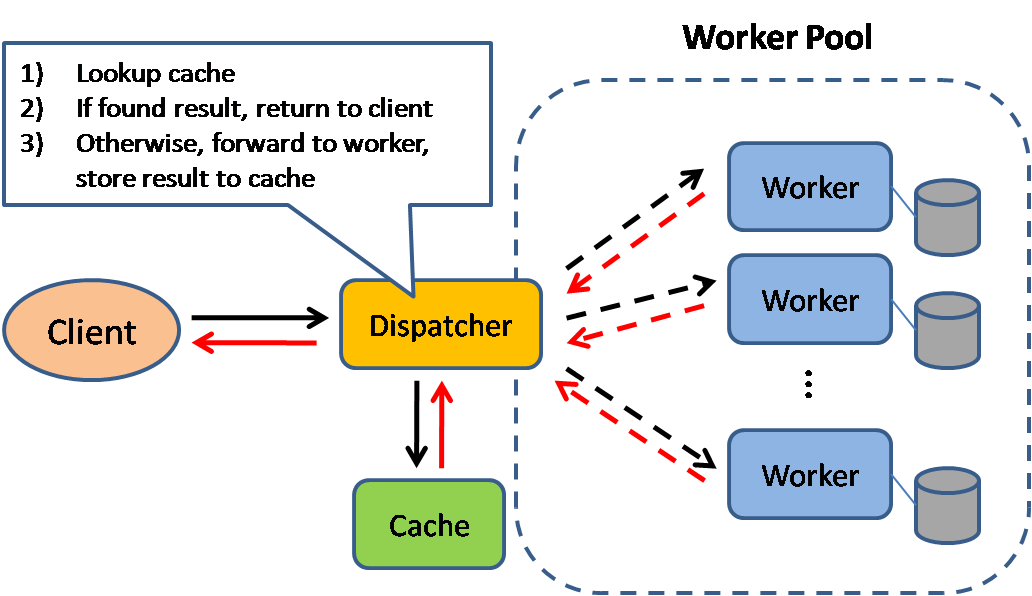
2No need for complex joins

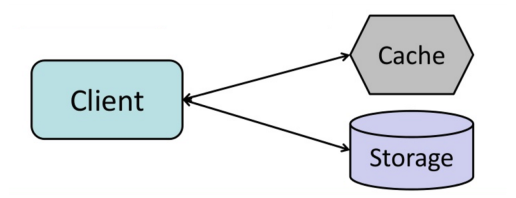
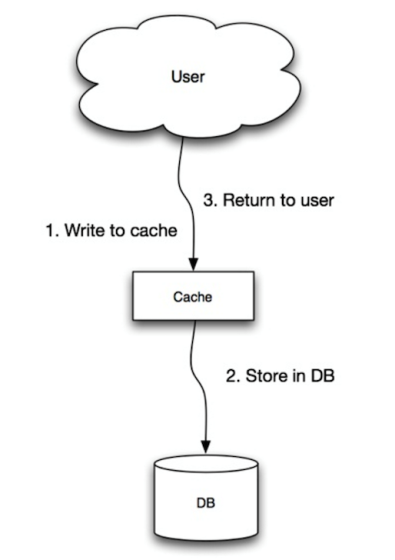
3Store many TB (or PB) of data

4Very data intensive workload

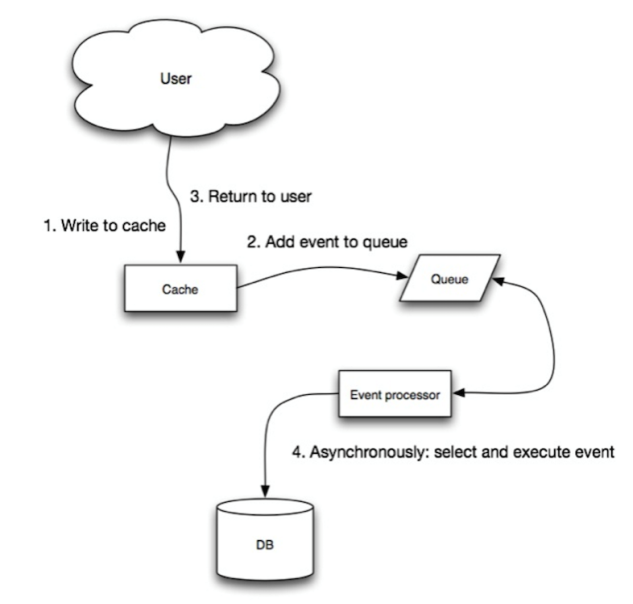
**Cache**

Client caching, CDN caching, Web server caching, Database caching

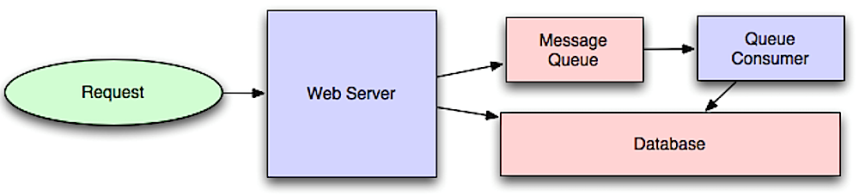


**Cache-aside:** such as Memcached **Write-through**: (might data loss)  
 

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



**Asynchronism**



Asynchronous workflows help reduce request times for expensive operations that would otherwise be performed in-line.

An application publishes a job to the queue, then notifies the user of job status

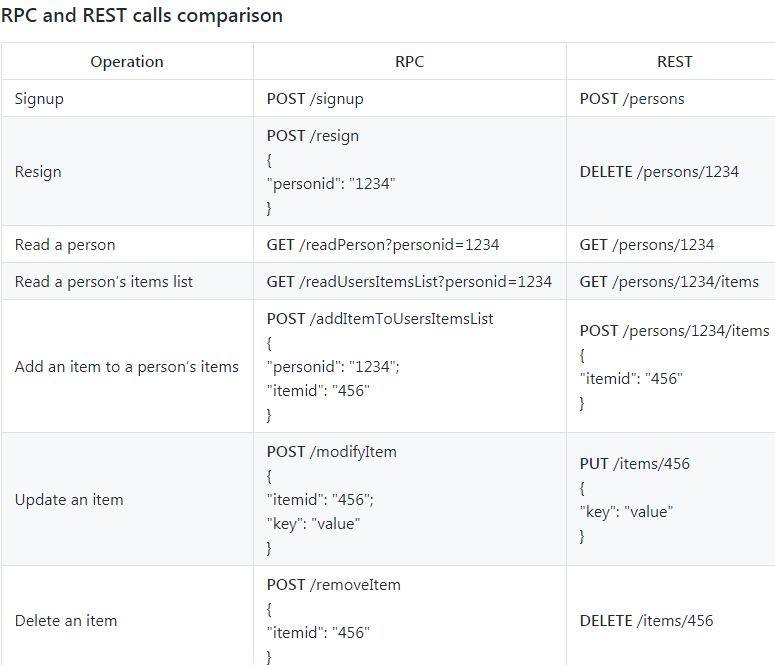
The user is not blocked and the job is processed in the background. For example, if posting a tweet, the tweet could be instantly posted to your timeline, but it could take some time before your tweet is actually delivered to all of your followers.

**Back pressure**

If queues start to grow significantly, the queue size can become larger than memory. Back pressure can help by limiting the queue size. Clients get a server busy or HTTP 503 status code to try again later.

**TCP vs UDP**

**RPC vs REST**



**Back of the Envelop Calculation**

-Read sequentially from disk at 30 MB/s

-Read sequentially from 1 Gbps Ethernet at 100 MB/s

-Read sequentially from SSD at 1 GB/s

-Read sequentially from main memory at 4 GB/s

-6-7 world-wide round trips per second

-2,000 round trips per second within a data center

After learnt all knowledge above, you should be able to understand and draw this diagram:

