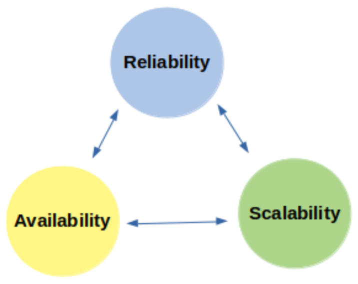
System Design is the process of designing the architecture, components, and interfaces for a system so that it meets the end-user requirements.



Reliability in System Design –

A system is Reliable when it can meet the end-user requirement. When you are designing a system you should have planned to implement a set of features and services in your system. If your system can serve all those features without wearing out then your System can be considered to be **Reliable**.

A **Fault Tolerant** system can be one that can continue to be functioning reliably even in the presence of faults. **Faults** are the errors that arise in a particular component of the system. An occurrence of fault doesn't guarantee Failure of the System.

Failure is the state when the system is not able to perform as expected. It is no longer able to provide certain services to the end-users.

Availability in System Design –

Availability is a characteristic of a System which aims to ensure an agreed level of Operational Performance, also known as **uptime**. It is essential for a system to ensure high availability in order to serve the user's requests.

The extent of Availability varies from system to system. Suppose you are designing a Social Media Application then high availability is not much of a need. A delay of a few seconds can be tolerated. Getting to view the post of your favorite celebrity on Instagram with a delay of 5 to 10 seconds will not be much of an issue. But if you are designing a system for hospitals, Data Centers, or Banking, then you should ensure that your system is highly available. Because a delay in the service can lead to a huge loss.

There are various principles you should follow in order to ensure the availability of your system:

- Your System should not have a **Single Point of Failure**. Basically, your system should not be dependent on a single service in order to process all of its requests. Because when that service fails then your entire system can be jeopardized and end up becoming unavailable.
- Detecting the Failure and resolving it at that point.

Scalability in System Design –

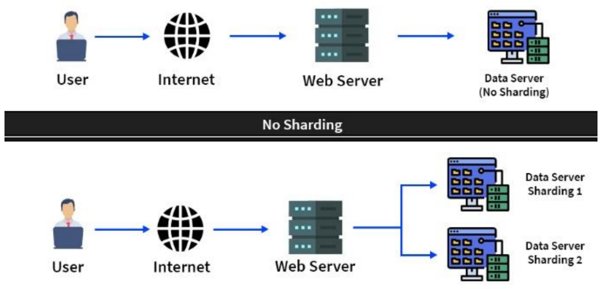
Scalability refers to the ability of the System to cope up with the increasing load. While designing the system you should keep in mind the load experienced by it. It's said that if you have to design a system for load **X** then you should plan to design it for **10X** and Test it for **100X**. There can be a situation where your system can experience an increasing load. Suppose you are designing an E-commerce application then you can expect a spike in the load during a Flash Sale or when a new Product is Launched for sale. In that case, your system should be smart enough to handle the increasing load efficiently and that makes it **Scalable**.

In order to ensure scalability you should be able to compute the load that your system will experience. There are various factors that describe the Load on the System:

- Number of requests coming to your system for getting processed per day
- Number of Database calls made from your system
- Amount of Cache Hit or Miss requests to your system
- Users currently active on your system

What is Sharding or Data Partitioning?

It is basically a database architecture pattern in which we split a large dataset into smaller chunks (logical shards) and we store/distribute these chunks in different machines/database nodes (physical shards). Each chunk/partition is known as a "**shard**" and each shard has the same database schema as the original database. We distribute the data in such a way that each row appears in exactly one shard. It's a good mechanism to improve the **scalability** of an application.



Database shards are autonomous; they don't share any of the same data or computing resources. In some cases, though, it may make sense to replicate certain tables into each shard to serve as reference tables.

Advantages of Sharding

- **Solve Scalability Issue:** With a single database server architecture any application experience performance degradation when users start growing on that application. Reads and write queries become slower and the network bandwidth starts to saturate. At some point, you will be running out of disk space. Database sharding fixes all these issues by partitioning the data across multiple machines.
- **High Availability:** A problem with single server architecture is that if an outage happens then the entire application will be unavailable which is not good for a website with more number of users. This is not the case with a sharded database. If an outage happens in sharded architecture, then only some specific shards will be down. All the other shards will continue the operation and the entire application won't be unavailable for the users.
- **Speed Up Query Response Time:** When you submit a query in an application with a large monolithic database and have no sharded architecture, it takes more time to find the result. It has to search every row in the table and that slows down the response time for the query you have given. This doesn't happen in sharded architecture. In a sharded database a query has to go through fewer rows and you receive the response in less time.
- **More Write Bandwidth:** For many applications writing is a major bottleneck. With no master database serializing writes sharded architecture allows you to write in parallel and increase your write throughput.
- **Scaling Out:** Sharding a database facilitates ***horizontal scaling***, known as *scaling out*. In horizontal scaling, you **add more machines** in the network and distribute the load on these machines for faster processing and response. This has many advantages. You can do more work simultaneously and you can handle high requests from the users, especially when writing data because there are parallel paths through your system. You can also load balance web servers that access shards over different network paths, which are processed by different CPUs, and use separate caches of RAM or disk IO paths to process work.

Disadvantages of Sharding

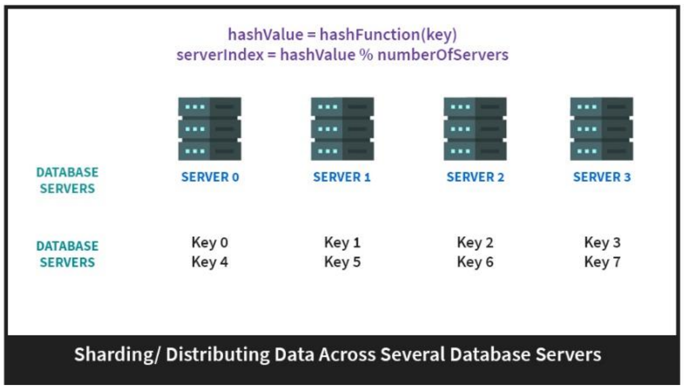
- **Adds Complexity in the System:** You need to be careful while implementing a proper sharded database architecture in an application. It's a complicated task and if it's not implemented properly then you may lose the data or get corrupted tables in your database. You also need to manage the data from multiple shard locations instead of managing and accessing it from a single entry point. This may affect the workflow of your team which can be potentially disruptive to some teams.
- **Rebalancing Data:** In a sharded database architecture, sometimes shards become unbalanced (when a shard outgrows other shards). Consider an example that you have two shards of a database. One shard store the name of the customers begins with letter A through M. Another shard store the name of the customer begins with the letters N through Z. If there are so many users with the letter L then shard one will have more data than shard two. This will affect the performance (slow down) of the application and it will stall out for a significant portion of your users. The A-M shard will become unbalance and it will be known as ***database hotspot***. To overcome this problem and to rebalance the data you need to do re-sharding for even data distribution. Moving data from one shard to another shard is not a good idea because it requires a lot of downtimes.
- **Joining Data From Multiple Shards is Expensive:** In a single database, joins can be performed easily to implement any functionalities. But in sharded architecture, you need to pull the data from different shards and you need to perform joins across multiple networked servers You can't submit a single query to get the data from various shards. You need to submit **multiple queries** for each one of the shards, pull out the data, and join the data across the network. This is going to be a very expensive and time-consuming process. It adds **latency** to your system.
- **No Native Support:** Sharding is not natively supported by every database engine. For example, PostgreSQL doesn't include automatic sharding features, so there you have to do manual sharding. You need to follow the "roll-your-own" approach. It will be difficult for you to find the tips or documentation for sharding and troubleshoot the problem during the implementation of sharding.

Sharding Architectures

1. Key Based Sharding

This technique is also known as **hash-based** sharding. Here, we take the value of an entity such as customer ID, customer email, IP address of a client, zip code, etc and we use this value as an input of the **hash function**. This process generates a **hash value** which is used to determine which shard we need to use to store the data. We need to keep in mind that the values entered into the hash function should all come from the **same column** (shard key) just to ensure that data is placed in the correct order and in a consistent manner. Basically, shard keys act like a *primary key* or a unique identifier for individual rows.

Consider an example that you have 3 database servers and each request has an application id which is incremented by 1 every time a new application is registered. To determine which server data should be placed on, we perform a modulo operation on these applications id with the number 3. Then the remainder is used to identify the server to store our data.

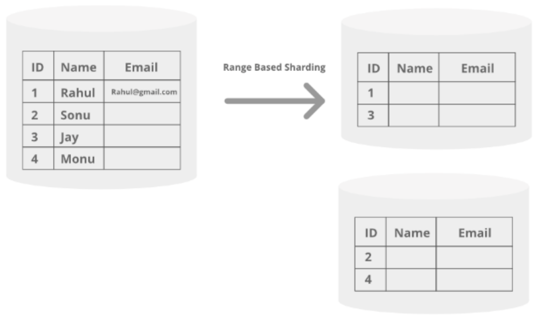

hashValue = hashFunction(key)
serverIndex = hashValue % numberOfServers
DATABASE SERVERS
SERVER 0
SERVER 1
SERVER 2
SERVER 3
DATABASE SERVERS
Key 0
Key 4
Key 1
Key 5
Key 2
Key 6
Key 3
Key 7
Sharding/ Distributing Data Across Several Database Servers

The downside of this method is **elastic load balancing** which means if you will try to add or remove the database servers dynamically it will be a difficult and expensive process. For example, in the above one if you will add 5 more servers then you need to add more corresponding hash values for the additional entries. Also, the majority of the existing keys need to be remapped to their new, correct hash value and then migrated to a new server. The hash function needs to be changed from modulo 3 to modulo 8. While the migration of data is in effect both the new and old hash functions won't be valid. During the migration, your application won't be able to service a large number of requests and you'll experience downtime for your application till the migration completes.

Note: *A shard shouldn't contain values that might change over time. It should be always static otherwise it will slow down the performance.*

2. Horizontal or Range Based Sharding

In this method, we split the data based on the **ranges** of a given value inherent in each entity. Let's say you have a database of your online customers' names and email information. You can split this information into two shards. In one shard you can keep the info of customers whose first name starts with A-P and in another shard, keep the information of the rest of the customers.

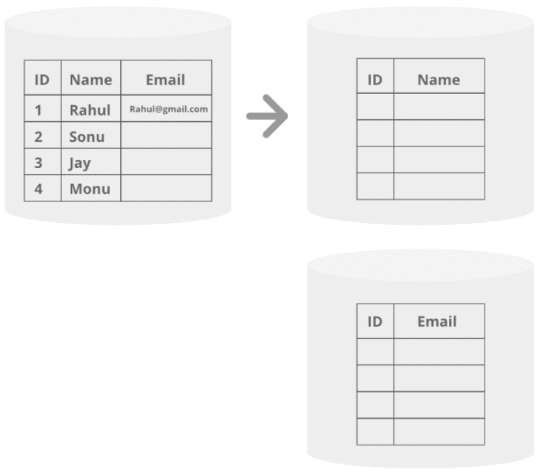


Range-based sharding is the simplest sharding method to implement. Every shard holds a different set of data but they all have the same schema as the original database. In this method, you just need to identify in which range your data falls, and then you can store the entry to the corresponding shard. This method is best suitable for storing non-static data (example: storing the contact info for students in a college.)

The drawback of this method is that the data may not be **evenly distributed** on shards. In the above example, you might have a lot of customers whose names fall into the category of A-P. In such cases, the first shard will have to take more load than the second one and it can become a system bottleneck.

3. Vertical Sharding

In this method, we split the entire column from the table and we put those columns into new distinct tables. Data is totally independent of one partition to the other ones. Also, each partition holds both distinct rows and columns. Take the example of Twitter features. We can split different features of an entity in different shards on different machines. On Twitter users might have a profile, number of followers, and some tweets posted by his/her own. We can place the user profiles on one shard, followers in the second shard, and tweets on a third shard.

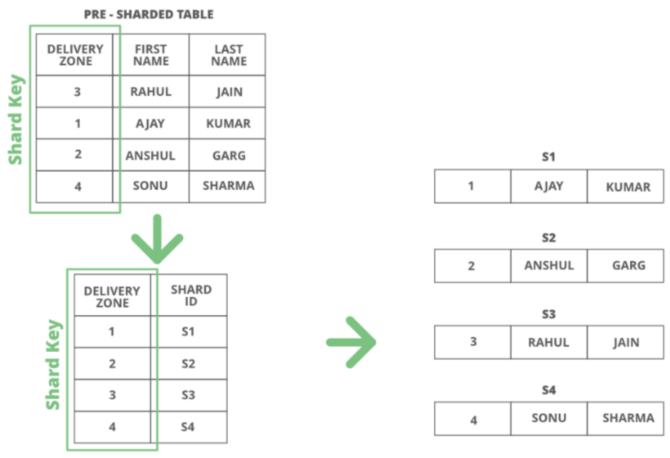


In this method, you can separate and handle the critical part (for example user profiles) non-critical part of your data (for example, blog posts) individually and build different replication and consistency models around it. This is one of the main advantages of this method.

The main drawback of this scheme is that to answer some queries you may have to combine the data from different shards which unnecessarily increases the development and operational complexity of the system. Also, if your application will grow later and you add some more features in it then you will have to further shard a feature-specific database across multiple servers.

4. Directory-Based Sharding

In this method, we create and maintain a **lookup service** or lookup table for the original database. Basically we use a **shard key** for lookup table and we do **mapping** for each entity that exists in the database. This way we keep track of which database shards hold which data.



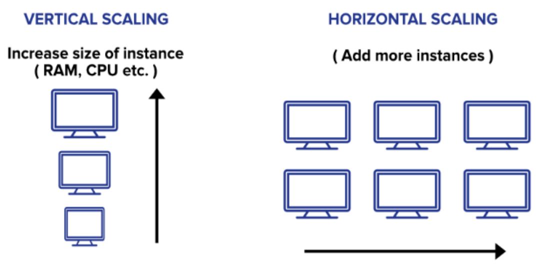
The lookup table holds a static set of information about where specific data can be found. In the above image, you can see that we have used the delivery zone as a shard key. Firstly the client application queries the lookup service to find out the shard (database partition) on which the data is placed. When the lookup service returns the shard it queries/updates that shard.

Directory-based sharding is much more flexible than range based and key-based sharding. In range-based sharding, you're bound to specify the ranges of values. In key-based, you are bound to use a fixed hash function which is difficult to change later. In this approach, you're free to use any algorithm you want to assign to data entries to shards. Also, it's easy to add shards dynamically in this approach.

The major drawback of this approach is the single point of failure of the lookup table. If it will be corrupted or failed then it will impact writing new data or accessing existing data from the table.

Option for scaling your database can be grouped into two major categories…

- **Vertical Scaling**
- **Horizontal** Scaling



1. Vertical Scaling

In simple terms upgrading the capacity of a single machine or moving to a new machine with more power is called vertical scaling. You can add more powers to your machine by adding better processors, increasing RAM, or other power increasing adjustments. Vertical scaling can be easily achieved by switching from small to bigger machines but remember that this involves downtime. You can enhance the capability of your server without manipulating your code.

- This approach is also referred to as the '**scale-up**' approach.
- It doesn't require any partitioning of data and all the traffic resides on a **single node with more capacity**.
- Easy implementation.
- Less administrative efforts as you need to manage just one system.
- Application compatibility is maintained.
- Mostly used in small and mid-sized companies.
- MySQL and Amazon RDS is a good example of vertical scaling.

Drawbacks

- Limited Scaling.
- Limited potential for improving network I/O or disk I/O.
- Replacing the server will require downtime in this approach.
- Greater risk of outages and hardware failures.
- Finite scope of upgradeability in the future.
- Implementation cost is expensive.

2. Horizontal Scaling

This approach is the best solution for projects which have requirements for high availability or failover. In horizontal scaling, we enhance the performance of the server by adding more machines to the network, sharing the processing and memory workload across multiple devices. We simply add more instances of the server to the existing pool of servers and distribute the load among these servers. In this approach, there is no need to change the capacity of the server or replace the server. Also, like vertical scaling, there is no downtime while adding more servers to the network. Most organizations choose this approach because it includes increasing I/O concurrency, reducing the load on existing nodes, and increasing disk capacity.

- This approach is also referred to as the '**scale-out**' approach.
- Horizontal scalability can be achieved with the help of a distributed file system, clustering, and load–balancing.
- Traffic can be managed effectively.
- Easier to run fault-tolerance.
- Easy to upgrade
- Instant and continuous availability.
- Easy to size and resize properly to your needs.
- Implementation cost is less expensive compared to scaling-up
- Google with its Gmail and YouTube, Yahoo, Facebook, eBay, Amazon, etc. are heavily utilizing horizontal scaling.
- Cassandra and MongoDB is a good example of horizontal scaling.

Drawbacks

- Complicated architectural design
- High licensing fees
- High utility costs such (cooling and electricity)
- The requirement of extra networking equipment such as routers and switches.

A Short Comparison

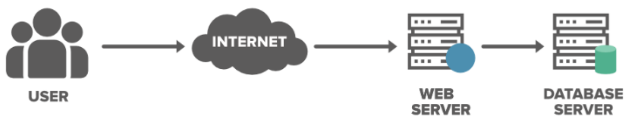
We have understood the meaning of both the major categories of scaling an application. We also have discussed some pros and cons of each one of them. Let's do a quick comparison of these two approaches based on these pros and cons…

Horizontal Scaling	Vertical Scaling
Load balancing required	Load balancing not required
Resilient to system failure	Single point of failure
Utilizes Network Calls	Interprocess communication
Data inconsistency	Data consistent
Scales well	Hardware limit

- **Load Balancing:** Horizontal scaling requires load balancing to distribute or spread the traffic among several machines. In the vertical machine, there is just one machine to handle the load so it doesn't require a load balancer.
- **Failure Resilience:** Horizontal scaling is more resistant to system failure. If one of the machines fails you can redirect the request to another machine and the application won't face downtime. This is not in the case of vertical scaling, it has a single machine so it will have a single point of failure. This simply means in horizontal scaling you can achieve availability but in vertical scaling, DB is still running on a single box so it doesn't improve availability.
- **Machine Communication:** Horizontal scaling requires network communication, or calls, between machines. Network calls are slow and more to prone failure. This is not in the case of vertical scaling, Vertical scaling works on inter-process communication that is quite fast.
- **Data Consistency:** Data is inconsistent in horizontal scaling because different machines handle different requests which may lead to their data becoming out of sync which must be addressed. On the other side, vertical machines have just one single machine where all the requests will be redirected, so there is no issue of inconsistency of data in vertical scaling.
- **Limitations:** Depends on budget, space, or requirement you can add as many servers as you want in horizontal scaling and scales your application as much as you want. This is not in the case of vertical scaling. There is a finite limit to the capacity achievable with vertical scaling. Scaling beyond that capacity results in downtime and comes with an upper limit. So a single machine can only be improved upon until it reaches the current limits of computing.

What is a Load Balancer?

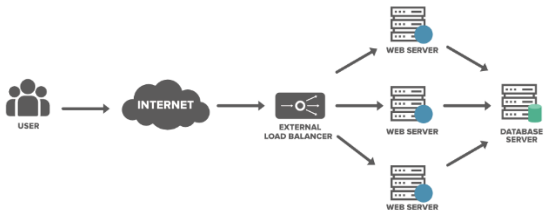
A load balancer works as a "traffic cop" sitting in front of your server and routing client requests across all servers. It simply distributes the set of requested operations (database write requests, cache queries) effectively across multiple servers and ensures that no single server bears too many requests that lead to degrading the overall performance of the application. A load balancer can be a physical device or a virtualized instance running on specialized hardware or a software process. Consider a scenario where an application is running on a single server and the client connects to that server directly without load balancing.



- **Single Point of Failure:** If the server goes down or something happens to the server the whole application will be interrupted and it will become unavailable for the users for a certain period. It will create a bad experience for users which is unacceptable for service providers.

- **Overloaded Servers:** There will be a limitation on the number of requests that a web server can handle. If the business grows and the number of requests increases the server will be overloaded. To solve the increasing number of requests we need to add a few more servers and we need to distribute the requests to the cluster of servers.

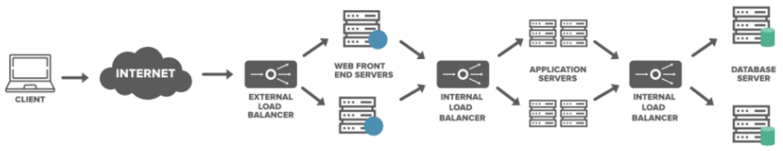
To solve the above issue and to distribute the number of requests we can add a load balancer in front of the web servers and allow our services to handle any number of requests by adding any number of web servers in the network. We can spread the request across multiple servers. For some reason, if one of the servers goes offline the service will be continued. Also, the latency on each request will go down because each server is not bottle-necked on RAM/Disk/CPU anymore.



- Load balancers minimize server response time and maximize throughput.
- Load balancer ensures high availability and reliability by sending requests only to online servers
- Load balancers do continuous health checks to monitor the server's capability of handling the request.
- Depending on the number of requests or demand load balancers add or remove the number of servers.

Where Are Load Balancers Typically Placed?

Below is the image where a load balancer can be placed…



- In between the client application/user and the server
- In between the server and the application/job servers
- In between the application servers and the cache servers
- In between the cache servers the database servers

Types of Load Balancers

1. Software Load Balancers in Clients

As the name suggests all the logic of load balancing resides on the client application (Eg. A mobile phone app). The client application will be provided with a list of web servers/application servers to interact with. The application chooses the first one in the list and requests data from the server. If any failure occurs persistently (after a configurable number of retries) and the server becomes unavailable, it discards that server and chooses the other one from the list to continue the process. This is one of the cheapest ways to implement load balancing.

2. Software Load Balancers in Services

These load balancers are the pieces of software that receive a set of requests and redirect these requests according to a set of rules. This load balancer provides much more flexibility because it can be installed on any standard device (Ex: Windows or Linux machine). It is also less expensive because there is no need to purchase or maintain the physical device, unlike hardware load balancers. You can have the option to use the off-the-shelf software load balancer or you can write your custom software (Ex: load balance Active Directory Queries of Microsoft Office365) for load balancing.

3. Hardware Load Balancers

As the name suggests we use a physical appliance to distribute the traffic across the cluster of network servers. These load balancers are also known as Layer 4-7 Routers and these are capable of handling all kinds of HTTP, HTTPS, TCP, and UDP traffic. HLDs provide a virtual server address to the outside world. When a request comes from a client application, it forwards the connection to the most appropriate real server doing bi-directional network address translation (NAT). HLDs can handle a large volume of traffic but it comes with a hefty price tag and it also has limited flexibility.

HLDs keep doing the health checks on each server and ensure that each server is responding properly. If any of the servers don't produce the desired response, it immediately stops sending the traffic to the servers. These load balancers are expensive to acquire and configure, that is the reason a lot of service providers use them only as the first entry point of user requests. Later the internal software load balancers are used to redirect the data behind the infrastructure wall.

Different Categories of Load Balancing

Generally, load balancers are grouped into three categories…

1. Layer 4 (L4) Load Balancer

In the OSI model layer 4 is the transport layer(TCP/SSL) where the routing decisions are made. Layer 4 load balancer is also referred to as **Network Load Balancing** and as the name suggests it leverages network layer information to make the routing decision for the traffic. It can control millions of requests per second and it handles all forms of TCP/UDP traffic. The decision will be based on the TCP or UDP ports that packets use along with their source and destination IP addresses. The L4 load balancer also performs Network Address Translation (NAT) on the request packet but it doesn't inspect the actual contents of each packet. This category of load balancer maximizes the utilization and availability by distributing the traffic across IP addresses, switches, and routers.

2. Layer 7 (L7) Load Balancer

Layer 7 load balancer is also referred to as **Application Load Balancer** or **HTTP(S) Load Balancer**. It is one of the oldest forms of load balancing. In the OSI model, Layer 7 is the application layer (HTTP/HTTPS) where the routing decisions execute. Layer 7 adds content switching to load balancing and it uses information such as HTTP header, cookies, uniform resource identifier, SSL session ID, and HTML form data to decide the routing request across the servers.

3. Global Server Load Balancing (GSLB)

Today a lot of applications are hosted in cloud data centers in multiple geographic locations. This is the reason a lot of organizations are moving to a different load balancer that can deliver applications with greater reliability and lower latency to any device or location. With the significant change in the capability of the load balancers, GSLB fulfills these expectations of IT organizations. GSLB extends the capability of L4 and L7 servers in different geographic locations and distributes a large amount of traffic across multiple data centers efficiently. It also ensures a consistent experience for end-users when they are navigating multiple applications and services in a digital workspace.

Load Balancing Algorithms

We need a load balancing algorithm to decide which request should be redirected to which backend server. The different system uses different ways to select the servers from the load balancer. Companies use varieties of load balancing algorithm techniques depending on the configuration. Some of the common load balancing algorithms are given below:

1. Round Robin

Requests are distributed across the servers in a sequential or rotational manner. For example, the first request goes to the first server, the second one goes to the second server, the third request goes to the third server and it continues further for all the requests. It is easy to implement but it doesn't consider the load already on a server so there is a risk that one of the servers receives a lot of requests and becomes overloaded.

2. Weighted Round Robin

It is much similar to the round-robin technique. The only difference is, that each of the resources in a list is provided a weighted score. Depending on the weighted score the request is distributed to these servers. So in this method, some of the servers get a bigger share of the overall request.

3. Least Connection Method

In this method, the request will be directed to the server with the fewest number of requests or active connections. To do this load balancer needs to do some additional computing to identify the server with the least number of connections. This may be a little bit costlier compared to the round-robin method but the evaluation is based on the current load on the server. This algorithm is most useful when there is a huge number of persistent connections in the traffic unevenly distributed between the servers.

4. Least Response Time Method

This technique is more sophisticated than the Least connection method. In this method, the request is forwarded to the server with the fewest active connections and the least average response time. The response time taken by the server represents the load on the server and the overall expected user experience.

5. Source IP Hash

In this method, the request is sent to the server based on the client's IP address. The IP address of the client and the receiving compute instance are computed with a cryptographic algorithm.

Caching – An Introduction

Let's say you prepare dinner every day and you need some ingredients for food preparation. Whenever you prepare the food, will you go to your nearest shop to buy these ingredients? Absolutely no. That's a time-consuming process and every time instead of visiting the nearest shop, you would like to buy the ingredients once and you will store that in your refrigerator. That will save a lot of time. This is caching and your **refrigerator works like a cache/local store/temporary store**. The cooking time gets reduced if the food items are already available in your refrigerator.

The same things happen in the system. In a system accessing data from primary memory (RAM) is faster than accessing data from secondary memory (disk). **Caching acts as the local store** for the data and retrieving the data from this local or temporary storage is easier and faster than retrieving it from the database. Consider it as ***a short-term memory that has limited space but is faster and contains the most recently accessed items***. So If you need to rely on a certain piece of data often then cache the data and retrieve it faster from the memory rather than the disk.

Note: You know the benefits of the cache but that doesn't mean you store all the information in your cache memory for faster access. You can't do that for multiple reasons. One of the reasons is the hardware of the cache which is much more expensive than a normal database. Also, the search time will increase if you store tons of data in your cache. So in short a cache needs to have the most relevant information according to the request which is going to come in the future.

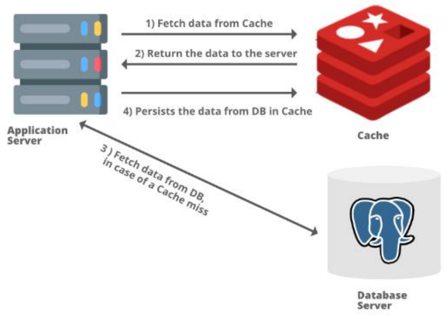
Where Cache Can be Added?

Caching is used in almost every layer of computing. In hardware, for example, you have various layers of cache memory. You have layer 1 cache memory which is the CPU cache memory, then you have layer 2 cache memory and finally, you would have the regular RAM (random access memory). You also have to cache in the operating systems such as caching various kernel extensions or application files. You also have caching in a web browser to decrease the load time of the website. So caching can be used in almost every layer: hardware, OS, Web browsers, and web applications, but are often found nearest to the front-end.

How Does Cache Work?

Typically, web application stores data in a database. When a client requests some data, it is fetched from the database and then it is returned to the user. Reading data from the database needs **network calls and I/O operation** which is a time-consuming process. Cache reduces the network call to the database and speeds up the performance of the system. Take the example of Twitter: when a tweet becomes viral, a huge number of clients request the same tweet. Twitter is a gigantic website that has millions of users. It is inefficient to read data from the disks for this large volume of user requests. To **reduce the number of calls** to the database, we can use cache and the tweets can be provided much faster.

In a typical web application, we can add an application server cache, an **in-memory store** like Redis alongside our application server. When the first time a request is made a call will have to be made to the database to process the query. This is known as a **cache miss**. Before giving back the result to the user, the result will be saved in the cache. When the second time a user makes the same request, the application will check your cache first to see if the result for that request is cached or not. If it is then the result will be returned from the in-memory store. This is known as a **cache hit**. The response time for the second time request will be a lot less than the first time.


1) Fetch data from Cache
2) Return the data to the server
4) Persists the data from DB in Cache
Application Server
Cache
3) Fetch data from DB, in case of a Cache miss
Database Server

Types of Cache

In common there are four types of Cache…

1. Application Server Cache

In the "**How does Cache work?**" section we discussed how **application server cache** can be added to a web application. In a web application, let's say a web server has a single node. A cache can be added in in-memory alongside the application server. The user's request will be stored in this cache and whenever the same request comes again, it will be returned from the cache. For a **new request**, data will be fetched from the disk and then it will be returned. Once the new request will be returned from the disk, it will be stored in the same cache for the next time request from the user. Placing cache on the request layer node enables local storage.

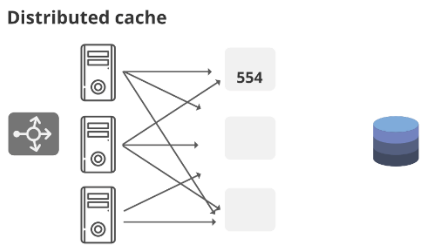
Note: When you place your cache in memory the amount of memory in the server is going to be used up by the cache. If the number of results you are working with is really small then you can keep the cache in memory.

The problem arises when you need to **scale your system**. You add multiple servers in your web application (because one node can not handle a large volume of requests) and you have a **load balancer** that sends requests to any node. In this scenario, you'll end up with a lot of **cache misses** because each node will be unaware of the already cached request. This is not great and to overcome this problem we have two choices: Distribute Cache and Global Cache. Let's discuss that…

2. Distributed Cache

In the distributed cache, each node will have a part of the whole cache space, and then using the consistent hashing function each request can be routed to where the cache request could be found. Let's suppose we have 10 nodes in a distributed system, and we are using a load balancer to route the request then…

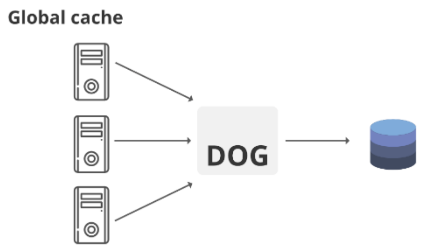
- Each of its nodes will have a small part of the cached data.
- To identify which node has which request the cache is divided up using a consistent hashing function each request can be routed to where the cached request could be found. If a requesting node is looking for a certain piece of data, it can quickly know where to look within the distributed cache to check if the data is available.
- We can easily increase the cache memory by simply adding the new node to the request pool.



3. Global Cache

As the name suggests, you will have a single cache space and all the nodes use this single space. Every request will go to this single cache space. There are two kinds of the global cache

- First, when a cache request is not found in the global cache, it's the responsibility of the cache to find out the missing piece of data from anywhere underlying the store (database, disk, etc).
- Second, if the request comes and the cache doesn't find the data then the requesting node will directly communicate with the DB or the server to fetch the requested data.



4. CDN (Content Distribution Network)

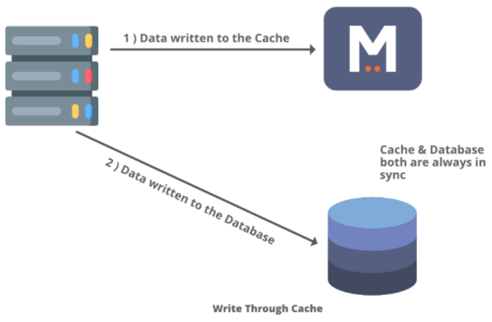
CDN is used where a large amount of static content is served by the website. This can be an HTML file, CSS file, JavaScript file, pictures, videos, etc. First, request ask the CDN for data, if it exists then the data will be returned. If not, the CDN will query the backend servers and then cache it locally.

Cache Invalidation

Caching is great but what about the data which is constantly being updated in the database? If the data is modified in DB, it should be invalidated to avoid inconsistent application behavior. So how would you keep data in your cache coherent with the data from your source of the truth in the database? For that, we need to use some cache invalidation approach. There are three different cache invalidation schemes. Let's discuss that one by one…

1. Write Through Cache

As the name suggests, the data is first written in the cache and then it is written to the database. This way you can keep the consistency of your data between your database and your cache. Every read done on the Cache follows the most recent write.

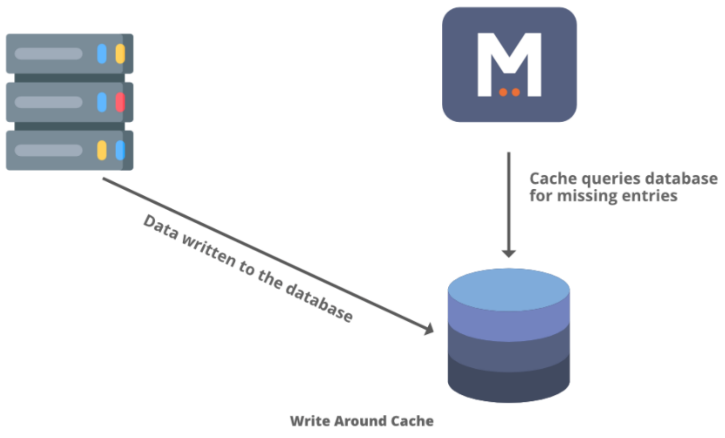

1) Data written to the Cache
Cache & Database both are always in sync
2) Data written to the Database
Write Through Cache

The advantage of this approach is that you minimize the risk of data loss because it's written in both the cache and the database. But the downside of this approach is the higher latency for the write operation because you need to write the data at two places for a single update request. If you don't have a large amount of data then it is fine but if you have heavy write operation then this approach is not suitable in those cases.

We can use this approach for the applications which have frequent re-read data once it's persisted in the database. In those applications write latency can be compensated by lower read latency and consistency.

2. Write Around Cache

Similar to the write-through you write to the database but in this case you don't update the cache. So data is written directly to the storage, bypassing the cache. You don't need to load the cache with data that wouldn't be re-read. This approach reduces the flooded write operation compared to the write-through cache. The downside of this approach is that a read request for recently written data results in a cache miss and must be read from a slower backend. So this approach is suitable for applications that don't frequently re-read the most recent data.


Cache queries database for missing entries
Data written to the database
Write Around Cache

3. Write Back Cache

We have discussed that the write-through cache is not suitable for the write-heavy system due to the higher latency. For these kinds of systems, we can use the write-back cache approach. Firstly flush the data from the cache, and then write the data to the cache alone. Once the data is updated in the cache, mark the data as modified, which means the data needs to be updated in DB later. Later an async job will be performed and at regular intervals, the modified data from the cache will be read to update the database with the corresponding values.

The problem with this approach is that until you schedule your database to be updated, the system is at risk of data loss. Let's say you updated the data in the cache but there is a disk failure and the modified data hasn't been updated into the DB. Since the database is the source of truth, if you read the data from the database you won't get an accurate result.

Eviction Policy

We have discussed so many concepts of caching….now you might have one question in your mind. ***When do we need to make/load an entry into the cache and which data do we need* to *remove from the cache*?**

The cache in your system can be full at any point in time. So, we need to use some algorithm or strategy to remove the data from the cache, and we need to load other data that has more probability to be accessed in the future. To make this decision we can use some cache eviction policy. Let's discuss some cache eviction policies one by one…

1. LRU (Least Recently Used)

LRU is the most popular policy due to several reasons. It is simple, has good runtime performance, and has a decent hit rate in common workloads. As the name suggests this policy evicts the least recently used item first from the cache. When the cache becomes full, it removes the least recently used data and the latest entry is added to the cache.

Whenever you need to add the entry to the cache keep it on the top and remove the bottom-most entries from the cache which is least recently used. The top entries are going to be maybe seconds ago and then you keep going down the list minutes ago, hours ago, years ago and then you remove the last entry (which is least recently used).

Consider the example of any social media site, there is a celebrity who’s made a post or made a comment and everyone wants to pull that comment. So you keep that post on the top of the cache and it stays on the top of the cache depending on how latest the post is. When the post becomes cooler or people stop looking or viewing that post, it keeps getting pushed at the end of the cache, and then it is removed completely from the cache. We can implement the LRU using a doubly-linked list and a hash function containing the reference of the node in the list.

2. LFU (Least Frequently Used)

This policy counts the frequency of each requested item and discards the least frequent one from the cache. So here we count the number of times a data item is accessed, and we keep track of the frequency for each item. When the cache size reaches a given threshold we remove the entry with the lowest frequency.

In real life, we can take the example of typing some texts on your phone. Your phone suggests multiple words when you type something in the text box. Instead of typing the whole word, you have the choice to select one word from these multiple words. In this case, your phone keeps track of the frequency of each word you type and maintains the cache for it. Later the word with the lowest frequency is discarded from the cache when it’s needed. If we find a tie between multiple words then the least recently used word is removed.

3. MRU (Most Recently Used)

This approach removes the most recently used item from the cache. We give preference to the older item to remain in the cache. This approach is suitable in cases where a user is less interested in checking out the latest data or item. Now you might be thinking that most often users are interested in the latest data or entries so where it can be used? Well, you can take the example of the dating app Tinder where MRU can be used.

Tinder maintains the cache of all the potential matches of a user. It doesn’t recommend the same profile to the user when he/she swipes the profile left/right in the application. It will lead to a poor user experience if the same profile will be recommended again and again. So tinder removes the profile from the cache which is observed most recently i.e. either left/right-swiped profiles.

4. Random Replacement

As the name suggests we randomly select an item and discard it from the cache to make space whenever necessary.

Case Study Google Drive

Google drive can be used to upload any size of files from any device, and it can be found on our mobile, laptop, personal computer, etc.

Users should be able to upload and download files/photos from any device. And the files will be synchronized in all the devices that the user is logged in.

If we consider 10Million users, 100 M requests/day in the service, the number of writing and read operations will be huge. For simplification, we're just designing the Google Drive storage. In other words, users can upload and download files, which effectively stores them in the cloud.

The requirements of the system

Functional requirement:

Users should be able to upload and download files from any device. And the files will be synchronized in all the devices that the user is logged in.

Non-Functional requirement:

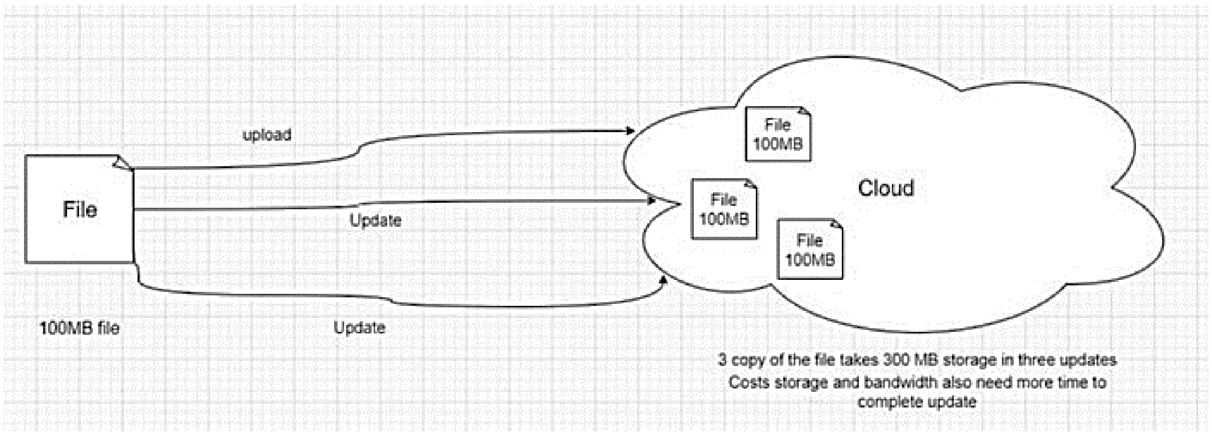
Users can upload and download files from any device. The service should support storing a single large file up to 1 GB. Service should synchronize automatically between devices; if one file is uploaded from a device, it should be synced on all devices that the user is logged in.

Server-side Component Design

Our user in this system can upload and download files. The user uploads files from the client application/browser, and the server will store them. And user can download updated files from the server. So, let's see how we handle upload and download of files for such a massive amount of users.

Upload/Download File:

From the figure, we can see that if we upload the file with full size, it will cost us storage and bandwidth. And also, latency will be increased to complete upload or download.


upload
File
Update
Update
100MB file
File 100MB
File 100MB
File 100MB
Cloud
3 copy of the file takes 300 MB storage in three updates
Costs storage and bandwidth also need more time to complete update

Handle file transfer efficiently:

We may divide each file into smaller chunks. Then we can modify only small pieces where data is changed, not the whole file. In case data upload failure also this strategy will help. We need to divide each file into a fixed size, say 2 MB.

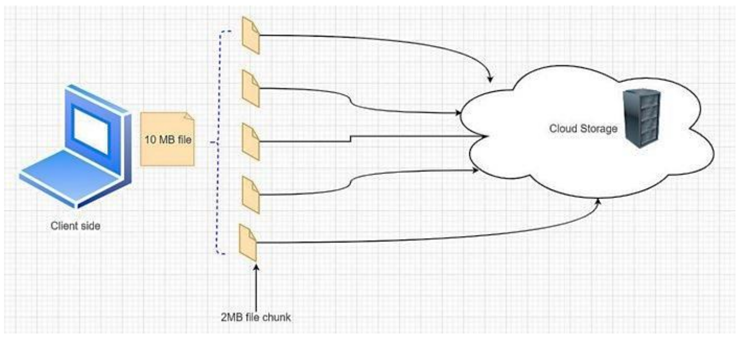


Figure: Divide a file into smaller chunks to optimize storage utilization and bandwidth

Our chunk size needs to be smaller. It will help to optimize space utilization, and network bandwidth is another considering factor while making the decision. Metadata should include the record of each file's chunk information.

Less amount of data transfer between clients and cloud storage will help us achieve a better response time. Instead of transmitting the entire file, we can send only the modified chunks of the files.

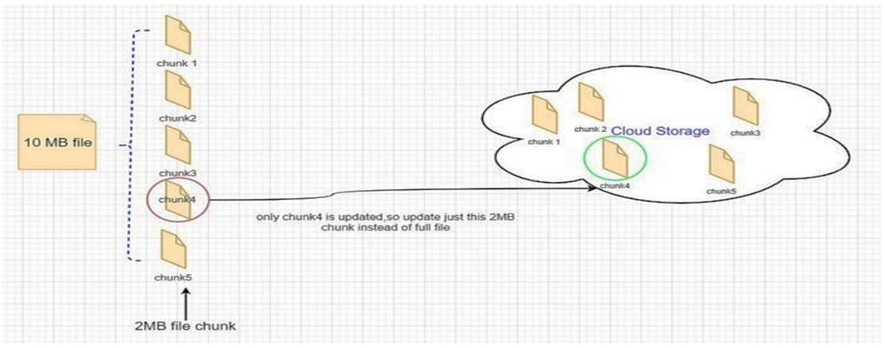


Figure: Transfer only the updated chunk only

In that case, the updated part of the file will be transmitted. We will be dividing files into 2MB chunks and transfer the modified portion of files only, as you can see from the figure.

From the figure above, you may see, instead of updating the whole 10 MB file, we can just update the modified 2MB potion of the file. It will decrease bandwidth consumption and cloud storage for the user. Most importantly, the response time will be faster.

What will happen when the client is offline?

A client component, Watcher, will observe client-side folders. If any change occurs by the user, it will notify the Index Controller (another client component) about the action of the user. It will also monitor if any change is happening on other clients(devices), which are broadcasted by the Notification server.

When the Metadata service receives an update/upload request, it needs to check with the metadata DB for consistency and then proceed with the update. After that, a notification will be sent to all subscribed devices to report the file update.

Metadata Database:

We need a database that is responsible for keeping information about files, users, etc. It can be a relational database like MySQL or NoSQL like MongoDB. We need to save data like chunks, files, user information, etc. in the Database.

Using a SQL database may give us the benefit of the implementation of the synchronization as they support ACID properties.

NoSQL databases do not support ACID properties. But they provide support for scalability and performance. So, we need to provide support for ACID properties programmatically in the logic of our Metadata server for this type of Database.

Synchronization:

Now the client updates a file from a device; there needs to be a component that process updates and applies the change to other devices. It needs to sync the client’s local Database and remote Metadata DB. MetaData server can perform the job to manage metadata and synchronize the user’s files.

Message Queue:

Now think about it; such a huge amount of users are uploading files simultaneously, how the server can handle such a large number of requests. To be able to handle such a huge amount of requests, we may use a message queue between client and server.

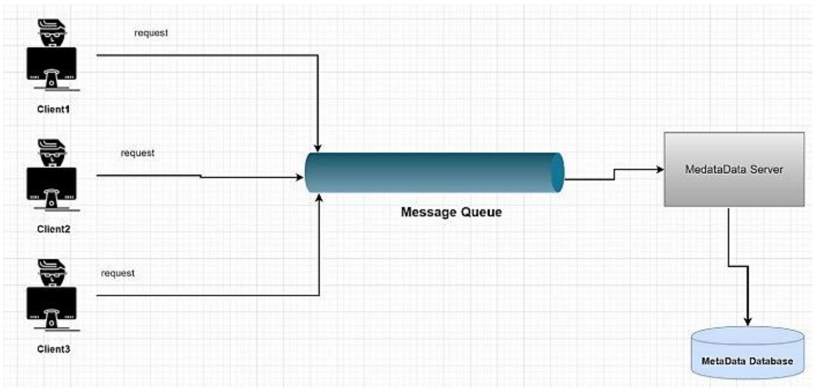


Figure: The messaging queue provides scalable request queuing and change notifications to support a high number of clients using a pull or push strategy.

The message queue provides temporary message storage when the destination program is busy or not connected. It provides an **asynchronous communications protocol.** It is a system that puts a message onto a queue and does not require an immediate response to continue processing. RabbitMQ, Apache Kafka, etc. are some of the examples of the messaging queue.

In case of a message queue, messages will be deleted from the queue once received by a client. So, we need to create several Response Queues for each subscribed device of the client.

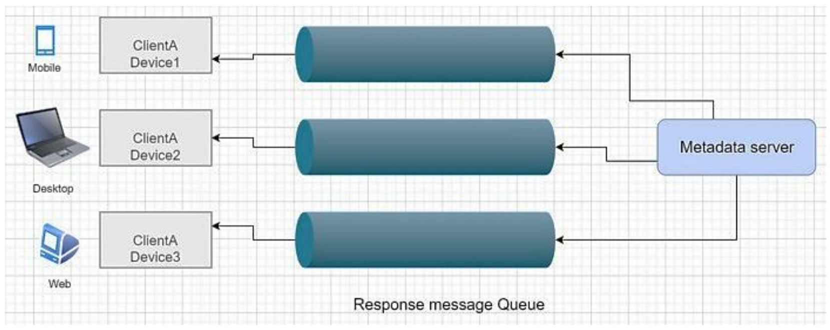


Figure: Response message queue for each device type

For a massive amount of users, we need a scalable message Queue that supports asynchronous message-based communication between client and synchronization service. The service should be able to efficiently store any number of messages in a highly available, reliable, and scalable queue. Example: apache Kafka, rabbitMQ, etc.

Cloud Storage:

Nowadays, there are many platforms and operating systems like smartphones, laptops, personal computers, etc. They provide mobile access from any place at any time.

If you keep files in the local storage of your laptop and you are going out but want to use it on your mobile phone, how can you get the data? That's why we need cloud storage as a solution.

It stores files (chunks) uploaded by the users. Clients can interact with the storage through **File Processing Server** to send and receive objects from it. It holds only the files; Metadata DB keeps the data of the chunk size and numbers of a file.

File processing Workflow:

Client A uploads chunk to cloud storage. Client A updates metadata and commits changes in MetadataDB using the Metadata server. The client gets confirmation, and notifications are sent to other devices of the same user. Other devices receive metadata changes and download updated chunks from cloud storage.

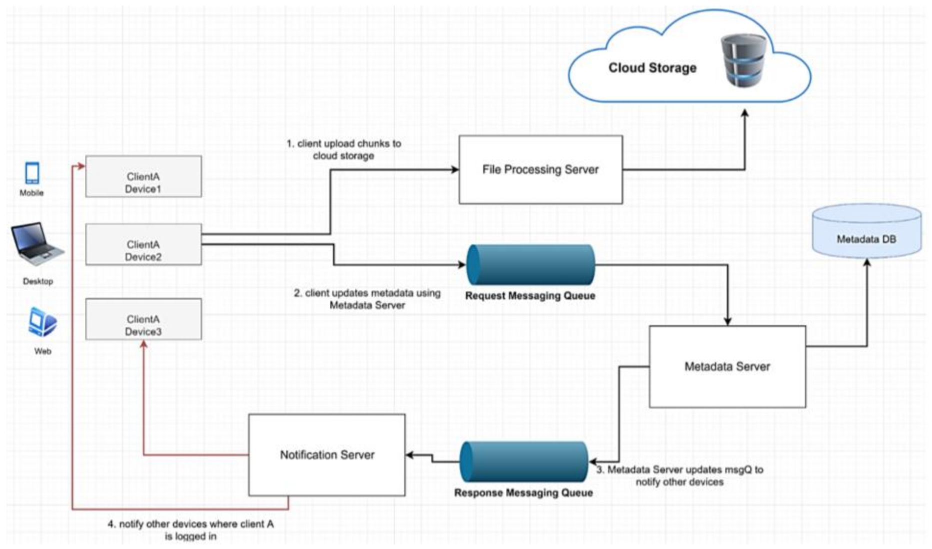

Cloud Storage
1. client upload chunks to cloud storage
File Processing Server
Mobile
ClientA Device1
Desktop
ClientA Device2
Web
ClientA Device3
Metadata DB
2. client updates metadata using Metadata Server
Request Messaging Queue
Metadata Server
Notification Server
Response Messaging Queue
3. Metadata Server updates msgQ to notify other devices
4. notify other devices where client A is logged in

Figure: Workflow of File Process for client A

Scalability

We need to partition the metadata database so that we can store information about 1 million users and billions of files/chunks. We can partition data to distribute the read-write request on servers.

MetaData Partitioning:

i) We can store file-chunks in partitions based on the first letter of the File Path. For example, we keep all the files starting with the letter 'A' in one partition and those beginning with the letter 'B' into another partition and so on. *This is called range-based partitioning*. Less frequently occurring letters like 'Z' or 'Y,' we can combine them into one partition.

The main problem is that some letters are common in case of a starting letter. For example, if we put all files starting with the letter 'A' into a DB partition, and we have too many files that begin with the letter 'A,' so that that we cannot fit them into one DB partition. In such cases, this approach will have a disadvantage.

ii) We may also partition based on the hash of the 'fileId' of the file. Our hash function will randomly generate a server number, and we will store the file in that server. But we might need to ask all the servers to find a suggested list and merge them together to get the result. So, response time latency might be increased.

If we use this approach, it can still lead to overloaded partitions, which can be solved by using Consistent Hashing.

Caching:

As we know, **caching is a common technique for performance. T**his is very helpful to lower the latency. The server may check the cache server before hitting the Database to see if the search list is already in the cache. We can't have all the data in the cache; it's too costly.

When the cache is full, and we need to replace a chunk with a newer chunk. Least Recently Used (LRU) can be used for this system. In this approach, the least recently used chunk is removed from the cache first.

Security:

In a file-sharing service, the privacy and security of user data are essential. To handle this, we can store the permissions of each file in the metadata database to give perm what files are visible or modifiable by which user.

Client-Side:

The client application (web or mobile) transfers all files that users upload in cloud storage. The application will upload, download, or modify files to cloud storage. A client can update metadata like rename file name, edit a file, etc.

The client app features include upload, download files. As mentioned above, we will divide each file into smaller chunks of 2MB so that we transfer only the modified chunks, not the whole file.

In case any conflict arises due to the offline status of the user, the app needs to handle it. *Now, we can keep a local copy of metadata on the client-side to enable us to do offline updates.*

The client application needs to detect if any file is changed in the client-side folder. We may have a component, Watcher. It will check if any file changes occurred on the client-side.

How would clients know change is done in cloud storage?

The client can periodically check with the server if there is any change, which is a manual strategy. But if the client frequently checks server changes, it will be pressure for the server, keep servers busy.

We may use HTTP Long polling technique instead. In this technique, the server does not immediately respond to client requests. Instead of sending an empty response, the server keeps the request open. Once new information is ready, then the server sends a response to the client.

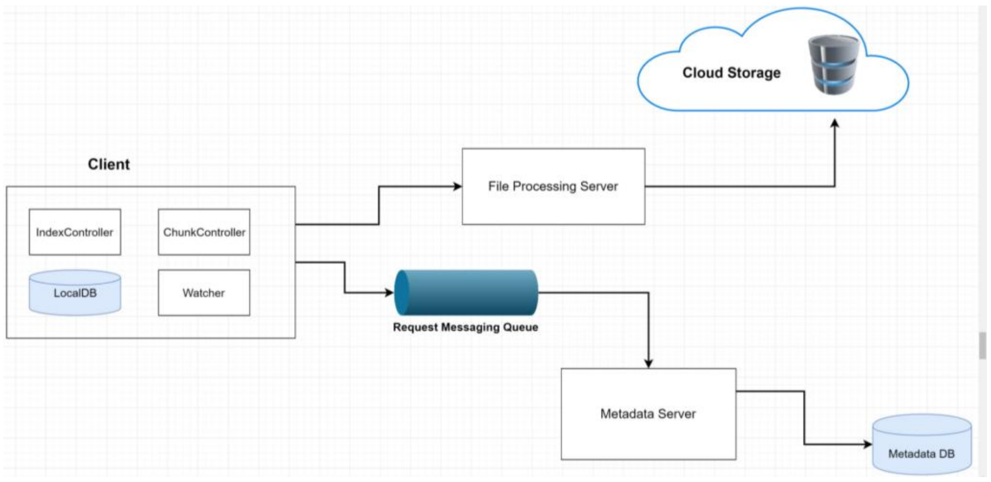


Figure: Client App requests to metadata server for update metadata info

We can divide client application into these parts:

✓ Local Database will keep track of all the files, chunks, directory path, etc. in the client system.

✓ The Chunk Controller will split files into smaller pieces. It will also perform the duty to reconstruct the full file from its chunks. And this part will help to determine only the latest modified chunk of a file. And only modified chunks of a file will be sent to the server, which will save bandwidth and server computation time.

✓ *The Watcher will observe client-side folders, and if any change occurs by the user, it will notify the Index Controller about the action of the user. It will also monitor if any change is happening on other clients(devices), which are broadcasted by Synchronization service.*

✓ The Index controller will process events received from the Watcher and update the local Database about modified file-chunk information. It will communicate with the Metadata service to transfer changes to other devices and update the metadata database. This request will be sent to the metadata service via the message request queue.

Below is the full diagram of the system:

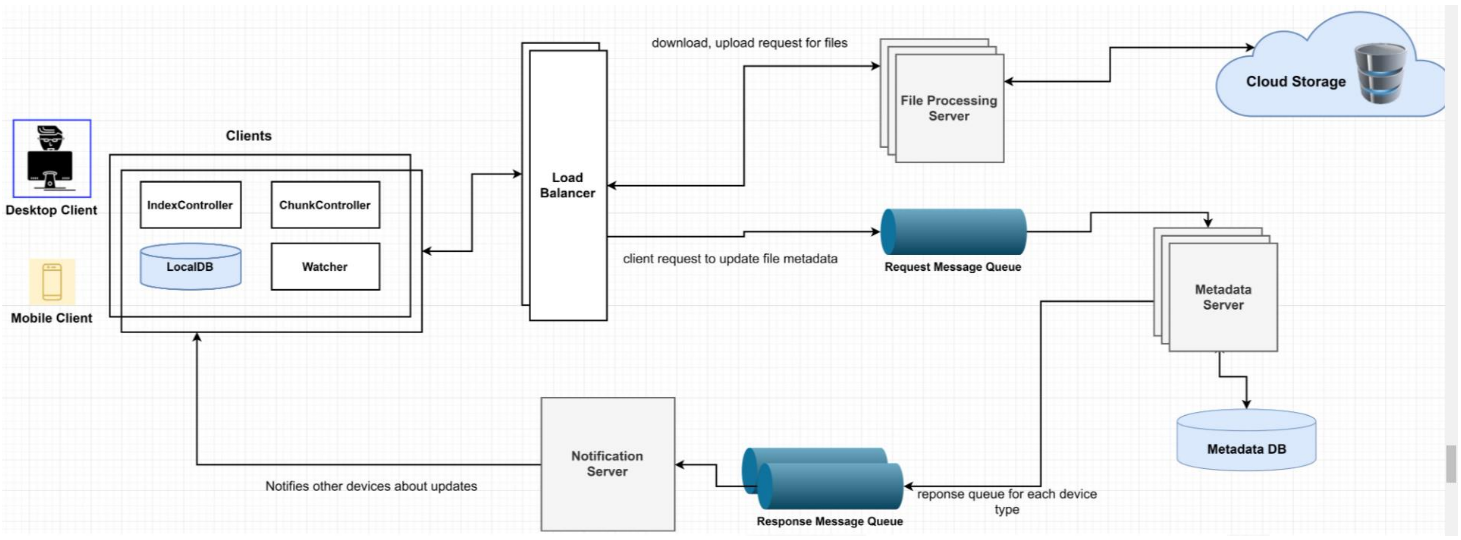


Figure: System Design of Google drive