

How to approach System Interview questions

- Consider your interviewer as a team member and take this round as an opportunity to work with him where you both are supposed to solve a real-world problem related to your company's goal but here you need to take **ownership** and lead everything.
- The main purpose of this round is to understand how capable you are of building a large-scale system and your thought process behind designing a service. Clarity of thoughts matters a lot because if you can explain it to the interviewer, you can do this in your team as well.
- One of the good things for you in this round is that you are supposed to come up with the best solution for all kinds of **open-ended problems** instead of accurate solutions. Your ability to articulate your thoughts matters more than the final design you present to them.

1. Understand the Goal and Gather All the Requirements

You need to first understand your end goal before jumping to the solution so gather all the basic requirements from your interviewer. Ask relevant questions to clear your doubts. Design questions are basically open-ended conversation which doesn't have one correct solution so it's good to start with some basic assumptions.

- What's the end goal of the system or service?
- Who are the end-users? How they will use the service? What features does your interviewer want you to include and what exclude?
- What should be the input and its output or final outcome?

A product like Facebook, Twitter, or Reddit is a well-known product so even if you know how to design such kind of system still it's your responsibility to share your assumptions and discuss with the interviewer what features they care about and what they don't. They may want you to include some features which don't exist in this kind of system or they may tell you to exclude some features. So make sure you have a better understanding of all the requirements and features. Consider the example of designing twitter. Some of the questions are who can post a tweet? who can read the tweet? who can follow the user? like, comments, pictures, active users, and total users, and discuss other features your interviewer wants you to include.

2. System Interface Definition and Establish Scope

After the first step, you need to identify what kind of APIs your system needs to get the job done based on whatever features you have included. Also, discuss the scope and availability of the system, and discuss some relevant questions like do you both care about only end-to-end experience or just the API? Client support (mobile, web, etc). Authentication, analytics, integration, performance, etc. Is the system going to work if the host is down or the entire data system is down? So discuss with the interviewer how much availability he/she cares about the system. From both, the above two steps make sure you know the exact scope of the problem and the complexity of the system.

3. Scalability Estimation:

You design a service that works for a hundred users, but is it going to work for a thousand users or million users? Is it going to scale and work fine as we add more users or more requests? Basically, you need to consider the same feature for different scales and it's very important to get the right scale because different answer requires a different design. Scalability also helps in load balancing, caching, and partitioning so you can ask questions like:

- What's the limit of the data or network or bandwidth we need to care about?
- How much storage do we need?
- What's the average response time?

4. Start with High-level Components then move to Detailed Design

Start to cover the end-to-end process based on your goal, so identify each component to solve the actual problem or to implement your complete system. How do all the components come together to meet the actual requirement? Below are some quick points to guide you properly while explaining the components:

- Divide your complete system into 6-7 **higher level or core components**.
- Drill down further and discuss the role and responsibility of each component also how they are going to interact or communicate with each other. While explaining the components your interviewer may guide you towards 1-2 components and wants you to explain those components in depth. So there you need to further discuss that in detail.
- Discuss the frontend, backend, networking, caching, load balancing, queueing, database, external API calls, user interaction, offline processes, etc.
- Tell the interviewer what technologies or databases you can use in your system.

5. Tradeoffs and Resolving Bottleneck

Design interviews are open-ended conversations so there is no single answer and every decision will have a **tradeoff** when you will present your architecture and your thought process to the interviewer. The **complex system always requires compromises** so you need to tell them different approaches, their pros, and cons, and **why would you choose one over another**.

- Which database fits in your system and why?
- Why would you choose a specific technology in different layers or components?
- Which frameworks can be good for your design and which one do you need to choose?
- What are the different security options available to keep your data safe and which one you would choose?

You also need to think about and resolve the bottlenecks like what kind of **failure** can occur in your system and what's the solution for that. Do you need to keep a backup or you will take the help of any other resources? Do you have any backup for your data in case your server crashes and you lose the entire data? How would you monitor the performance of the

service? If any component fails then what's the solution to run your system smoothly and properly? Basically, you should have an organized and clear plan to deal with all these kinds of critical failures in your system. **Quick Tips:**

- Try to follow the **80-20 rule** during your interview, where 80% of the time you will be speaking and explaining everything and 20% of the time your interviewer.
- Don't use **buzzwords** and pretend to be an expert if you don't know something. You read some blog posts or a few topics today and tomorrow is your interview, during your interview if you throw some buzzwords like "No-SQL", "Mongo DB" and "Cassandra" then it may backfire on you. You can't make a fool of the interviewer who is an industry expert, always consider that your interviewer may ask for more details and justification so if you are using technology X or database Y then "why?", prepare yourself for this kind of question.
- **Do not go into detail prematurely.** Many times it happens when a candidate starts explaining one part of the system, they go into too much detail about the component and forget about the strict timeframe and other components. Maybe the interviewer wants you to stop somewhere where they don't need too much detail. So to avoid this mistake wait for the interviewer's feedback or response. They will give you some hints or will direct you to whatever part of the system they want you to explain further.
- **Don't have a set architecture in mind** like MVC or event-driven and try to fit the requirement somehow in that architecture. Maybe it's not suitable as per the requirement. Requirements may change during the interview to test your flexibility so try to avoid this mistake.
- Be honest during your interviews and if you have never used technology X then you don't need to be fake in that situation. Try to find common solutions and show them your **honesty, confidence, and willingness to learn** something. That will make a good impression on the interviewer.
- Your practical experience, your knowledge, understanding of modern software system, and how you express yourself clearly during your interview matters a lot to designing a system successfully.

Five Tips to Crack Low-Level System Design Interviews

Low-level design is very important when you have to design software-level components. It is one of the important components of software design and at the time of requirements you have to collect all the necessary points which are required to design the system. So, you can say low-level is a step-by-step refinement process.

The main goal of low-level design is to *design the internal logical design for code*. After that, it is easy to understand internal low-level design architecture.

Key points:

- "No right or wrong" answer for the Design Question
- Understand and Read the Question properly

- List the requirements
- Think and be clear with your answer
- Practice, Practice, and Practice

1. No Right or Wrong Answer For the Design Question

In industries getting a better design of the product is an iterative process and a lot of times refactoring of code also happens to improve it. However, in the interviews, we only have limited time to come up with a design that should be good enough to convince the interviewer. Always start designing from the basic entity in your system and then iteratively proceed towards the higher components. That way chances of missing out on the basic components are less in comparison to that if you start with big entities in starting.

2. Understand and Read the Question Properly

Often in the interviews, it happens that you get the question you have read earlier, don't rush, and start to write whatever you have just read, first understand the question completely, and then you must proceed. It may happen that the interviewer might expect some different requirements from what you read. Hence it is very important to be calm and listen to the interviewer patiently.

If you have read the answer prior to your interview it will definitely give you a head start, but don't just sound as if you have mugged up the answer. Also, there are high possibilities that some of the requirements which you have read are different during the interview and in that case, if you just blindly copy-paste your already read answer, your interview will get a negative impression and the chances for you to crack the interview will fall drastically. It's very important for you to understand the whole picture and then think before you jump to a conclusion.

3. List the Requirements

Don't assume, just clarify with your interviewer about all the requirements and then write them down, this will help you to avoid confusion later and it will also help you proceed step by step.

Once you have all the requirements clarified, they will become the base for design and you will be able to proceed step by step accordingly. Without listing the requirements, you will be lost during the designing process as you will not have a clear reference for what to do next. It is very-very essential for the design that you have gathered all the requirements.

Once you have all the requirements clarified, they will become the base for design and you will be able to proceed step by step accordingly. Start designing your system on the basis of listed requirements one by one.

4. Think and Be Clear With Your Answer

At the time of the interview, after getting all the requirements, think and discuss with your interviewer. But many times it happens you are telling the wrong answer but if you have to

think correctly then your interviewer can guide you. So, you can give the exact answer whatever the interviewer expected and give you some hints, so discuss.

Remember the time in the interview is limited to 45 minutes to 1 hour and if you think that first, you will think of coming up with the whole design in one go and then explain the complete to your interviewer, believe me, it's a blunder and that will not be successful, especially in design interviews. Because there are chances where you can make mistakes during the process and if you will be continuously discussing it with your interviewer then he or she will definitely give you hints and tell you that you are going in the wrong direction, do so and so to correct it. Also, it will save you a lot of time which you would be spending on correction and explanation later.

Always remember that your interviewer is there for hiring you and not for rejecting you, so feel free to seek help and hints if required, however, do not make it a habit at each and every step, seeking too many hints and help can also be negative sometimes, so you must be aware of each and every scenario.

5. Practice, Practice, and Practice

Just don't underestimate the power of practice and being consistent, the only key to becoming better in anything is practice. So, before your interviews, you must have to read the notes which you have prepared or you can read the materials you have available to you. Sometimes it happens that whatever you have practiced the same question is being asked, in that case, you will always have an edge, but as we mentioned earlier do not just start writing the same blindly.

Take a few questions and try to solve them and come up with the design on your own first and then if the solution is available somewhere just compare with that what is being missed or what can be done in a better way. Just do not go and directly read the solution, that way the learning curve will not be exponential. First, try and then go for a solution.

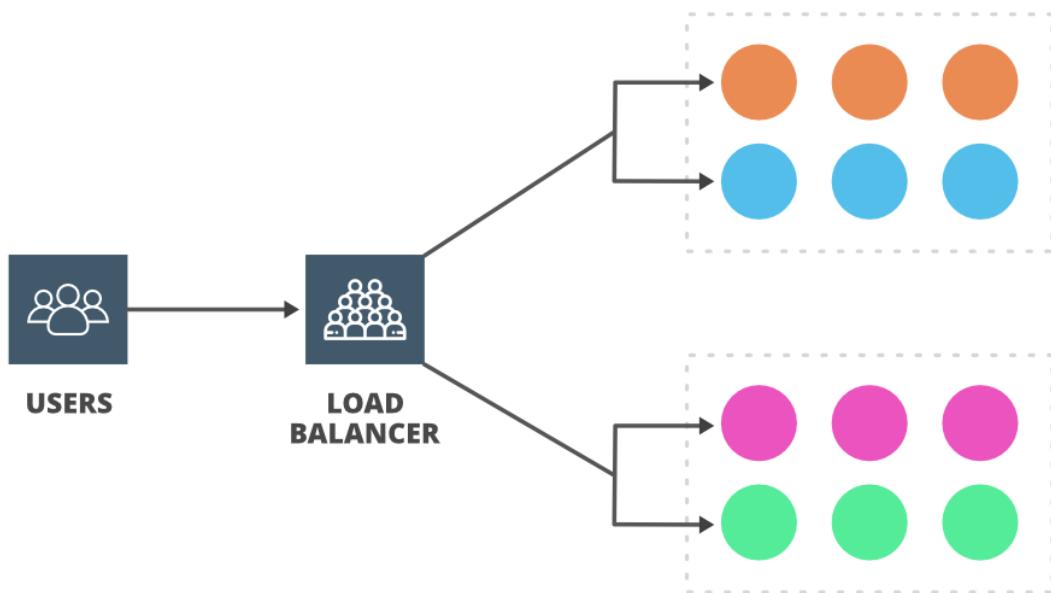
Often people make the mistake of just reading the solution and going for the interviews, believe me, that's not a very good habit, always try yourself first and then watch out for the available solution.

Five System Design Concepts

1. Load Balancing

In a system, a server has a certain amount of capacity to handle the load or request from users. If a server receives a lot of requests simultaneously more than its capacity then the throughput of the server gets reduced and it can slow down. Also, it can fail (no availability) if it continues for a longer period. You can add **more servers (horizontal scaling)** and resolve this issue by distributing the number of requests among these servers. Now the question is who is going to take ownership of distributing the request and balancing the load. Who is going to decide which request should be allocated to which server to ease the burden of a single server? Here comes the role of the load balancer.

A load balancer's job is to **distribute traffic** to many different servers to help with **throughput, performance, latency, and scalability**. You can put the **load balancer** in **front** of the **clients** (it can be also inserted in other places) and then the load balancer will route the incoming request across multiple web servers. In short, load balancers are traffic managers and they take responsibility for the availability and throughput of the system. Nginx, Cisco, TP-Link, Barracuda, Citrix, and Elastic Load Balancing from AWS, are some popular load balancers available in the market.



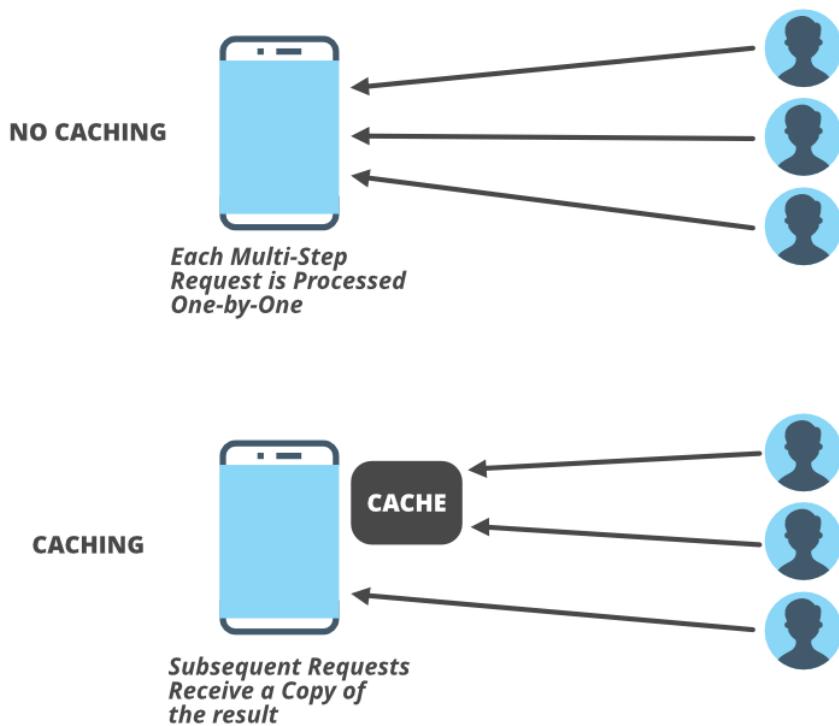
2. Caching

We talked about the load on the servers in the load balancing section but one thing you need to know is usually your web server is not the first to go down, in fact, quite often your **database server may be under high loads** for lots of writes or reads operations. Quite often we hit the database for various queries and joins which slows down the performance of the system. To handle these queries and lots of reads and writes, caching is the best technique to use.

Do you go to your nearest shop to buy some essentials every time you need something in your kitchen? absolutely no. Instead of visiting the nearest shop every time we want to buy and store some basics in our refrigerator and our food cupboard. This is caching. The cooking time gets reduced if the food items are already available in your refrigerator. This saves a lot of time. The same things happen in the system. Accessing data from primary memory (RAM) is faster than accessing data from secondary memory (disk). By using the caching technique you can speed up the performance of your system.

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. This process reduces the workload on the backend

servers. Caching helps in reducing the network calls to the database. Some popular caching services are **Memcache**, **Redis**, and **Cassandra**. A lot of websites use CDN (content delivery network) which is a global network of servers. CDN caches static assets files like images, javascript, HTML, or CSS and it makes accessing very fast for the users. You can insert caching in on the client (e.g. browser storage), between the client and the server (e.g. CDNs), or on the server itself.



3. Proxies

Quite often you may have seen some notification on your PC to add and configure the proxy servers but what exactly proxy servers are and how does it work? Typically proxy servers are some bit of code or intermediary piece of hardware/software that sits between a client and another server. It may reside on the user's local computer or anywhere between the clients and the destination servers. A proxy server receives requests from the client and transmits them to the origin servers, then forwards the received response from the server to the originator client. In some cases, when the server receives the request the IP address is not associated with the client but is of the proxy server. This happens when the proxy server hides the identity of the client.

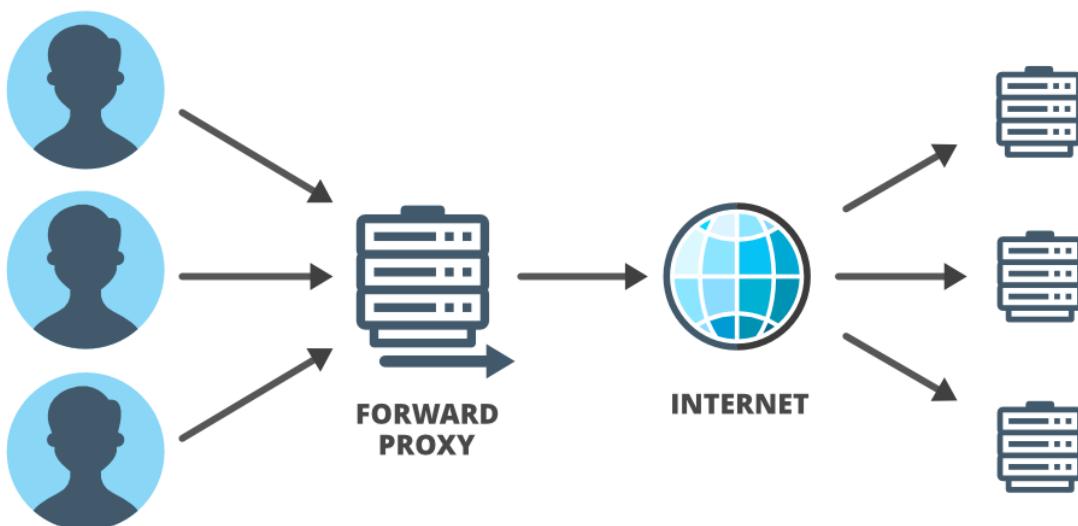
In general, when people use the term proxy they refer to 'forward proxy'. 'Forward proxy' is designed to help users and it acts on behalf of (substitute for) the client in the interaction between client and server. It forwards the user's requests and acts as a personal representative of the user. In system design, especially in complex systems, proxies are very

useful, particularly ‘reverse proxies’ are useful. ‘Reverse proxies’ are the opposite of ‘forward proxy’. A **reverse proxy acts on behalf of a server and is designed to help servers**.

In the ‘forward proxy’ server won’t know that the request and response are routed through the proxy, and in a reverse proxy, the client won’t know that the request and response are traveling through a proxy. A ‘reverse proxy’ can be assigned a lot of tasks to help the main server and it can act as a **gatekeeper**, a **screener**, a **load-balancer**, and an all-around **assistant**.

Typically proxies are used to handle requests, filter requests or log requests, or sometimes transform requests (by adding/removing headers, encrypting/decrypting, or compression). It helps in coordinating requests from multiple servers and it can be used to optimize request traffic from a system-wide perspective.

Forward Proxy:

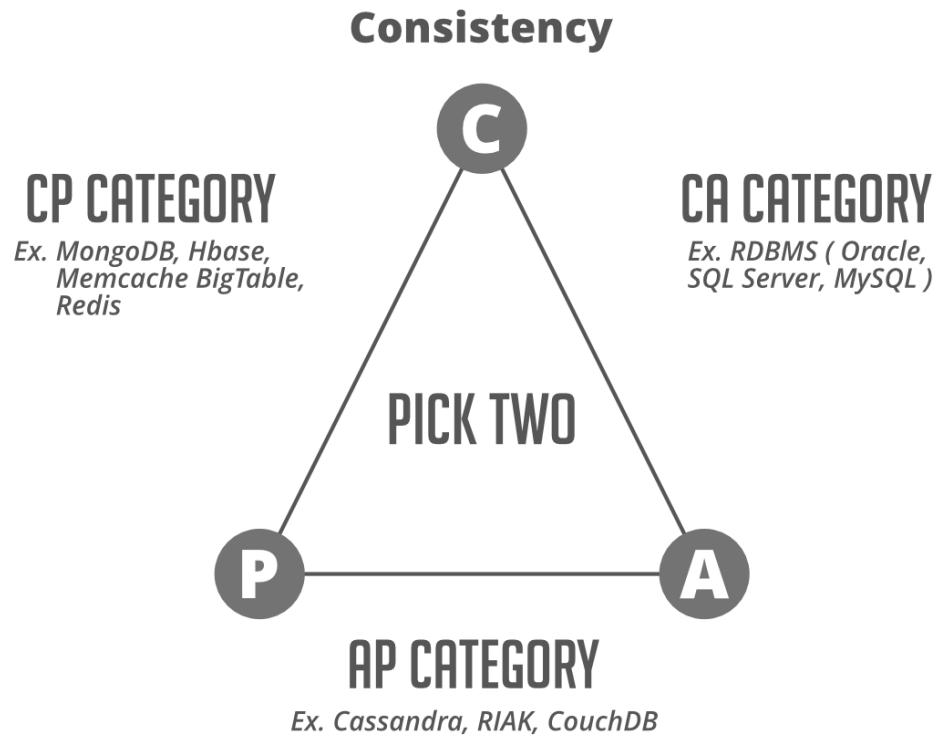


Reverse Proxy:



4. CAP Theorem

CAP stands for **Consistency**, **Availability**, and **Partition tolerance**. The theorem states that you cannot achieve all the properties at the best level in a single database, as there are natural trade-offs between the items. You can only **pick two out of three** at a time and that totally depends on your priorities based on your requirements. For example, if your system needs to be available and partition tolerant, then you must be willing to accept some latency in your consistency requirements. **Traditional relational databases** are a natural fit for the **CA** side whereas **Non-relational database** engines mostly satisfy **AP** and **CP** requirements.



- **Consistency** means that any read request will return the most recent write. Data consistency is usually “strong” for SQL databases and for NoSQL databases consistency may be anything from “eventual” to “strong”.
- **Availability** means that a non-responding node must respond in a reasonable amount of time. Not every application needs to run 24/7 with 99.999% availability but most likely you will prefer a database with higher availability.
- **Partition tolerance** means the system will continue to operate despite network or node failures.

Keep in mind this CAP theorem in your system design interview. Take the decision depending on the type of application and your priorities. Is it actually ok if your system goes down for a few seconds or a few minutes, if not then availability should be your prime concern. If you’re dealing with something with real transactional information like a stock transaction or financial transaction you might value consistency above all. Try to choose the technology that is best suited to the trade-offs that you want to make.

Note: CA systems are not defined for distributed systems. However, in the case of a single node setup, you can get CA capabilities. Further, distributed systems have to support partition tolerance due to network failures. Hence, either you choose Consistency or Availability i.e build a CP or an AP system.

5. Databases

In the system design interviews, it’s not uncommon that you ought to be asked to design the database schema about what tables you may be using. what the primary key is going to look

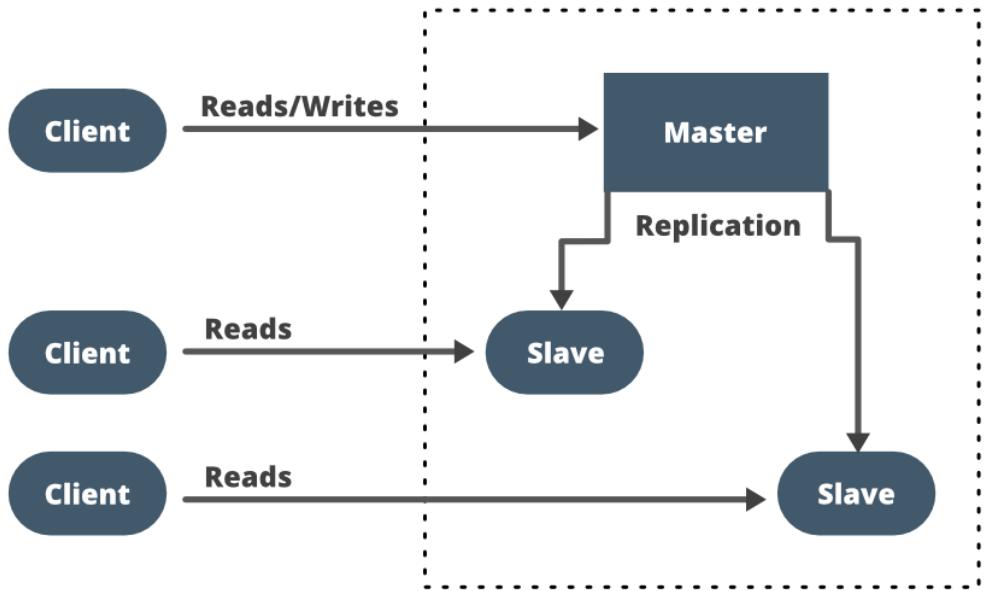
like and what are your indices? You also need to choose the different types of storage solutions (relational or non-relational) designed for different use cases.

- **Database Indexing:** Database indexes are typically a data structure that facilitates the fast searching of databases..but how? let's understand with an example. Suppose you have a database table with 200 million rows and this table is used to look up one or two values in each record. Now if you need to retrieve a value from a specific row then you need to iterate over the table which can be a time-consuming process especially if it's the last record in the table. We can use indexing for these kinds of problems.

Basically, indexing is the way of sorting a number of records on multiple fields. When you add an index in a table on a field, it creates another data structure that holds the field value, and a pointer to the record it relates to. This index structure is then sorted, allowing binary searches to be performed on it. If you have 200 million records in a table with names and ages and you want to retrieve lists of people belonging to an age group, then you need to add an index on the age attribute into the database.

- **Replication:** What will happen if your database handles so much load? It will crash at a certain point and your entire system will stop working because all the requests depend on data in the servers. To avoid this kind of failure we use replication which simply means duplicating your database (master) and allowing only read operation on these replicas (slave) of your database. Replication solves the availability issue in your system and ensures redundancy in the database if one goes down. You created the replica (slave) of your database but how would you pull the data from the original (master) database? how would you synchronize the data across the replicas, since they're meant to have the same data?

You can choose a synchronous (at the same time as the changes to the main database) or an asynchronous approach depending on your needs. If it's asynchronous then you may have to accept some inconsistent data because changes in the master database may not reflect in the slave before it crashes. If you need the state between the two databases to be consistent then the replication needs to be rapid and you can go with a synchronous approach. You also need to ensure that if the write operation to the replica fails, the write operation to the main database also fails (atomicity).



- **Sharding or Data Partitioning:** Replication of data solves the availability issue but it doesn't solve the throughput and latency issues (speed). In those cases, you need to share your database which simply means 'chunking down' or partitioning your data records and storing those records across multiple machines. So sharding data breaks your huge database into smaller databases. Take the example of Twitter where a lot of writing is performed. To handle this case you can use database sharding where you split up the database into multiple master databases. There are mainly two ways to shard your database- horizontal sharding and vertical sharding. In **vertical sharding**, you take each table and **put each table into a new machine**. So if you have a **user table**, a **tweets table**, a **comments table**, and a **user supports table** then **each of these will be on different machines**. Now, what if you have a single tweet table and it is very large? In that case, you can use **horizontal sharding** where you **take a single table and you split that across multiple machines**. You can take some sort of key like a user ID you can break the data into pieces and then you can allocate data to different machines. So horizontal partitioning depends on one key which is an attribute of the data you're storing to partition data.

Original Table

Roll. Number	First Name	Last Name	House Number
1	Deepak	Soni	141
2	Ishant	Garg	1521
3	Monu	Sharma	45
4	Sonu	Sharma	485

Vertical Partitions

VP1

Roll. Number	First Name	Last Name
1	Deepak	Soni
2	Ishant	Garg
3	Monu	Sharma
4	Sonu	Sharma

VP2

Roll. Number	House Number
1	141
2	1521
3	45
4	485

Horizontal Partitions

HP1

Roll. Number	First Name	Last Name	House Number
1	Deepak	Soni	141
2	Ishant	Garg	1521

HP2

Roll. Number	First Name	Last Name	House Number
3	Monu	Sharma	45
4	Sonu	Sharma	485

Monolithic vs Microservices architecture

Monolithic Architecture:

Monolith — means a **single** unified unit, all in one piece. Monolithic is an all-in-one architecture composed of software aspects as a single unit. These types of architectures are developed to manage multiple tasks as a single unit.

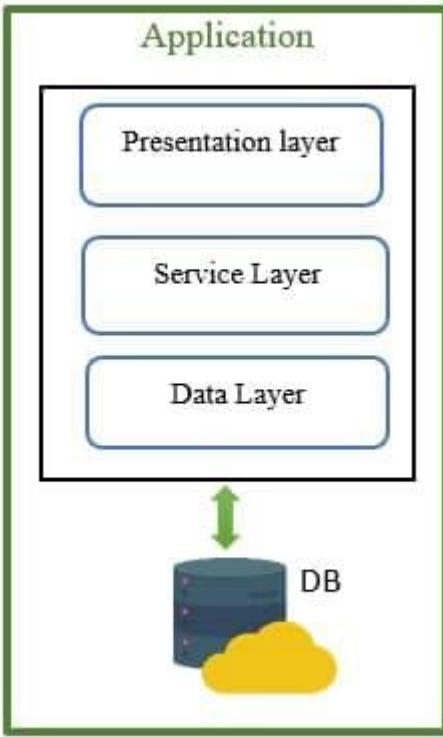


Figure 1: Monolithic architecture model

Monolith is a unified designed model architecture for various business software applications. Mainly, it has three parts:

- Presentation (client) Layer — is for handling HTTP requests and responses sent as HTML or JSON/XML
- Service/Business Layer — Business Logic Layer
- Data (Persistence) Layer — Data Access Layer

You can see the above example of three main layers talking to each other to retrieve data from a database which is the main component of Monolithic architecture.

Advantages and disadvantages of Monolithic Architecture:

Benefits:

- Developing simple legacy applications is very easy, the structure is simple, and a forward process with simple methodologies for the development life cycle.
- Deploying and managing is relatively easier and simple compared to Microservices architecture.
- Testing is much easier with a system using ready testing tools like Selenium.

Drawbacks:

- Maintenance — when its size gets large, difficult to manage it

- Deployment — increases when application size increases
- For a small change, the whole application must be redeployed
- Difficult to new technology adoption
- Not reliable, a small bug in any module may bring down the whole application

Microservices Architecture:

Microservices — means an architecture where it consists of a collection of smaller independent services or units communicating with each other directly through defined methods APIs (Application Programming Interfaces).

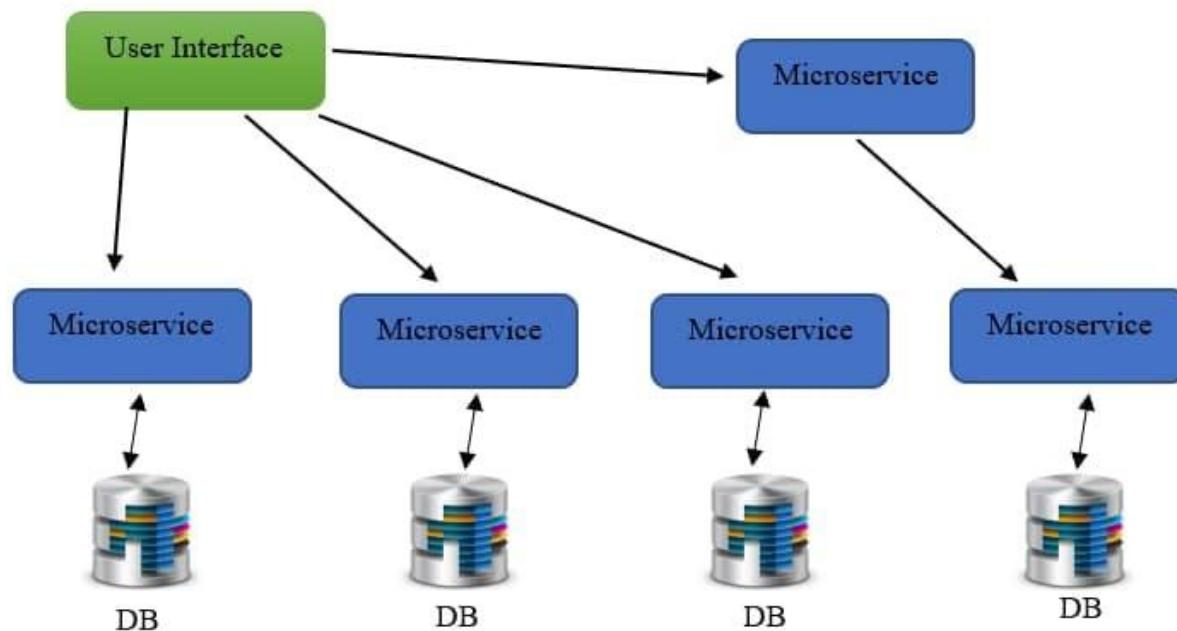


Figure 2: Microservice architecture model

Advantages and disadvantages of Microservices Architecture:

Benefits:

- Independent components — all services can be updated and deployed independently. A single bug in some module can't impact the entire application
- Better scalability — each service module can be scaled independently which is time-consuming
- Easy to manage — with smaller size, much easier to manage compared to a whole monolithic application.
- Easy understanding — split into smaller size and simpler services, much easier to understand and manage

- Flexible in choosing technology

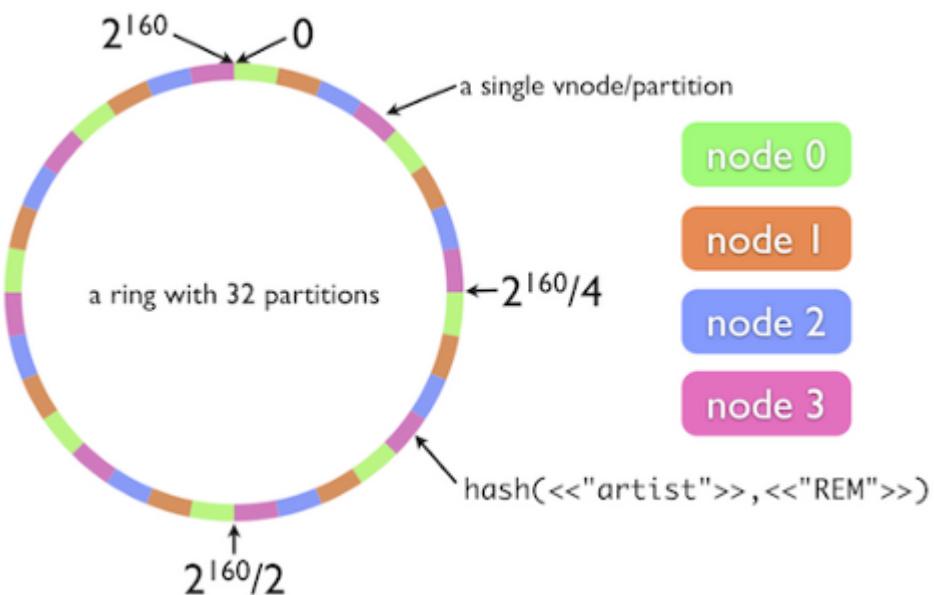
Drawbacks:

- As a distributed system, it's more complex than monolithic applications
- Complicated independent deployment
- Much costly in terms of network usage as services need to interact with each other
- Less secure relatively to monolithic applications
- Debugging and testing is difficult due to control flows over microservices, where exactly error occurred

Choosing a monolithic architecture for a small team with start-ups, simple applications, quick launch applications are recommended, whereas choosing microservices architecture for microservices expertise, complex and scalable applications, and enough engineering skills with multiple teams, are suggested.

Consistent Hashing

Concepts and considerations for Consistent Hashing in System Design



1. Concepts

In a distributed system, consistent hashing helps in solving the following scenarios:

1. To provide elastic scaling (a term used to describe dynamic adding/removing of servers based on usage load) for cache servers.
2. Scale-out a set of storage nodes like NoSQL databases.

2. Why do we need Consistent Hashing?

Our goal is to design database storage (can be other systems) system such that:

1. We should be able to distribute the incoming queries uniformly among the set of “n” database servers
2. We should be able to dynamically add or remove a database server
3. When we add/remove a database server, we need to move the minimal amount of data between the servers

Consistent hashing solves the **horizontal scalability problem** by ensuring that every time we scale up or down, we DO NOT have to re-arrange all the keys or touch all the database servers.

3. How does it work

- Consistent hashing maps a key to an integer.
- Imagine that the integers in the range are placed on a ring such that the values are wrapped around.
- Given a list of servers, hash them to integers in the range.
- To map a key to a server:
 1. Hash it to a single integer.
 2. Move clockwise on the ring until finding the first cache it encounters.
- When the hash table is resized (a server is added or deleted), only k/n keys need to be remapped (k is the total number of keys, and n is the total number of servers).
- To handle hot spots, add “virtual replicas” for caches.
 1. Instead of mapping each cache to a single point on the ring, map it to multiple points on the ring (replicas). This way, each cache is associated with multiple portions of the ring.
 2. If the hash function is “mixes well,” as the number of replicas increases, the keys will be more balanced.

Consistent hashing facilitates the distribution of data across a set of nodes in such a way that minimizes the re-mapping/ reorganization of data when nodes are added or removed. Here's how it works:

1. **Creating the Hash Key Space:** Consider we have a hash function that generates integer hash values in the range $[0, 2^{32}-1]$
2. **Representing the hash space as a Ring:** Imagine that these integers generated in step # 2 are placed on a ring such that the last value wraps around.
3. **Placing DB Servers in Key Space (HashRing):** We're given a list of database servers to start with. Using the hash function, we map each DB server to a specific

place on the ring. For example, if we have 4 servers, we can use a hash of their IP addresses to map them to different integers using the hash function. This simulates placing the four servers into a different place on the ring as shown below.

4. Determining Placement of Keys on Servers
5. Adding a server to the Ring
6. Removing a server from the ring

4. Key things to remember

4.1 Where to use

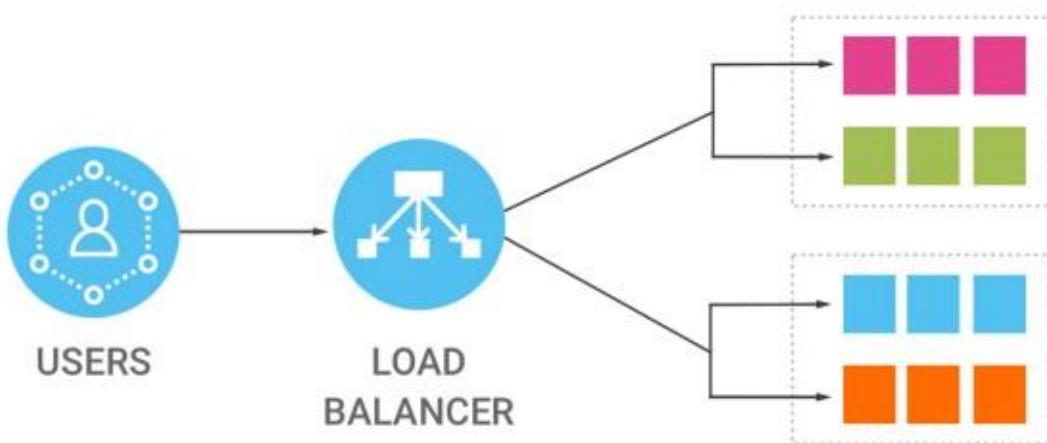
1. You have a cluster of databases and you need to elastically scale them up or down based on traffic load. For example, add more servers during Christmas to handle the extra traffic.
2. You have a set of cache servers that need to elastically scale up or down based on traffic load.

4.2 Benefits

1. Enables Elastic Scaling of a cluster of database/cache servers
2. Facilitates Replication and partitioning of data across servers
3. Partitioning of data enables uniform distribution which relieves hot spots
4. Points a-c enables higher availability of the system as a whole.

Load Balancing

Concepts and considerations for Load Balancing in System Design



1. Concepts

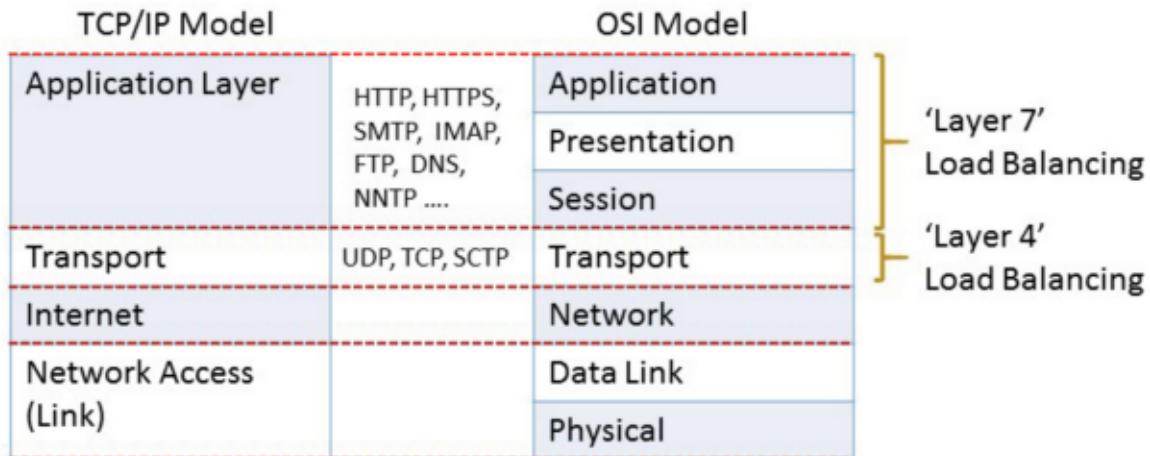
A load balancer is a device that acts as a reverse proxy and distributes network or application traffic across a number of servers. It helps scale **horizontally** across an ever-increasing number of servers.

2. How does the load balancer work?

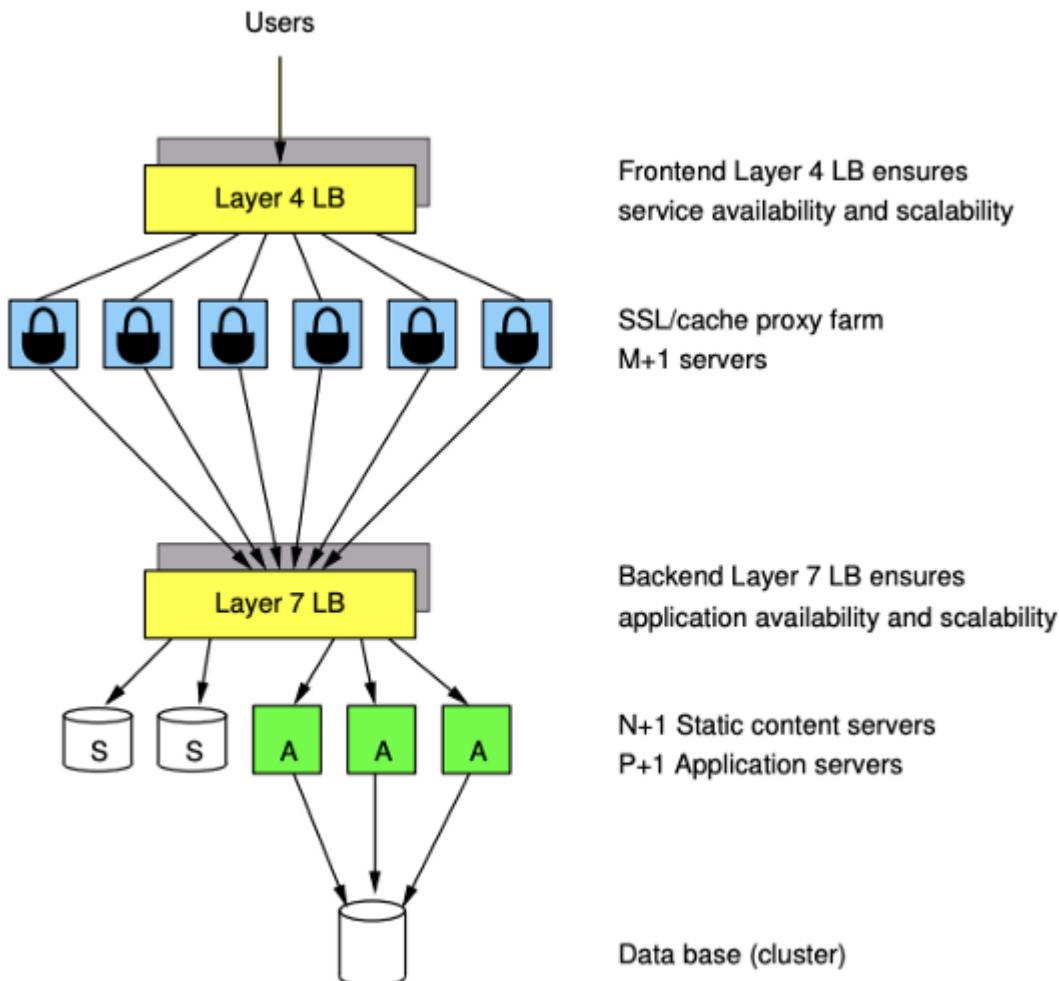
- **Define IP or DNS name for LB:** Administrators define one IP address and/or DNS name for a given application, task, or website, to which all requests will come. This IP address or DNS name is the load balancing server.
- **Add backend pool for LB:** The administrator will then enter into the load balancing server the IP addresses of all the actual servers that will be sharing the workload for a given application or task. This pool of available servers is only accessible internally, via the load balancer.
- **Deploy LB:** Finally, your load balancer needs to be deployed — either as a **proxy**, which sits between your app servers and your users worldwide and accepts all traffic, or as a **gateway**, which assigns a user to a server once and leaves the interaction alone thereafter.
- **Redirect requests:** Once the load balancing system is in place, all requests to the application come to the load balancer and are redirected according to the administrator's preferred algorithm.

3. Types of LBs

- Load balancers are generally grouped into two categories: Layer 4 and Layer 7.



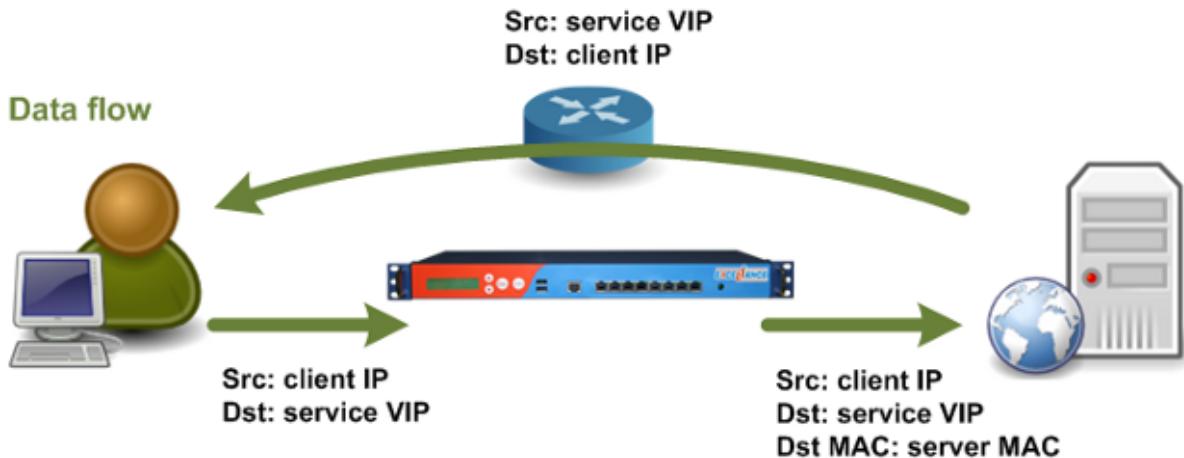
L4 vs L7



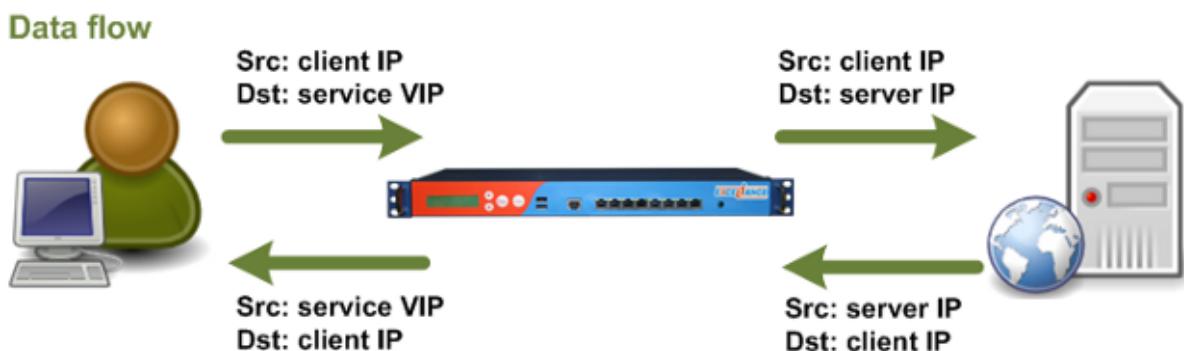
L4 vs L7

Layer 4 load balancers

- It acts upon data found in **network and transport layer** protocols (IP, TCP, FTP, UDP). They are mostly the **network address translators** (NATs) which share the load to the different servers getting translated to by these load balancers.
- Session persistence can be achieved at the IP address level
- No termination for TCP connections
- Two modes of L4 LB:
 1. Direct Server Return (DSR): The TCP connection is established directly between the client and the backend. The load-balancer sees only the requests and **just changes the destination MAC address** of the packets. The backends answer directly to the client using the **service IP (VIP)** configured on a loopback interface (this is why the src is VIP for response). **Note that the requests pass through the load balancer while the responses not.** LB won't be the bandwidth bottleneck.
 2. NAT Mode: The clients get connected to the **service VIP**. The load balancer chooses a server in the pool then forwards packets to it by **changing the destination IP address (DNAT)**, the LB becomes the default gateway for the real servers, and the source IP is the client's IP. **All traffic will pass through the load balancer**, and output bandwidth is limited by load balancer output capacity. There is only one connection established.



DSR Mode



NAT Mode

- Example: Azure Load Balancer, [Nginx as TCP and UDP load balancer](#)

Layer 7 load balancers

- It distributes requests based upon data found in **application layer** protocols such as HTTP. They can further distribute requests based on application-specific data such as HTTP headers, cookies, or data within the application message itself, such as the value of a specific parameter.
- For the client, the destination IP will be the IP of the load balancer, for the backend server, the source IP will be the IP of the load balancer.
- The cookie can be used to achieve a sticky session.
- IP of the client will be kept with the **X-Forwarded-For** header.
- To get the HTTP information, the connection is terminated at the load balancer, thus, there will be 2 TCP connections: Client-LB, LB-Backend.
- Example: Azure Application Gateway, [Nginx as HTTP load balancer](#)

4. LB Locations

- Between user and web servers (User => Web Servers)

- Between web servers and an internal platform layer (application servers, cache servers) (Webservers => App or Cache servers)
- Between internal platform layer and database (App or Cache servers => Database servers)

5. Algorithms

- Least connection
- Least response time
- Least bandwidth
- Round robin
- Weighted round-robin
- IP hash

6. Implementations

Reference: <https://www.thegeekstuff.com/2016/01/load-balancer-intro/>

Smart clients

- Developers lean towards smart clients because they are developers, and so they are used to writing software to solve their problems, and smart clients are software.
- The smart client is a client that takes a pool of service hosts and balances load across them, detects downed hosts, and avoids sending requests their way (they also have to detect recovered hosts, deal with adding new hosts, etc, making them fun to get working decently and a terror to setup).

Hardware load balancers

- Hardware load balancers are specialized hardware deployed in-between the server and the client. It can be a **switching/routing hardware or even a dedicated system** running a load balancing software with specialized capabilities. The hardware load balancers are implemented on Layer4 and Layer7 of the OSI model so prominent among this hardware are L4-L7 routers.
- Hardware Load Balancer Examples:
 1. [F5 BIG-IP load balancer](#) (Setup [HTTPS load balance on F5](#))
 2. CISCO system catalyst
 3. Barracuda load balancer
 4. Coytepoint load balancer
 5. Citrix **NetScaler**

Software load balancers

- Software load balancers generally implement a combination of one or more scheduling algorithms. They are often installed on the servers and consume the processor and memory of the servers.
- The following are a few examples of software load balancers:
 1. [HAProxy](#) — A TCP load balancer.

2. [NGINX](#) — An HTTP load balancer with SSL termination support.
3. [mod_athena](#) — Apache-based HTTP load balancer.
4. Varnish — A reverse proxy-based load balancer.
5. Balance — Open-source TCP load balancer.
6. LVS — Linux virtual server offering layer 4 load balancing.

7. Other Topics

[7.1 Load balancer vs Reverse proxy](#)

- They are components in a client-server computing architecture. Both act as intermediaries in the communication between the clients and servers, performing functions that improve efficiency. **Most load balancer programs are also reverse proxy servers**, which simplifies web application server architecture.
- **Load balancers** are most commonly deployed when a site needs ***multiple servers***. Some load balancers also provide ***session persistence***. It refers to direct a client's requests to the same backend web or application server for the duration of a "session" or the time it takes to complete a task or transaction (e.g. shopping).
- It often makes sense to deploy a **reverse proxy** even with just ***one web server or application server***.

7.2 Local vs Global Load Balancing

- Originally, load balancing referred to the distribution of traffic across servers in one locality — a single data center.
- As more and more computing is done online, load balancing has taken on a broader meaning. [Global Server Load Balancing](#) is the same in principle but the load is distributed planet-wide instead of just across a data center. Early generation GSLB solutions relied on [DNS load balancing](#), which limited both their performance and efficacy as it is now considered acceptable mostly for simple applications or websites.
- Another implement of GSLB is through a [Content Delivery Network \(CDN\)](#), such as the [Cloudflare CDN](#). A global CDN service will take data from their customers' origin servers and cache it on a geographically distributed network of servers, providing fast and reliable delivery of Internet content to users around the world.

7.3 Load Balancer and Certificates

- <https://serverfault.com/questions/68753/does-each-server-behind-a-load-balancer-need-their-own-ssl-certificate>
- If we do load balancing on the TCP or IP layer (OSI layer 4/3, a.k.a L4, L3), then all HTTP servers will need to have the SSL certificate installed.
- If we do load balance on the HTTPS layer (L7), then you'd commonly install the certificate on the load balancer alone, and use plain unencrypted HTTP over the local network between the load balancer and the webservers (for best performance on the web servers). The certificate installed on the load balancer can be used to decrypt the data (cookie, etc). This is called L7 TSL termination.
- [How to Setup F5 HTTPS SSL Load Balancing in Big-IP](#)

[7.4 L4 Load Balancer Configuration with Nginx \(TCP and UDP\)](#)

The Hash load-balancing method is also used to configure *session persistence*. As the hash function is based on the client's IP address, connections from a given client are always passed to the same server unless the server is down or otherwise unavailable. Specify an optional consistent parameter to apply the [ketama](#) consistent hashing method: **hash \$remote_addr consistent;**

```
stream {  
  
    upstream stream_backend {  
  
        hash $remote_addr;  
  
        server backend1.example.com:12345 weight=5;  
  
        server backend2.example.com:12345 max_fails=2 fail_timeout=30s;  
  
        server backend3.example.com:12345 max_conns=3;  
  
    }  
  
    upstream dns_servers {  
  
        least_conn;  
  
        server 192.168.136.130:53;  
  
        server 192.168.136.131:53;  
  
        server 192.168.136.132:53;  
  
    }  
  
    server {  
  
        listen      12345;  
  
        proxy_pass  stream_backend;  
  
        proxy_timeout 3s;  
  
        proxy_connect_timeout 1s;  
  
    }  
}
```

```

server {
    listen 53 udp;
    proxy_pass dns_servers;
}

```

[7.5 L7 Load Balancer Configuration with Nginx \(HTTP\)](#)

Load Balance the traffic based on requesting URI ([Nginx reverse proxy](#)). By default the round-robin algorithm will be used, we can also use least_conn, configure sticky cookie, etc.

```

http {
    server {
        listen 80;      location / {
            proxy_pass http://backend;
        }
        location /images {
            proxy_pass http://backend-static;
        }
    } upstream backend {
        server web-server1:80;
        server web-server2:80;
        sticky cookie srv_id expires=1h domain=.example.com path=/;
    } upstream backend-static {
        least_conn;
        server web-server3:80;
        server web-server4:80;
    }
}

```

We can also do TSL termination if we listen 443 and configure below.

```

server {

    listen      443 ssl http2 default_server;
    listen      [::]:443 ssl http2 default_server;
    server_name _;
    root       /usr/share/nginx/html;

    ssl_certificate "/etc/pki/nginx/server.crt";
    ssl_certificate_key "/etc/pki/nginx/private/server.key";
    ssl_session_cache shared:SSL:1m;
    ssl_session_timeout 10m;
    ssl_ciphers HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers on;

    # Load configuration files for the default server block.

    include /etc/nginx/default.d/*.conf;

}

location / {
}

```

[7.6 Setup an HA load balancer with Nginx](#)

In 7.4 and 7.5, we demonstrated how to set up L4 and L7 load balancers with Nginx, but how can we make sure the load balancer itself is highly available? The basic idea for an active-passive high-availability (HA) setup is as below:

1. Setup an Nginx cluster with 2+ nodes with [keepalived](#) daemon.
2. An implementation of the [Virtual Router Redundancy Protocol](#) (VRRP) to manage virtual routers (virtual IP addresses, or VIPs). VRRP ensures that there is a master node at all times. The backup node listens for VRRP advertisement packets from the master node. If it does not receive an advertisement packet for a period longer than three times the configured advertisement interval, the backup node takes over as master and assigns the configured VIPs to itself.

3. A health-check facility to determine whether a service (for example, a web server, PHP backend, or database server) is up and operational.

If a service on a node fails the configured number of health checks, keepalived reassigned the virtual IP address from the master (active) node to the backup (passive) node. On each node, the configuration file will contain the following information:

- The IP address of the local and remote nodes (one of which will be configured as a master, the other as a backup)
- One free IP address to be used as the cluster endpoint's (floating) VIP

To see which node is currently the master for a given VIP, run the ip addr show command for the interface on which the VRRP instance is defined (in the following commands, interface eth0 on nodes centos7-1 and centos7-2):

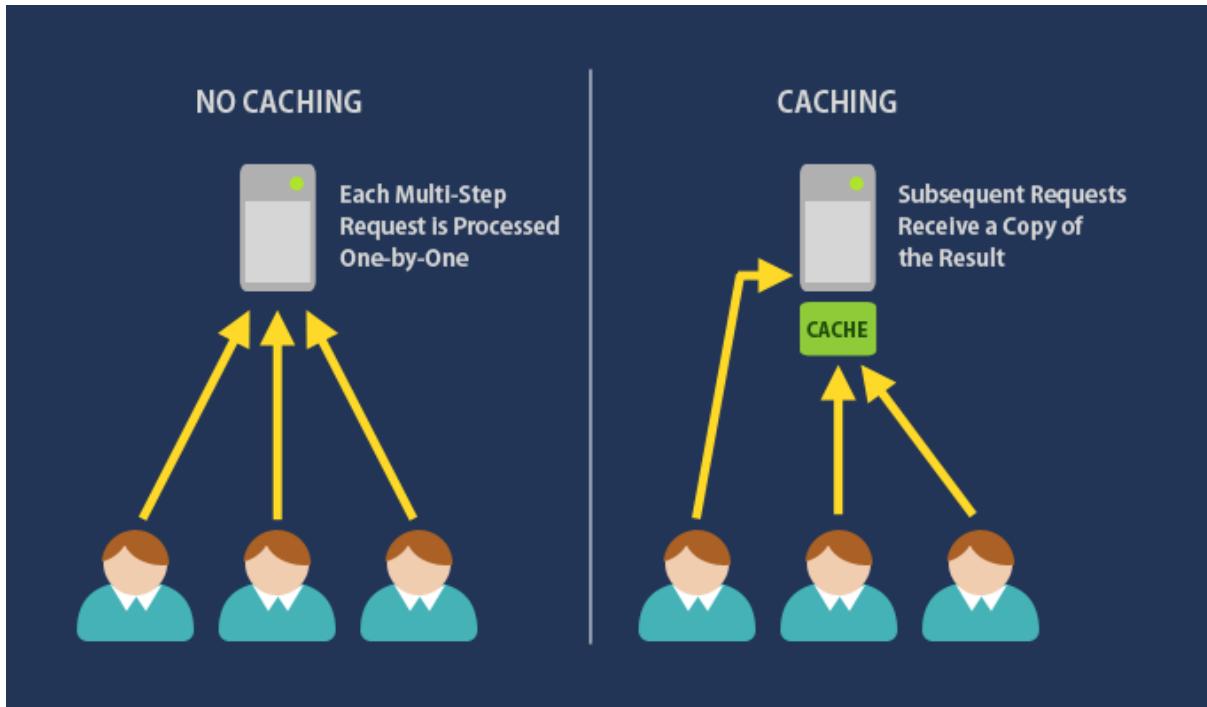
```
centos7-1 # ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
          UP qlen 1000
          link/ether 52:54:00:33:a5:a5 brd ff:ff:ff:ff:ff:ff
          inet 192.168.100.100/24 brd 192.168.122.255 scope global dynamic eth0
            valid_lft 3071sec preferred_lft 3071sec
          inet 192.168.100.150/32 scope global eth0
            valid_lft forever preferred_lft forever
```

```
centos7-2 # ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
          UP qlen 1000
          link/ether 52:54:00:33:a5:87 brd ff:ff:ff:ff:ff:ff
          inet 192.168.100.101/24 brd 192.168.122.255 scope global eth0
            valid_lft forever preferred_lft forever
```

In this output, the second inet line for centos7-1 indicates that it is master – the defined VIP (192.168.100.150) is assigned to it. The other inet lines in the output show the master node's real IP address (192.168.100.100) and the backup node's IP address (192.168.100.101).

Caching

Concepts and considerations for Caching in System Design



1. Concepts

- Take advantage of the locality of reference principle: recently requested data is likely to be requested again.
- Exist at all levels in architecture, but often found at the level nearest to the front end.
- A cache is like short-term memory which has a limited amount of space. It is typically faster than the original data source.
- Caching consists of
 1. **precalculating results** (e.g. the number of visits from each referring domain for the previous day)
 2. **pre-generating expensive indexes** (e.g. suggested stories based on a user's click history)
 3. **storing copies** of frequently accessed data in a faster backend (e.g. Memcache instead of PostgreSQL).

2. Caches in different layers

2.1 Client-side

- Use case: Accelerate retrieval of web content from websites (browser or device)
- Tech: HTTP Cache Headers, Browsers
- Solutions: Browser Specific

2.2 DNS

- Use case: Domain to IP Resolution
- Tech: DNS Servers
- Solutions: Amazon Route 53

2.3 Web Server

- Use case: Accelerate retrieval of web content from web/app servers. Manage Web Sessions (server-side)
- Tech: HTTP Cache Headers, CDNs, Reverse Proxies, Web Accelerators, Key/Value Stores
- Solutions: Amazon CloudFront, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions

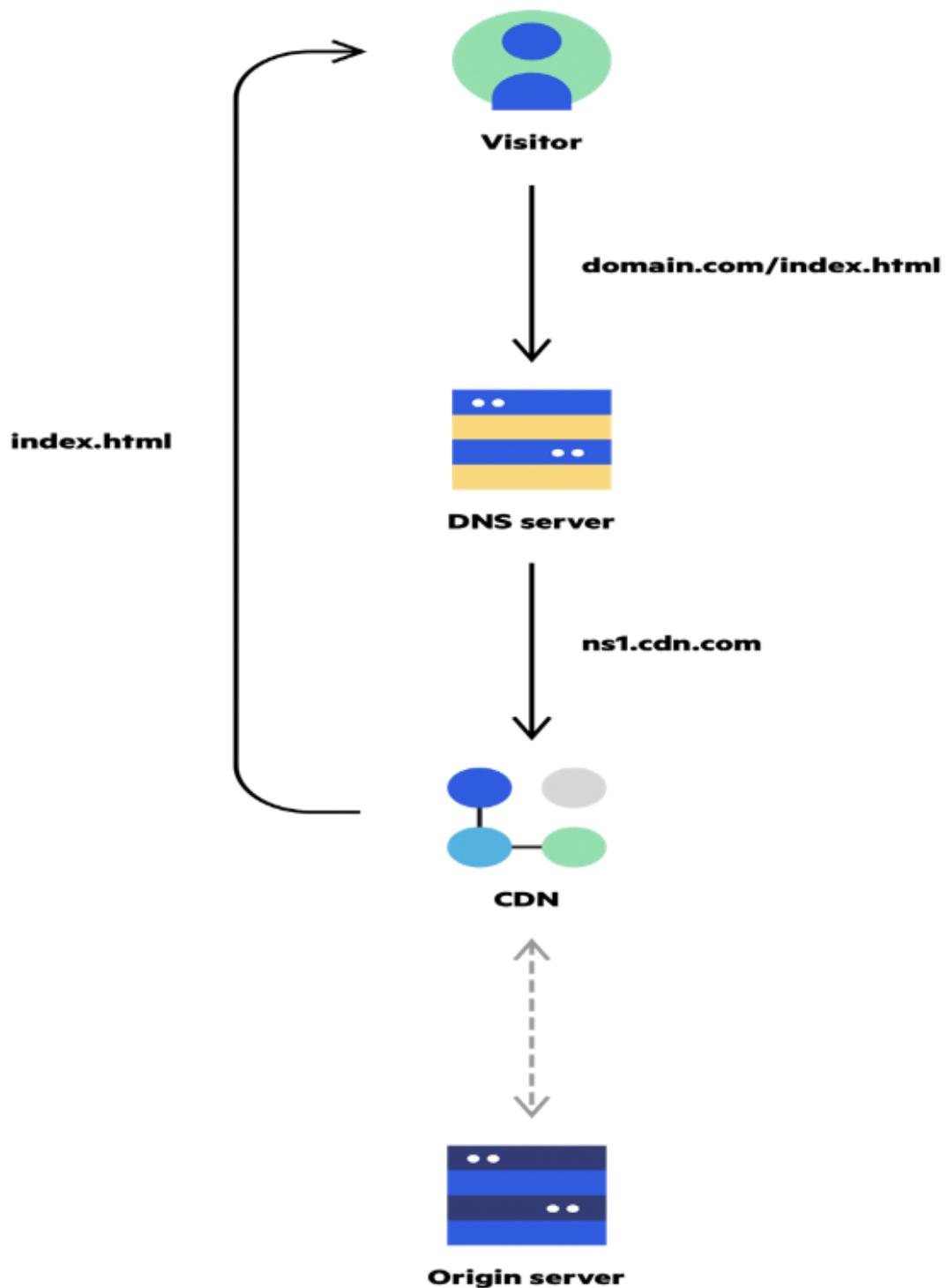
2.4 Application

- Use case: Accelerate application performance and data access
- Tech: Key/Value data stores, Local caches
- Solutions: Redis, Memcached
- Note: Basically it **keeps a cache directly on the Application server**. Each time a request is made to the service, the node will quickly return local, cached data if it exists. If not, the requesting node will query the data by going to network storage such as a database. When the application server is expanded to many nodes, we may face the following issues:
 1. The load balancer randomly distributes requests across the nodes.
 2. The same request can go to different nodes, increase cache misses.
 3. Extra storage since the same data will be stored in two or more different nodes.
 Solutions for the issues:
 1. Global caches
 2. Distributed caches

2.5 Database

- Use case: Reduce latency associated with database query requests
- Tech: Database buffers, Key/Value data stores
- Solutions: The 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, can also use Redis, Memcached

2.6 Content Distribution Network (CDN)



CDN

- Use case: Take the burden of serving static media off of your application servers and provide a geographic distribution.
- Solutions: [Amazon CloudFront](#), Akamai
- Note: **If the system is not large enough for CDN, it can be built like this:**
 1. Serving static media off a separate subdomain using a lightweight HTTP server

(e.g. Nginx, Apache).

2. Cutover the DNS from this subdomain to a CDN layer.

2.7 Other Cache

- CPU Cache: Small memories on or close to the [CPU](#) can operate faster than the much larger [main memory](#). Most CPUs since the 1980s have used one or more caches, sometimes [in cascaded levels](#); modern high-end [embedded](#), [desktop](#), and server [microprocessors](#) may have as many as six types of cache (between levels and functions)
- GPU Cache
- Disk Cache: While CPU caches are generally managed entirely by hardware, a variety of software manages other caches. The [page cache](#) in main memory, which is an example of disk cache, is managed by the operating system kernel

3. Cache Invalidation

If the data is modified in the database, it should be invalidated in the cache, if not, this can cause inconsistent application behavior. There are majorly three kinds of caching systems: **Write-through cache**, **Write-around cache**, **Write-back cache**.

1. **Write through cache:** Where writes go through the cache and write is confirmed as success only if writes to DB and the cache BOTH succeed. **Pro:** Fast retrieval, complete data consistency, robust to system disruptions.
Con: Higher latency for write operations.
2. **Write around cache:** Where write directly goes to the DB, bypassing the cache.
Pro: This may reduce latency.
Con: However, it increases cache misses because the cache system reads the information from DB in case of a cache miss. As a result, this can lead to higher read latency in the case of applications that write and re-read the information quickly. Read must happen from slower back-end storage and experience higher latency.
3. **Write back cache:** Where the write is directly done to the caching layer and the write is confirmed as soon as the write to the cache completes. The cache then asynchronously syncs this write to the DB.
Pro: This would lead to a really quick write latency and high write throughput for the write-intensive applications.
Con: However, there is a risk of losing the data in case the caching layer dies because the only single copy of the written data is in the cache. We can improve this by having more than one replica acknowledging the write in the cache.

4. Cache eviction policies

Following are some of the most common cache eviction policies:

1. **First In First Out (FIFO):** The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.

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

5. Other

5.1 Global caches

All the nodes use the same single cache space (a server or file store). Each of the application nodes queries the cache in the same way it would a local one. However, it is very easy to overwhelm a single global cache system as the number of clients and requests increase but is very effective in some architectures.

Two forms of handling cache miss:

- Cache server handles cache miss, which is used by most applications.
- Request nodes handle cache miss:
 1. Have a large percentage of the hot data set in the cache.
 2. An architecture where the files stored in the cache are static and shouldn't be evicted.
 3. The application logic understands the eviction strategy or hot spots better than the cache.

5.2 Distributed caches

The cache is divided up using a consistent hashing function and each of its nodes owns part of the cached data. If a requesting node is looking for a certain piece of data, it can quickly use the hashing function to locate the information within the distributed cache to determine if the data is available.

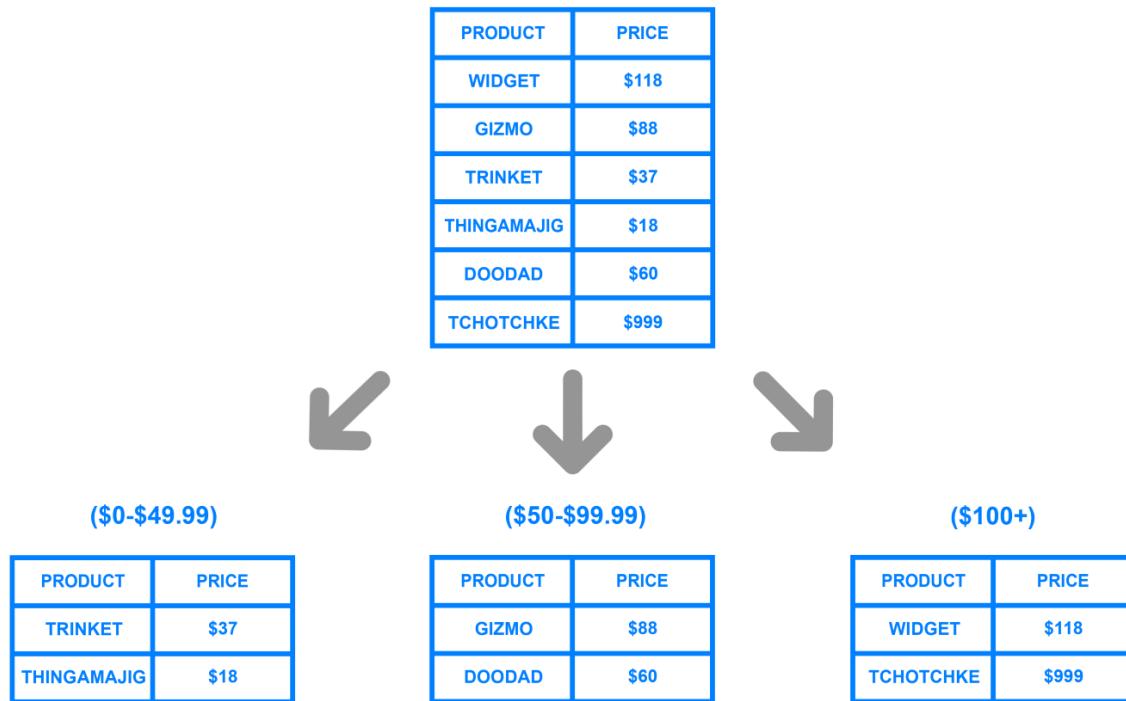
- Pros: Cache space can be increased easily by adding more nodes to the request pool.
- Cons: A missing node can lead to cache loss. We may get around this issue by storing multiple copies of the data on different nodes.

5.3 Design a Cache System

- A single machine is going to handle 1M QPS
- Map and LinkedList should be used as the data structures. We may get better performance on the double-pointer linked-list on the remove operation.
- Master-slave technique

Sharding / Data Partitioning

Concepts and considerations for Sharding and Data Partitioning in System Design



1. Partitioning method

1. 1 Horizontal partitioning — also known as sharding

It is a range-based sharding. You put different rows into different tables, the structure of the original table stays the same in the new tables, i.e., we have the same number of columns.

- When partitioning your data, you need to assess the number of rows in the new tables, so each table has the same number of data and will grow by a similar number of new customers in the future.
- Con: If the value whose range is used for sharding isn't chosen carefully, the partitioning scheme will lead to unbalanced servers.

1. 2 Vertical partitioning

This type of partition divides the table vertically (by columns), which means that the structure of the main table changes in the new ones.

- An ideal scenario for this type of partition is when you don't need all the information about the customer in your query.
- **Divide data for a specific feature to their own server.** When you have different types of data in your database, such as names, dates, and pictures. You could keep the string values in SQL DB (expensive), and pictures in an Azure Blob (cheap).
- Pro: Straightforward to implement. Low impact on the application.

- Con: To support the growth of the application, a database may need further partitioning.

1.3 Directory-based partitioning

- A lookup service that knows the partitioning scheme and abstracts it away from the database access code.
- Allow the addition of DB servers or change of partitioning schema without impacting the application.
- Con: Can be a single point of failure.

2. Partitioning criteria

Link: <https://dev.mysql.com/doc/mysql-partitioning-excerpt/5.7/en/partitioning-types.html>

2.1 Range partitioning

- This type of partitioning assigns rows to partitions based on column values falling within a given range. e.g, store_id is a column of the table.

```
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN (21)
);
```

2.2 Key or hash-based partitioning

- Apply a hash function to some key attribute of the entry to get the partition number. e.g. partition based on the year in which an employee was hired.

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
```

```

    store_id INT
)
PARTITION BY HASH( YEAR(hired) )
PARTITIONS 4;

```

- Problem
 1. Adding new servers may require changing the hash function, which would need a redistribution of data and downtime for the service.
 2. Workaround: [consistent hashing](#).

2.3 List partitioning

- Similar to partitioning by RANGE, except that the partition is selected based on columns matching one of a set of discrete values. Each partition is assigned a list of values. e.g.

```

PARTITION BY LIST(store_id) (
    PARTITION pNorth VALUES IN (3,5,6,9,17),
    PARTITION pEast VALUES IN (1,2,10,11,19,20),
    PARTITION pWest VALUES IN (4,12,13,14,18),
    PARTITION pCentral VALUES IN (7,8,15,16)
);

```

2.4 Round-robin partitioning

- With n partitions, the i tuple is assigned to partition $i \% n$.

2.5 Composite partitioning

- Combine any of the above partitioning schemes to devise a new scheme.
- Consistent hashing is a composite of hash and list partitioning.
- Key -> reduced key space through hash -> list -> partition.

3. Common problems of sharding

Most of the constraints are due to the fact that operations across multiple tables or multiple rows in the same table will no longer run on the same server.

3.1 Joins and denormalization

- Joins will not be performance efficient since data has to be compiled from multiple servers.

- Workaround: **denormalize the database so that queries can be performed from a single table**. But this can lead to data inconsistency.

3.2 Referential integrity

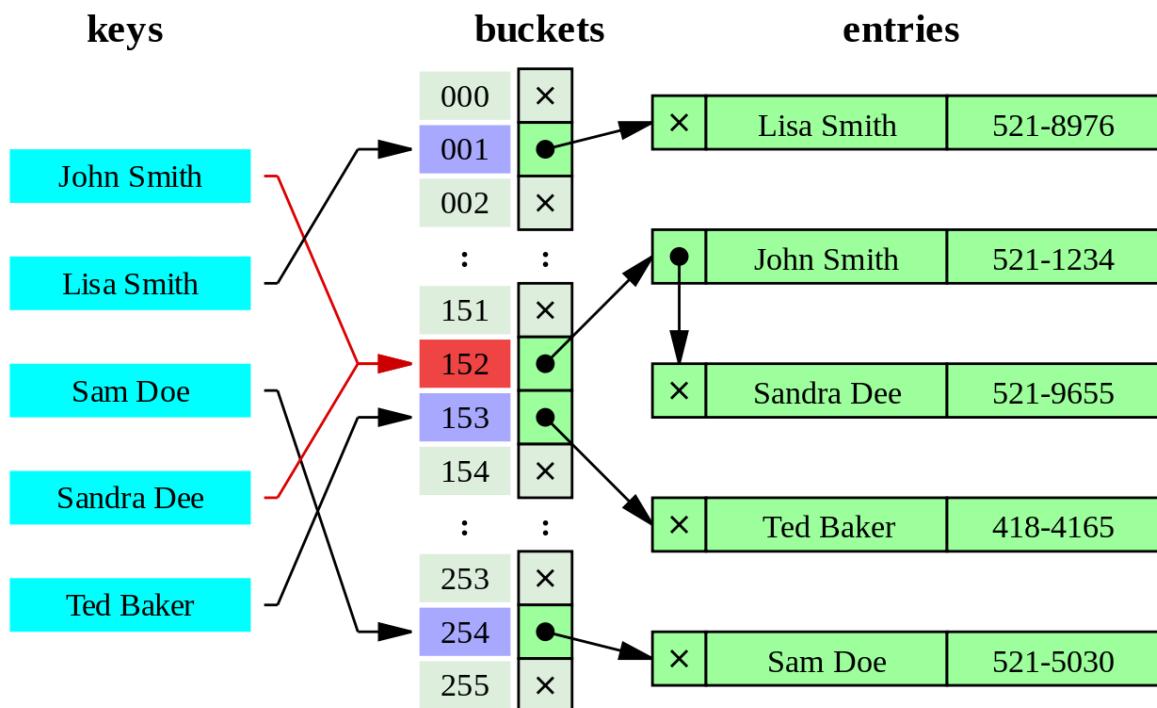
- Difficult to enforce data integrity constraints (e.g. foreign keys).
- Workaround
 1. Referential integrity is enforced by the application code.
 2. Applications can run SQL jobs to clean up dangling references.

3.3 Rebalancing

- Necessity of rebalancing
 1. Data distribution is not uniform.
 2. A lot of load on one shard.
- Creating more DB shards or rebalancing existing shards changes the partitioning scheme and requires data movement.

Indexes

Concepts and considerations for Indexes in System Design



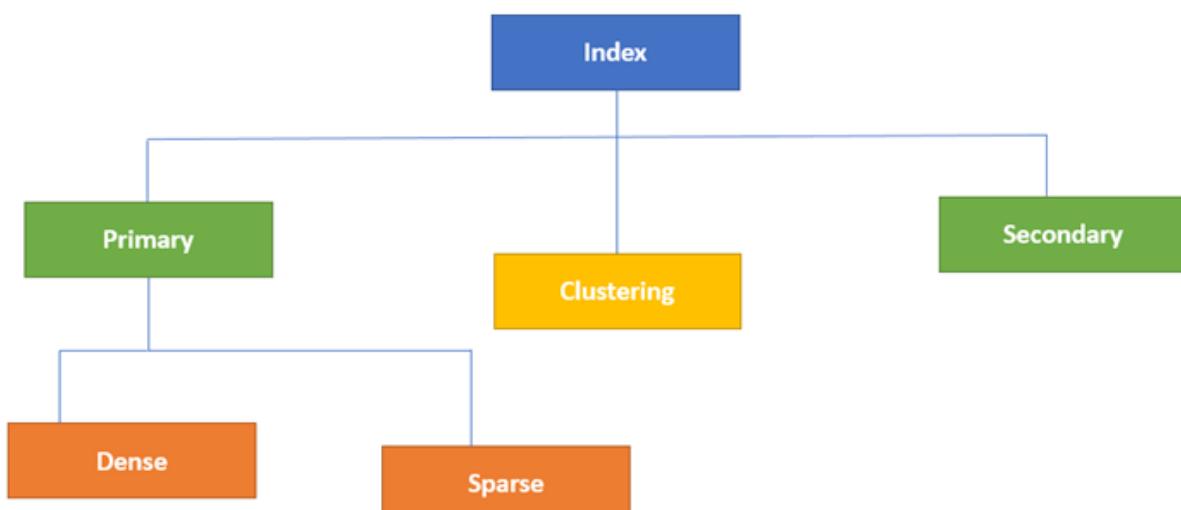
1. Concepts

- **Indexing** is a data structure technique that allows you to quickly retrieve records from a database file. An Index is a small table having only two columns. The first column comprises a copy of the primary or candidate key of a table. Its second column contains a set of pointers for holding the address of the disk block where that specific key-value stored.
- Improve the performance of search queries.
- Decrease the write performance. This performance degradation applies to all insert, update, and delete operations.
- The keys are based on the tables' columns. By comparing keys to the index it is possible to find one or more database records with the same value.
- The structure that is used to store a database index is called a B+ Tree.
- Since data is constantly updated in a database, it's important for the B+ Tree to keep its balance. Each time records are added, removed, or keys updated, special algorithms shift data and key values from block to block to ensure no one part of the tree is more than one level higher than the other.
- Indexes are created using some database columns. The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly (Note that the data may or may not be stored in sorted order).
- The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.

Search Key	Data Reference
------------	----------------

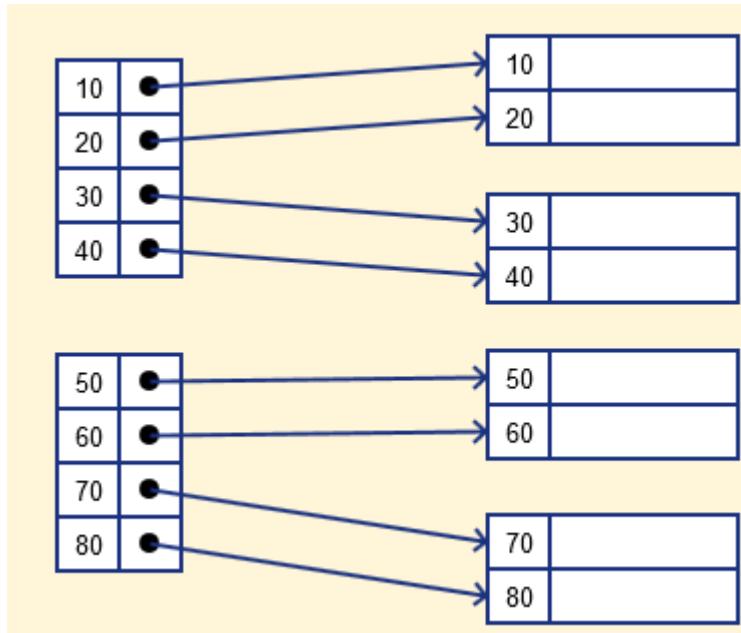
Structure of an index

2. Types of indexing methods



2.1 Primary Index

Primary Index is an ordered file which is fixed length size with two fields. The first field is the same as a primary key and the second field is pointed to that specific data block. In the primary Index, there is always one to one relationship between the entries in the index table.



Advantages of Primary Index:

- Primary index-based range queries are very efficient. There might be a possibility that the disk block that the database has read from the disk contains all the data belonging to the query since the primary index is clustered & records are ordered physically. So the locality of data can be provided by the primary index.
- Any query that takes advantage of the primary key is very fast.

Disadvantages of Primary Index:

- Since the primary index contains a direct reference to the data block address through the virtual address space & disk blocks are physically organized in the order of the index key, every time the OS does some disk page split due to DML operations like INSERT / UPDATE / DELETE, the primary index also needs to be updated. So DML operations put some pressure on the performance of the primary index.

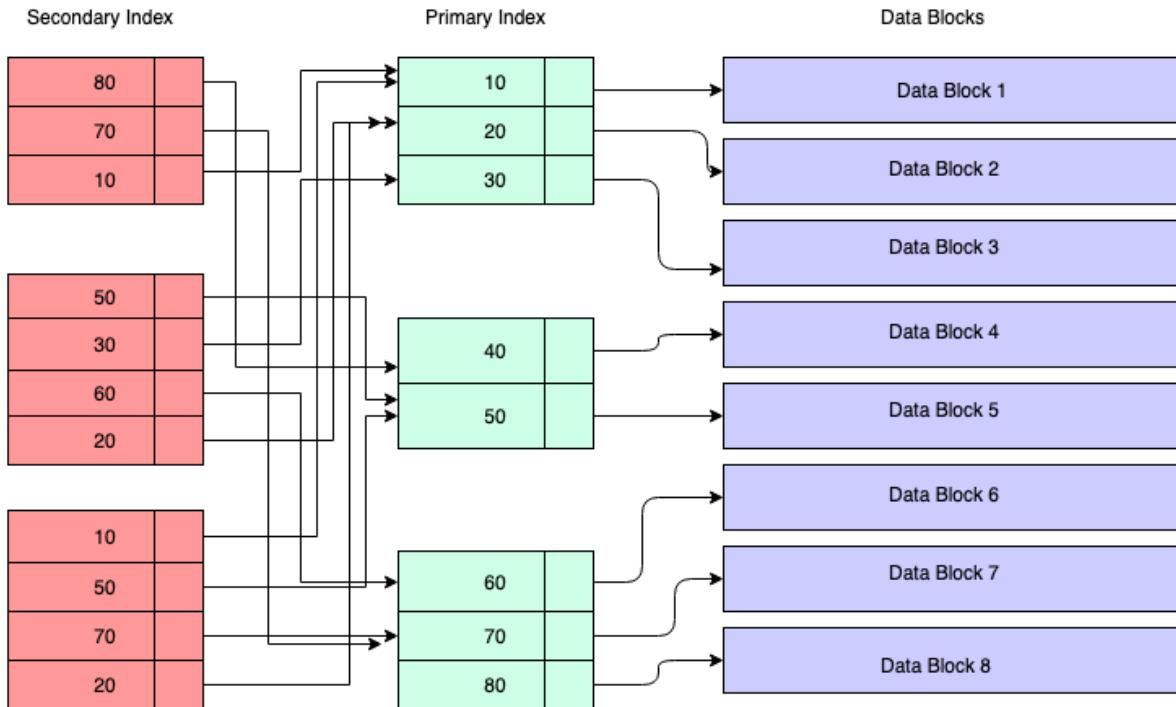
The primary Indexing is also further divided into two types:

- Dense Index
- Sparse Index

2.2 Secondary Index

The secondary Index can be generated by a field that has a unique value for each record, and it should be a candidate key. It is also known as a non-clustering index.

This two-level database indexing technique is used to reduce the mapping size of the first level. For the first level, a large range of numbers is selected because of this; the mapping size always remains small.



Advantages of a Secondary Index:

Logically you can create as many secondary indices as you want. But in reality how many indices actually required needs a serious thought process since each index has its own penalty.

Disadvantages of a Secondary Index:

With DML operations like DELETE / INSERT , the secondary index also needs to be updated so that the copy of the primary key column can be deleted/inserted. In such cases, the existence of lots of secondary indexes can create issues.

2.3 Clustering Index

Records themselves are stored in the Index and not pointers. Sometimes the Index is created on non-primary key columns which might not be unique for each record. In such a situation, you can group two or more columns to get the unique values and create an index which is called a clustered index. This also helps you to identify the record faster.

Advantage of Clustered Index:

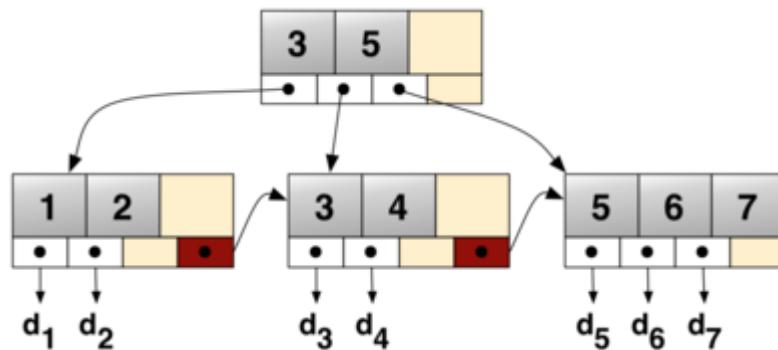
This ordering or co-location of related data actually makes a clustered index faster. When data is fetched from the disk, the complete block containing the data is read by the system since our disk IO system writes & reads data in blocks. So in the case of range queries, it's quite possible that the collocated data is buffered in memory.

Constraints of Clustered Index:

Since a clustered index impacts the physical organization of the data, there can be only one clustered index per table.

3. B-Tree Index

B-tree index is the widely used data structures for Indexing. It is a multilevel index format technique that is balanced binary search trees. All leaf nodes of the B tree signify actual data pointers. While the advantages of the B-Tree are numerous, the main advantage for our purposes is that it is sortable. When the data structure is sorted in order, it makes our search more efficient for the obvious reasons we pointed out above. When the index creates a data structure on a specific column, it is important to note that no other column is stored in the data structure.



A simple B+ tree example linking the keys 1–7 to data values d₁–d₇. The linked list (red) allows rapid in-order traversal. This particular tree's branching factor is b=4

Some possible candidates are — BTREE, HASH, RTREE or FULLTEXT.

4. Advantages of Indexing

Important pros/ advantage of Indexing are:

- It helps you to reduce the total number of I/O operations needed to retrieve that data, so you don't need to access a row in the database from an index structure.
- Offers Faster search and retrieval of data to users.
- Indexing also helps you to reduce tablespace as you don't need to link to a row in a table, as there is no need to store the ROWID in the Index. Thus you will be able to reduce the tablespace.
- You can't sort data in the lead nodes as the value of the primary key classifies it.

5. Disadvantages of Indexing

Important drawbacks/cons of Indexing are:

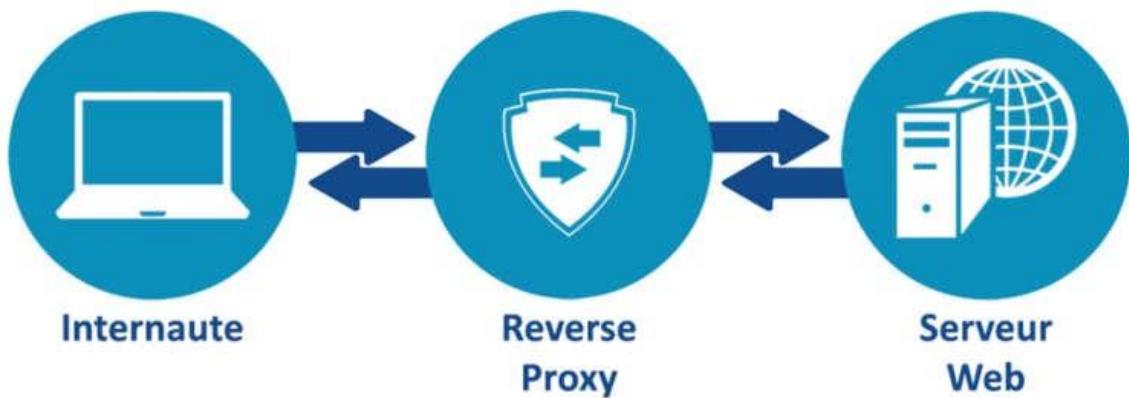
- To perform the indexing database management system, you need a primary key on the table with a unique value.
- You can't perform any other indexes on the Indexed data.
- You are not allowed to partition an index-organized table.
- SQL Indexing Decrease performance in INSERT, DELETE, and UPDATE query.

6. What Clauses use index?

It is a common misconception that indexes only help the where clause. B-tree indexes can also help the order by, group by, select and other clauses. It is just the B-tree part of an index—not the doubly linked list—that cannot be used by other clauses.

Proxies

Concepts and considerations for Proxies in System Design



1. Concepts

A proxy server is an intermediary piece of hardware/software sitting between the client and the backend server.

- **Filter requests:** It runs every request through a filter, looking up each address in its database of allowed or disallowed sites, and it allows or blocks each request based on its internal database. A system administrator can configure the proxy server to allow or block certain sites.
- Log requests
- Transform requests (encryption, compression, etc)

- **Cache:** A caching proxy server can also improve web performance by caching frequently used pages so the user request doesn't have to go all the way out to the Internet at large to get some of the data it needs to display a particular page.
- Batch requests
- Collapsed forwarding: enable multiple client requests for the same URI to be processed as one request to the backend server
- Collapse requests for data that is spatially close together in the storage to minimize the reads
- **Security:** A proxy server can also be used to beef up security for a business. A proxy server can provide network address translation, which makes the individual users and computers on the network anonymous when they are using the Internet. This makes it much harder for hackers to access individual computers on the network.

2. Proxy Server Types

Proxies can reside on the client's local server or anywhere between the client and the remote servers.

2.1 Open Proxy

An [open proxy](#) is a proxy server that is accessible by any Internet user. Generally, a proxy server only allows users within a networking group (i.e. a closed proxy) to store and forward Internet services such as DNS or web pages to reduce and control the bandwidth used by the group. With an open proxy, however, any user on the Internet is able to use this forwarding service.

2.2. Reverse Proxy

A [reverse proxy](#) retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client, appearing as if they originated from the proxy server itself. It typically sits behind the firewall in a private network and directs client requests to the appropriate backend server. A reverse proxy provides an additional level of abstraction and control to ensure the smooth flow of network traffic between clients and servers.

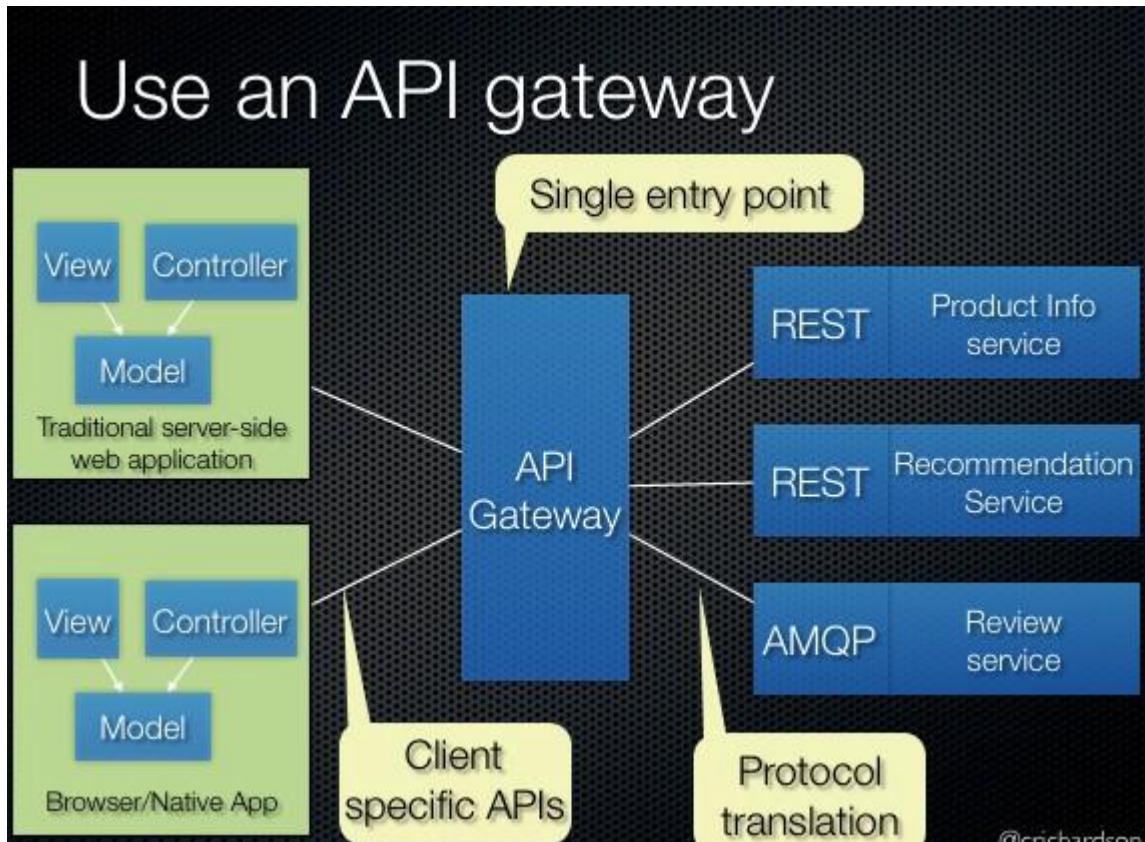
Common uses for a [reverse proxy server](#) include:

- **Load balancing** — A reverse proxy server can act as a “traffic cop,” sitting in front of your backend servers and distributing client requests across a group of servers in a manner that maximizes speed and capacity utilization while ensuring no one server is overloaded, which can degrade performance. If a server goes down, the [load balancer](#) redirects traffic to the remaining online servers.
- **Web acceleration** — Reverse proxies can compress inbound and outbound data, as well as cache commonly requested content, both of which speed up the flow of traffic between clients and servers. They can also perform additional tasks such as SSL encryption to take the load off of your web servers, thereby [boosting their performance](#).

- **Security and anonymity** — By intercepting requests headed for your backend servers, a reverse proxy server protects their identities and acts as an additional defense against security attacks. It also ensures that multiple servers can be accessed from a single record locator or URL regardless of the structure of your local area network.

3. API Gateway

3.1 Concepts



An API gateway is an [API management](#) tool that sits between a client and a collection of backend services. An API gateway is a layer 7 (HTTP) router that acts as a reverse proxy to accept all API calls, aggregate the various services required to fulfill them, and return the appropriate result.

With an API gateway, one simply exposes and scales a single collection of services (the API gateway) and updates the API gateway's configuration whenever a new upstream should be exposed externally. e.g. [Zuul](#) is an L7 application gateway that is able to auto-discover services registered in the [Eureka](#) server.

Exactly what the API gateway does will vary from one implementation to another. Some common functions include **authentication**, **routing**, **rate limiting**, **billing**, **monitoring**, **analytics**, **policies**, **alerts**, and **security**.

3.2 Main use cases

- Authentication: An API Gateway can take the overhead of authenticating an API call from outside, which can remove the check of security and lowering the network latency.
- Load Balancing: The API Gateway can work as an L7 load balancer to handle requests in the most efficient manner. It can keep a track of the request load it has sent to different nodes of a particular service.
- Service discovery and requests dispatching: it can make the communication between client and Microservices simpler. It hits all the required services and waits for the results from all services. After obtaining the response from all the services, it combines the result and sends it back to the client. An API Gateway can record the basic response time from each node of a service instance. For higher priority API calls, it can be routed to the fastest responding node.
- Response transformation: Being a first and single point of entry for all API calls, the API Gateway knows which type of client is calling: mobile, web client, or other external consumers; it can make the internal call to the client and give the data to different clients as per their needs and configuration.
- Circuit breaker: To handle a partial failure, the API Gateway uses a technique called circuit breaker pattern, which means that after a specific threshold, the API gateway will stop sending data to the component failing. This gives time to analyze the logs, implement a fix, and push an update. Or if necessary close the circuit until the issue is solved.

3.3 Pros and Cons

Benefits

- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips. For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also mean less overhead and improve user experience. An API gateway is essential for mobile applications.
- Simplifies the client by moving logic for calling multiple services from the client to the API gateway
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

Drawbacks

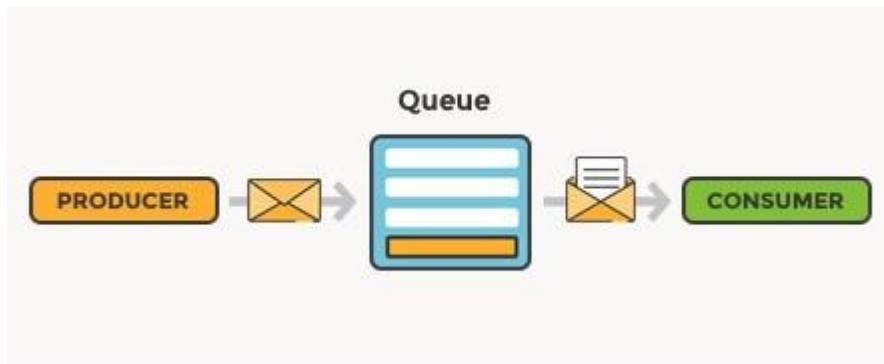
- Increased complexity — the API gateway is yet another moving part that must be developed, deployed, and managed
- Increased response time due to the additional network hop through the API gateway — however, for most applications the cost of an extra roundtrip is insignificant.

3.4 Implementation

- [Microsoft API Management](#): is a feature-rich service that can act as a gateway for Microservices.
- [NGINX Plus](#): A software load balancer with features that are provided at the API Gateway like security, web server, and content caching.
- [Amazon API Gateway](#): An AWS service for creating, publishing, maintaining, monitoring, and securing APIs at any scale.

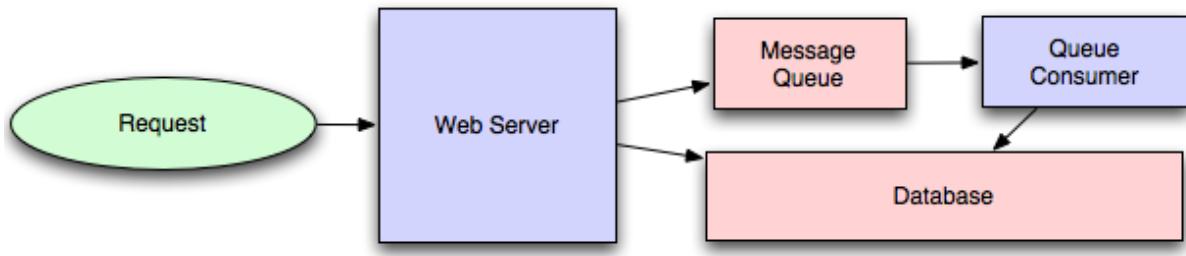
Message Queues

Concepts and considerations for Message Queues in System Design



1. Concepts

- Message queuing makes it possible for applications to communicate asynchronously, by sending messages to each other via a queue. A message queue provides temporary storage between the sender and the receiver so that the sender can keep operating without interruption when the destination program is busy or not connected. **Asynchronous processing** allows a task to call a service, and *move on to the next task* while the service processes the request at its own pace.
- A **queue** is a line of things waiting to be handled — in sequential order starting at the beginning of the line. A **message queue** is a queue of messages sent between applications. It includes a sequence of work objects that are waiting to be processed.
- A **message** is the data transported between the sender and the receiver application; it's essentially a byte array with some headers on top. An example of a message could be an event. One application tells another application to start processing a specific task via the queue.
- The basic architecture of a **message queue** is simple: there are client applications called **producers** that create messages and deliver them to the message queue. Another application, called a **consumer**, connects to the queue and gets the messages to be processed. Messages placed onto the queue are stored until the consumer retrieves them.
- The queue can provide protection from service outages and failures.
- Examples of queues: Kafka, Heron, real-time streaming, Amazon SQS, and RabbitMQ.



2. Design user interface when MQ is involved

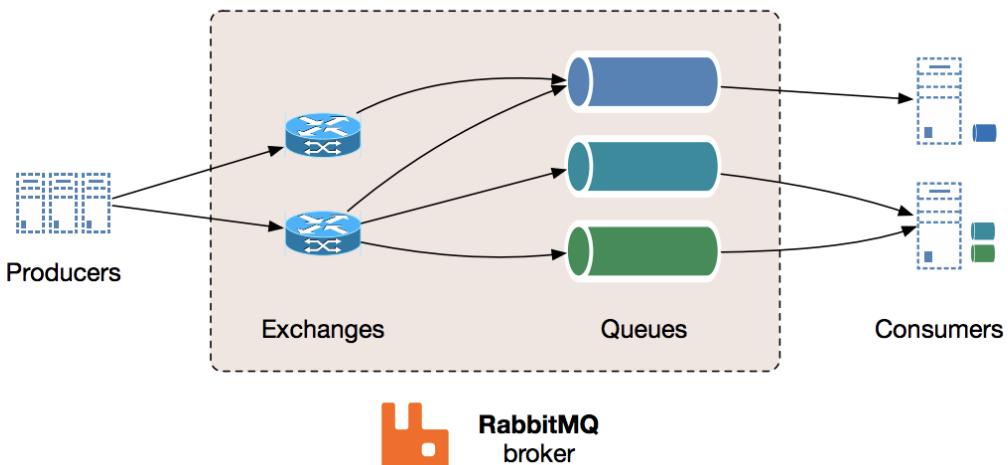
Dividing work between off-line work handled by a consumer and in-line work done by the web application depends entirely on the interface you are exposing to your users. Generally, you'll either:

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 afterward (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 the band; it's simple isn't feasible to update all the followers for a [Scobleizer](#) in real-time).

3. The role of message queuing in a microservice architecture

- In a microservice architecture, there are different functionalities divided across different services, that offer various functionalities. These services are coupled together to form a complete software application.
- Typically, in a microservice architecture, there are cross-dependencies, which entail that no single service can perform its functionalities without getting help from other services. **This is where it's crucial for your system to have a mechanism in place which allows services to keep in touch with each other without getting blocked by responses.**
- Message queuing fulfills this purpose by providing a means for services to push messages to a queue asynchronously and ensure that they get delivered to the correct destination. To implement a message queue between services, you need a **message broker, think of it as a mailman**, who takes mail from a sender and delivers it to the correct destination.

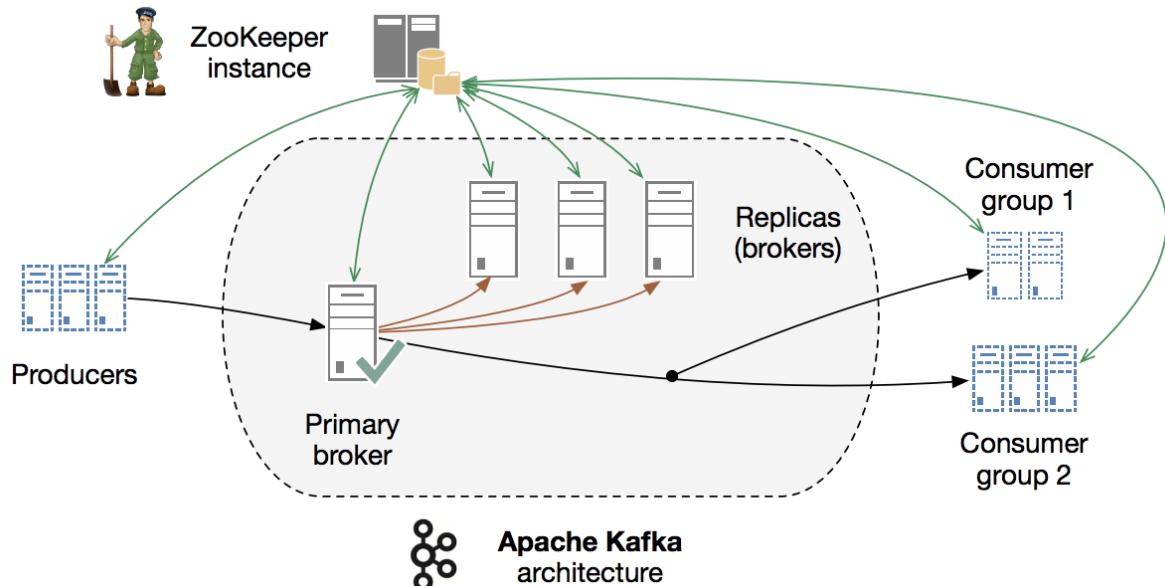
4. Message Broker — RabbitMQ



- RabbitMQ is one of the most widely used message brokers, it acts as the message broker, “the mailman”, a microservice architecture needs.
- RabbitMQ consists of:
 1. producer — the client that creates a message
 2. consumer — receives a message
 3. queue — stores messages
 3. exchange — enables to route messages and send them to queues
- The system functions in the following way:
 1. producer creates a message and sends it to an exchange
 2. exchange receives a message and routes it to queues subscribed to it
 3. consumer receives messages from those queues he/she is subscribed to

One should note that messages are filtered and routed depending on the type of exchange.
- [Use RabbitMQ via CloudAMQP](#)

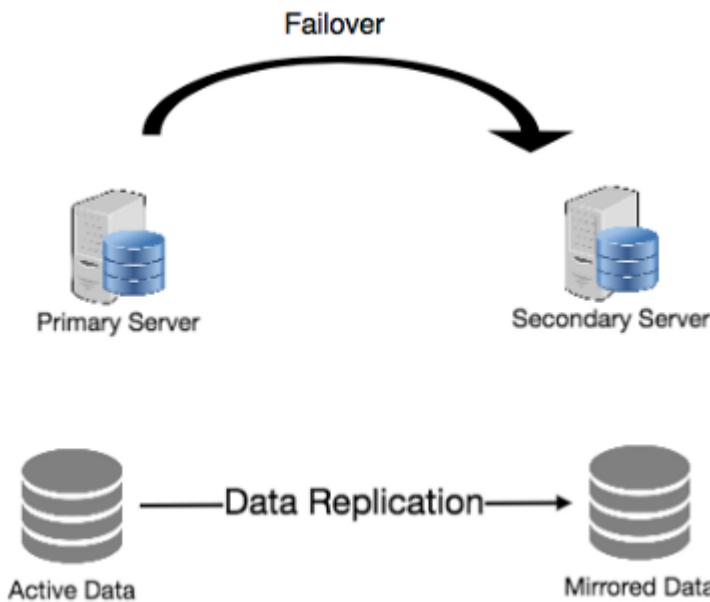
5. Apache Kafka



- [When to use rabbitmq or apache kafka](#)

Redundancy and Replication

Concepts and considerations for Redundancy and Replication in System Design



1. Concepts

- Duplication of critical data or services with the intention of increased reliability and availability of the system.

2. Server failover

- Remove single points of failure and provide backups (e.g. server failover).

3. Shared-nothing architecture

- Each node can operate independently of one another.
- No central service managing state or orchestrating activities.
- New servers can be added without special conditions or knowledge.
- No single point of failure.

4. Models of Redundancy

Link:

<https://www.ni.com/ja-jp/innovations/white-papers/08/redundant-system-basic-concepts.html>

4.1 Standby Redundancy

- Standby redundancy, also known as *Backup Redundancy* is when you have an identical secondary unit to back up the primary unit. The secondary unit typically does not monitor the system but is there just as a spare.
- The standby unit is not usually kept in sync with the primary unit, so it must reconcile its input and output signals on the takeover of the Device Under Control (DUC).
- You also need a third party to be the watchdog, which monitors the system to decide when a switchover condition is met and command the system to switch control to the standby unit and a voter.
- In Standby redundancy, there are two basic types, *Cold Standby* and *Hot Standby*.
 1. **Cold Standby:** The secondary unit is powered off, thus preserving the reliability of the unit, so it takes time to bring it online and it makes it more challenging to reconcile synchronization issues.
 2. **Hot Standby:** The secondary unit is powered up and can optionally monitor the DUC. It can also be used as the watchdog and/or voter to decide when to switch over. It shortens the downtime, which in turn increases the availability of the system.

4.2 N Modular Redundancy

- N Modular Redundancy, also known as Parallel Redundancy, refers to the approach of having multiple units running in parallel.
- All units are highly synchronized and receive the same input information at the same time.
- Their output values are then compared and a voter decides which output values should be used. This model easily provides bumpless switchovers.
- This model typically has faster switchover times than Hot Standby models, but the system is at more risk of encountering a common mode failure across all the units.
- In N Modular Redundancy, there are three main typologies: *Dual Modular Redundancy*, *Triple Modular Redundancy*, and *Quadruple Redundancy*.
 1. **Dual Modular Redundancy (DMR)** uses two functional equivalent units, thus either can control the DUC.

2. **Triple Modular Redundancy (TMR)** uses three functionally equivalent units to provide redundant backup.

3. **Quadruple Modular Redundancy (QMR)** is fundamentally similar to TMR but using four units instead of three to increase reliability.

4.3 1:N Redundancy

- 1:N is a design technique used where you have **a single backup for multiple systems** and this backup is able to function in the place of any single one of the active systems. This technique offers redundancy at a much lower cost than the other models by using **one standby unit for several primary units**.
- This approach only works well when the primary units all have very similar functions, thus allowing the standby to back up any of the primary units if one of them fails.
- The drawbacks of this approach are the added complexity of deciding when to switch and of a switch matrix that can reroute the signals correctly and efficiently.

5. Redundancy at different layers

5.1 Network Redundancy

- [Layer 2 redundancy \(switches\)](#)
 1. **Active/Standby** using Spanning Tree Protocol
 2. **Active/Active** using per VLAN spanning tree protocol and Multiple Spanning Tree Protocol
- [Layer 3 redundancy](#)
 1. **First Hop Redundancy Protocols** are designed to provide redundancy to clients by representing multiple default gateways in a group with a single IP address.

5.2 VM/Server Redundancy

- **How VMware HA Works:** VMware HA provides high availability for virtual machines by pooling them and the hosts they reside on into a cluster. Hosts in the cluster are monitored and in the event of a failure, **the virtual machines on a failed host are restarted on alternate hosts**.
- **Primary and Secondary Hosts in a VMware HA Cluster:**
 1. When you add a host to a VMware HA cluster, an agent is uploaded to the host and configured to communicate with other agents in the cluster.
 2. The first five hosts added to the cluster are designated as primary hosts, and all subsequent hosts are designated as secondary hosts.
 3. The primary hosts maintain and replicate all cluster state and are used to initiate failover actions.
 4. If a primary host is removed from the cluster, VMware HA promotes another (secondary) host to primary status.
 5. If a primary host is going to be offline for an extended period of time, you should remove it from the cluster, so that it can be replaced by a secondary host.
 6. Any host that joins the cluster must communicate with an existing primary host to complete its configuration (except when you are adding the first host to the cluster).
 7. At least one primary host must be functional for VMware HA to operate correctly.
 8. If all primary hosts are unavailable (not responding), no hosts can be successfully

configured for VMware HA. You should consider this limit of five primary hosts per cluster when planning the scale of your cluster.

- One of the primary hosts is also designated as the active primary host and its responsibilities include:
 1. Deciding where to restart virtual machines.
 2. Keeping track of failed restart attempts.
 3. Determining when it is appropriate to keep trying to restart a virtual machine.
- If the active primary host fails, another primary host replaces it.

5.3 Database Redundancy

- Have more than one copy of your data in a database system. It can be either at the table level or at the field level. Usually, the copies are called a replica.
- e.g. Replicas in [ClustrixDB](#) are distributed across the cluster for redundancy and to balance reads, writes, and disk usage.

5.4 Storage Redundancy

- [Azure Storage Redundancy](#)

Locally Redundant Storage (LRS)

LRS ensures that your data stays within a single data center in your chosen region. Data is replicated three times. LRS is cheaper than the other types of redundancy and doesn't provide protection against data center failures.

Zone-Redundant Storage (ZRS)

Only available for block blobs, ZRS keeps three copies of your data across two or three data centers, either within your chosen region or across two regions.

Geo-Redundant Storage (GRS)

This is the type of redundancy that Microsoft recommends by default, and it keeps six copies of your data. Three copies stay in the primary region, and the remaining three are replicated to a secondary region.

Read-Access Geo-Redundant Storage (RA-GRS)

The default redundancy setting, RA-GRS replicates data to a secondary region, where apps also get read access to the data.

5.5 Application Redundancy

Redundant applications, normally server applications, provide backup capability in the event that an application fails. That is, if one server (the primary server) goes out of service for some reason, such as lost connectivity, the other server (the backup server) can act as the primary server, with little or no loss of service.

SQL vs NoSQL

Concepts and considerations for SQL and NoSQL in System Design

1. Common types of NoSQL

1.1. Key-value stores

- An array of key-value pairs. The “key” is an attribute name.
- The value can be a JSON, BLOB(Binary Large Objects), string, etc.
- Used as a collection, dictionaries, associative arrays, etc. Key-value stores help the developer to store schema-less data. They work best for shopping cart contents.
- e.g. Redis, Voldemort, Dynamo, Riak.

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

1.2 Document databases

- Data is stored in documents. It stores and retrieves data as a key-value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.
- Each document can have an entirely different structure.
- The document type is mostly used for CMS systems, **blogging** platforms, **real-time analytics & e-commerce** applications. It should not use for complex transactions that require multiple operations or queries against varying aggregate structures.
- Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes

Col1	Col2	Col3	Col4
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data	Data	Data

Document 1

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Document 2

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Document 3

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Relational Vs. Document

1.3 Wide-column / columnar databases / column-based

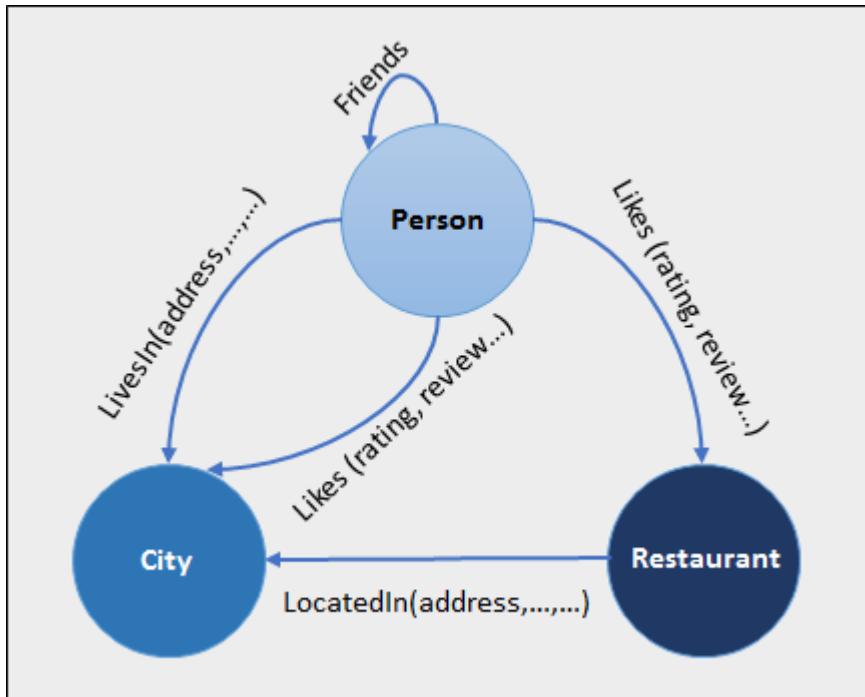
- Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. The values of single column databases are stored contiguously. Each row can have a different number of columns.
- They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN, etc. as the data is readily available in a column.
- Widely used to manage data warehouses, business intelligence, CRM, Library card catalogs.
- HBase, Cassandra, HBase, Hypertable.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

1.4 Graph database

- Data is stored in graph structures
 1. Nodes: entities
 2. Properties: information about the entities
 3. Lines: connections between the entities

- Compared to a relational database where tables are loosely connected, a Graph database is multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.
- Graph base databases are mostly used for **social networks, logistics, spatial data**.
- Neo4J, InfiniteGraph



2. Differences between SQL and NoSQL

2.1 Storage

- SQL: store data in tables.
- NoSQL: have different data storage models.

2.2 Schema

- SQL
 1. Each record conforms to a fixed schema.
 2. Schema can be altered, but it requires modifying the whole database.
- NoSQL:
 1. Schemas are dynamic.

2.3 Querying

- SQL
 1. Use SQL (structured query language) for defining and manipulating the data.
- NoSQL
 1. Queries are focused on a collection of documents.
 2. UnQL (unstructured query language).
 3. Different databases have different syntax.

2.4 Scalability

- SQL
 - 1. Vertically scalable (by increasing the horsepower: memory, CPU, etc) and expensive.
 - 2. Horizontally scalable (across multiple servers); but it can be challenging and time-consuming.
- NoSQL
 - 1. Horizontally scalable (by adding more servers) and cheap.

2.5 ACID

- Atomicity, consistency, isolation, durability
- SQL
 - 1. ACID-compliant
 - 2. Data reliability
 - 3. Guarantee of transactions
- NoSQL
 - 1. Most sacrifice ACID compliance for performance and scalability.

3. Which one to use?

3.1 SQL

- Ensure **ACID** compliance.
 - 1. Reduce anomalies.
 - 2. Protect database integrity.
- Data is structured and unchanging.

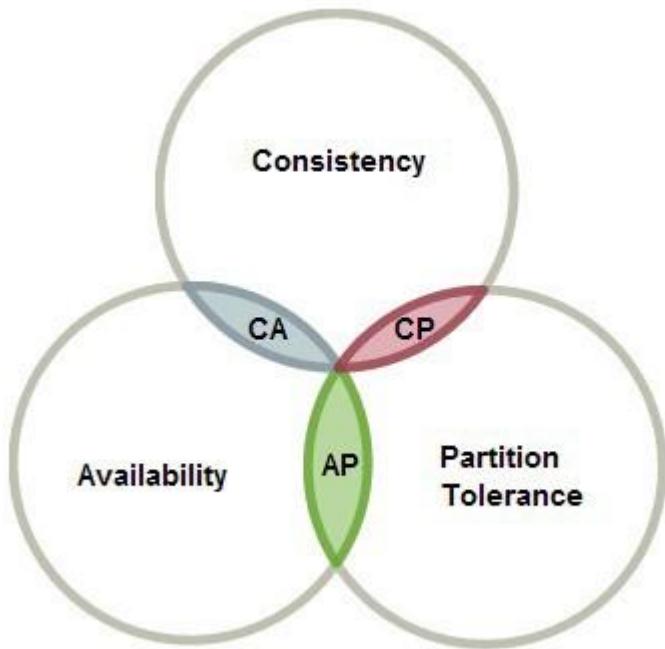
3.2 NoSQL

- Data has little or no structure.
- Make the most of cloud computing and storage.
 - 1. Cloud-based storage requires data to be easily spread across multiple servers to scale up.
- Rapid development.
 - 1. Frequent updates to the data structure.

In a lot of cases, you might need SQL and NoSQL technologies to co-exist side by side in the same system. For example, if you're building a photo-sharing application like Instagram, your photos might reside in a NoSQL database whereas your login/ ACLs information might reside in a SQL database.

CAP Problem

Concepts and considerations for CAP Problem in System Design

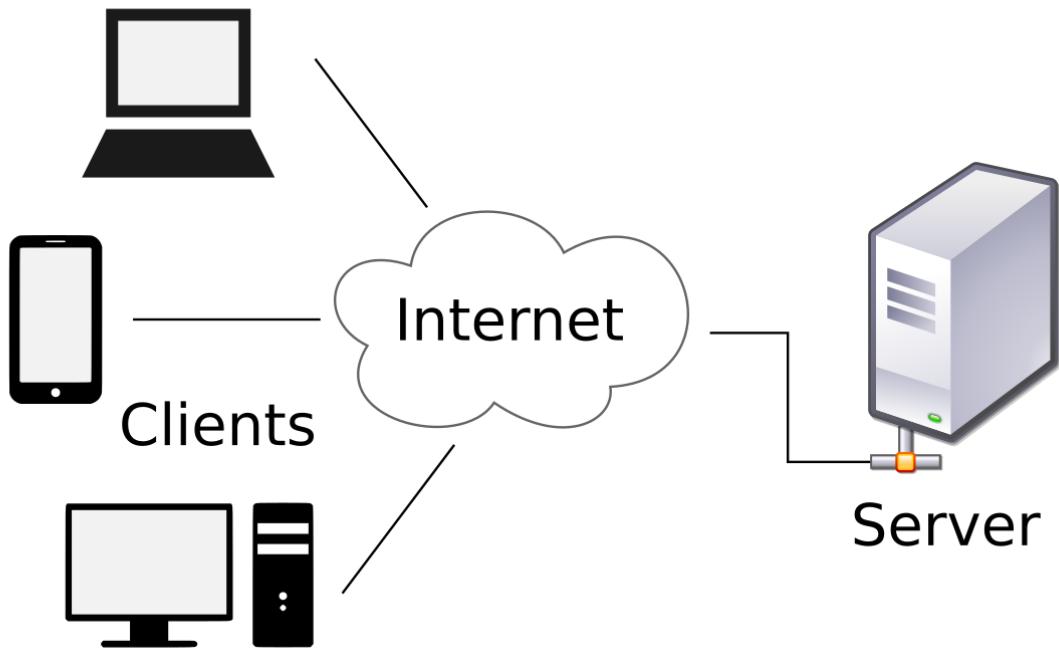


1. Concepts

- **Consistency:** every read receives the most recent write or an error.
- **Availability:** every request receives a response that is not an error.
- **Partition tolerance:** the system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.
- CAP theorem implies that in the presence of a network partition, one has to choose between consistency and availability.
- CAP is frequently misunderstood as if one has to choose to abandon one of the three guarantees at all times. In fact, the choice is really between consistency and availability only when a network partition or failure happens; at all other times, no trade-off has to be made.
- [ACID](#) databases choose consistency over availability.
- [BASE](#) systems choose availability over consistency.

Client-Server Communication

Concepts and considerations for Client-Server Communication in System Design



1. Client-Server Communication

The majority of the communication on the web happens over *HTTP*. There are two modes of data transfer between the client and the server. *HTTP PUSH* & *HTTP PULL*. For every response, there has to be a request first. The client sends the request & the server responds with the data. This is the default mode of HTTP communication, called the *HTTP PULL* mechanism. In *HTTP PUSH*, the client sends the request for particular information to the server, just for the first time, and after that, the server keeps pushing the new updates to the client whenever they are available. There are multiple technologies involved in the *HTTP Push* based mechanism such as:

- *Ajax Long polling*
- *Web Sockets*
- *HTML5 Event Source (Server-sent event)*
- *Message Queues*
- *Streaming over HTTP*

Long-Polling, WebSockets, and Server-Sent Events are popular communication protocols between a client like a web browser and a web server.

HTTP is synchronous in the sense that every request gets a response but asynchronous in the sense that requests take a long time and that multiple requests might be processed in parallel. Therefore, many HTTP clients and servers are implemented in an asynchronous fashion, but without offering an asynchronous API. Using multiple threads is one way to implement asynchronicity, but there are others like callbacks or event loops.

1.1 Ajax Polling

Polling is a standard technique used by the vast majority of **AJAX applications**. The client repeatedly polls (or requests) a server for data, and waits for the server to respond with data. If no data is available, an empty response is returned.

1. The Client opens a connection and requests data from the server **using regular HTTP**.
2. The requested webpage sends requests to the server **at regular intervals** (e.g., 0.5 seconds).
3. **The server calculates the response and sends it back**, like regular HTTP traffic.
4. The client repeats the above three steps periodically to get updates from the server.

Problems:

1. The client has to keep asking the server for any new data.
2. A lot of responses are empty, creating HTTP overhead.

1.2 HTTP Long-Polling

This is a variation of the traditional polling technique that **allows the server to push information to a client** whenever the data is available. The client requests information from the server exactly as in normal polling, but with the expectation that the server may not respond immediately.

1. The client makes an initial request using regular HTTP and then waits for a response.
2. The server **delays its response until an update is available**, or until a timeout has occurred.
3. When an update is available, the server sends a full response to the client.
4. The client typically sends a new long-poll request, either immediately upon receiving a response or after a pause to allow an acceptable latency period.
5. Each Long-Poll request has a timeout. The client has to reconnect periodically after the connection is closed, due to timeouts.

1.3 WebSockets

A persistent full-duplex communication channel over a single TCP connection. **Both server and client can send data at any time**.

- The client establishes a WebSocket connection through a process known as the WebSocket handshake. If the process succeeds, then the server and client can exchange data in both directions at any time.
- Low communication overhead and real-time data transfer.
- The server can send content to the browser without being asked by the client (without browser refresh) and allowing for messages to be passed back and forth while keeping the connection open.

Determining whether to use WebSockets for the job at hand is simple:

- Does your app involve multiple users communicating with each other?
- Is your app a window into server-side data that's constantly changing?

Use cases:

- Social feeds
- Collaborative editing/coding: with WebSockets, we can work on the same document and skip all the merges.
- [Chat apps](#)
- Tracking apps
- Live audience interaction
- IoT device updates

1.4 Server-Sent Event (SSE)

It is a technology that enables a browser (client) to receive automatic updates like text-based event data from a server via an HTTP connection. The client establishes a **persistent and long-term connection** with the server. The server uses this connection to send data to a client. If the client wants to send data to the server, it would require the use of another technology/protocol.

1. The client requests data from a server using regular HTTP.
2. The requested webpage opens a connection to the server.
3. The server sends the data to the client whenever there's new information available.

Use case:

- When real-time traffic from the server to the client is needed.
- When the server generates data in a loop and sends multiple events to the client.
- Facebook/Twitter updates, updating statuses, push notifications, newsletters and news feed, stock price updates, sports results, etc.

2. API Protocols

In recent years, there are many types of API protocols, such as RPC and GraphQL.

	First released	Formatting type	Key strength
SOAP	Late 1990s	XML	Widely used and established
REST	2000	JSON, XML, and others	Flexible data formatting
JSON-RPC	mid-2000s	JSON	Simplicity of implementation
gRPC	2015	Protocol buffers by default; can be used with JSON & others also	Ability to define any type of function
GraphQL	2015	JSON	Flexible data structuring
Thrift	2007	JSON or Binary	Adaptable to many use cases

6 API protocols

2.1 SOAP

SOAP is the oldest web-focused API protocol that remains in widespread use. It is an example of a Remote Procedural Call or RPC. SOAP stands for Simple Object Access Protocol. It requires a fair bit of work to make requests. You have to create XML documents to make calls, and the XML formatting that SOAP requires isn't exactly intuitive. This not only makes call implementation difficult but also makes problems hard to debug because debugging requires parsing through long strings of complex data.

On the other hand, SOAP relies on standard protocols, especially HTTP and SMTP. That means that you can use SOAP in virtually every type of environment because these protocols are available on all operating systems.

2.2. REST

REST, short for Representational State Transfer, is an API protocol that was introduced in a 2000 dissertation by Roy Fielding, whose goal was to solve some of the shortcomings of SOAP.

Like SOAP, REST relies on a standard transport protocol, HTTP, to exchange information between different applications or services. However, REST is more flexible in that it supports a variety of data formats, rather than requiring XML. JSON, which is arguably easier to read and write than XML, is the format that many developers use for REST APIs. **REST APIs can also offer better performance than SOAP because they can [cache information](#).**

2.3 JSON-RPC

While REST supports RPC data structures, it's not the only API protocol in this category. If you like JSON, you may prefer instead to use JSON-RPC, a protocol introduced in the mid-2000s.

Compared to REST and SOAP, JSON-RPC is relatively narrow in scope. It supports a small set of commands and does not offer as much flexibility as a protocol like REST with regard to exactly how you implement it. However, if you like simplicity and have a straightforward use case that falls with JSON-RPC's scope, it can be a better solution than REST.

JSON-RPC [can be used in HTTP and WebSocket](#).

Example of JSON-RPC request and response:

```
// request
```

```
curl -d "{\"jsonrpc\": \"2.0\", \"id\": 2, \"method\": \"subtract\", \"params\": {\"subtrahend\": 23.4, \"minuend\": 42.8},}" --header "Content-Type: application/json"  
http://myserver:8080/openroad/jsonpcservertest// response  
  
{"result":19.400, "id":2, "jsonrpc":"2.0"}
```

2.4 gRPC

gRPC is an open-source API that also falls within the category of RPC. Unlike SOAP, gRPC is much newer, having been released publicly by Google in 2015. Like REST and SOAP, gRPC uses **HTTP** as its transport layer. Unlike these other API protocols, however, gRPC allows developers to **define any kind of function calls** that they want, rather than having to choose from predefined options (like GET and PUT in the case of REST).

Another important advantage of gRPC is that when you make a call to a remote system using gRPC, the call appears to both the sender and the receiver as if it were a local call, rather than a remote one executed over the network. This simulation avoids much of the coding complexity that you'd otherwise have to contend with in order for an application to handle a remote call.

The ability of gRPC to simplify otherwise complex remote calls has helped make it popular in the context of building APIs for **microservices or Docker-based applications**, which entail massive numbers of remote calls.

2.5 GraphQL

In a way, GraphQL is to Facebook what gRPC is to Google: It's an API protocol that was developed internally by Facebook in 2013. It is officially defined as a query language, also represents an effort to overcome some of the limitations or inefficiencies of REST.

One of the key differences between REST and GraphQL is that **GraphQL lets clients structure data however they want when issuing calls to a server**. This flexibility improves efficiency because it means that clients can optimize the data they request, rather than having to request and receive data in whichever prepackaged form the server makes available (which could require receiving more data than the client actually needs, or receiving it in a format that is difficult to use). With GraphQL, the clients can get exactly what they want, without having to worry about transforming the data locally after they receive it.

2.6 Apache Thrift

Like GraphQL, Apache Thrift was born at Facebook and functions essentially as an RPC framework. However, the design goals and target use cases for Thrift differ significantly from those of GraphQL.

Thrift's main selling point is the ease with which it makes it possible to **modify the protocol used by a service once the service has been defined**. Combined with the fact that Thrift also supports an array of different transport methods and several different server-process implementations, this means that Thrift lends itself well to situations where you expect to need to modify or update your API architecture and implementation frequently. You might say that Thrift is great for avoiding "API architecture lock-in" because it ensures that you can easily change your API architecture whenever you need to, instead of being forced to keep the same architecture because of API inflexibility.

On the other hand, some developers might contend that the choice that Thrift gives you when it comes time to implementing the API is not ideal because it leads to less consistency than you get with other API protocols that offer only one way of doing things. If you like rigid consistency and predictability, Thrift may not be the best choice for you.

2.7 SOAP vs REST

Difference	SOAP	REST
Style	Protocol	Architectural style
Function	Function-driven: transfer structured information	Data-driven: access a resource for data
Data format	Only uses XML	Permits many data formats, including plain text, HTML, XML, and JSON
Security	Supports WS-Security and SSL	Supports SSL and HTTPS
Bandwidth	Requires more resources and bandwidth	Requires fewer resources and is lightweight
Data cache	Can not be cached	Can be cached
Payload handling	Has a strict communication contract and needs knowledge of everything before any interaction	Needs no knowledge of the API
ACID compliance	Has built-in ACID compliance to reduce anomalies	Lacks ACID compliance

2.8 Webhooks (it is not API)

Sometimes webhooks are referred to as a reverse API, but this isn't entirely true. They don't run backward, but instead, there doesn't need to be a request initiated on your end, data is sent whenever there's new data available.

To set up a webhook all you have to do is register a URL with the company providing the service you're requesting data from. That URL will accept data and can activate a workflow to turn the data into something useful. In most cases, you can even specify the situations in which your provider will deliver you the data.

Webhooks and APIs differ in how they make requests. For instance, APIs will place calls for data whether there's been a data update response, or not. While webhooks receive calls through HTTP POSTs only when the external system you're hooked to has a data update.

Storage

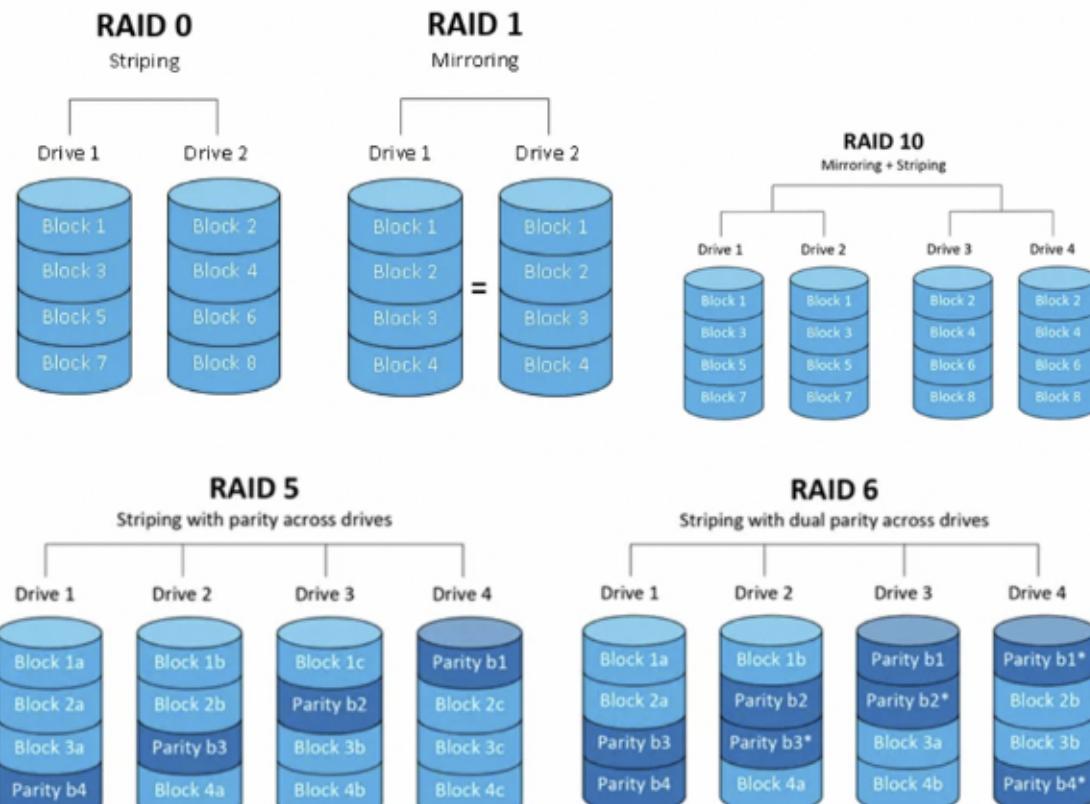
Storage concepts and considerations in System Design

1. Disk — RAID and Volume

1.1 RAID

RAID (Redundant Array of Inexpensive Disks or Redundant Array of Independent Disks) is a data [storage virtualization](#) technology that combines multiple physical [disk drive](#) components into one or more logical units for the purposes of [data redundancy](#), performance improvement, or both.

The standard RAID levels comprise a basic set of RAID configurations that employ the techniques of [striping](#), [mirroring](#), or [parity](#) to create large reliable data stores from multiple general-purpose computer hard disk drives (HDDs) or SSDs (Solid State Drives). A RAID system consists of two or more drives working in parallel. The following figure shows the main 5 RAID levels.



RAID

- **RAID 0** — striping. data are split up into blocks that get written across all the drives in the array.

- **RAID 1** — mirroring, at least two drives that contain the exact same data. If a drive fails, the others will still work.
- **RAID 10** — combining mirroring and striping. It consists of a minimum of four drives and combines the advantages of RAID 0 and RAID 1 in one single system. It provides security by mirroring all data on secondary drives while using striping across each set of drives to speed up data transfers. This means that RAID 10 can provide the speed of RAID 0 with the redundancy of RAID 1.
- **RAID 5** — striping with parity. requires the use of at least 3 drives, striping the data across multiple drives like RAID 0, but also has a **parity** distributed across the drives. In the event of a single drive failure, data is pieced together using the parity information stored on the other drives.
- **RAID 6** — striping with double parity. RAID 6 is like RAID 5, but the parity data are written to two drives. That means it requires at least 4 drives and can withstand 2 drives dying simultaneously.

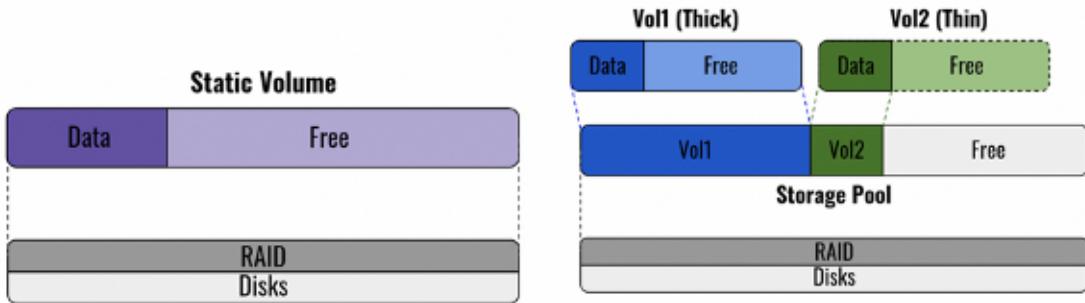
The following table is the comparison for different types of RAID.

Features	RAID 0	RAID 1	RAID 5	RAID 6	RAID 10
Minimum number of drives	2	2	3	4	4
Fault tolerance	None	Single-drive failure	Single-drive failure	Two-drive failure	Up to one disk failure in each sub-array
Read performance	High	Medium	Low	Low	High
Write Performance	High	Medium	Low	Low	Medium
Capacity utilization	100%	50%	67% – 94%	50% – 88%	50%
Typical applications	High end workstations, data logging, real-time rendering, very transitory data	Operating systems, transaction databases	Data warehousing, web serving, archiving	Data archive, backup to disk, high availability solutions, servers with large capacity requirements	Fast databases, file servers, application servers

RAID comparison

1.2 Volume

Volume is a fixed amount of storage on a disk or tape. The term volume is often used as a synonym for the storage itself, but it is possible for a **single disk to contain more than one volume or for a volume to span more than one disk**.



Types of Volumes

Static Volume: A **Static Volume** is a simple and easy-to-use volume that covers all available space on the disks and RAID array selected to create the volume. A static volume does not have a storage pool and therefore can not support advanced storage features such as snapshot and Qtier.

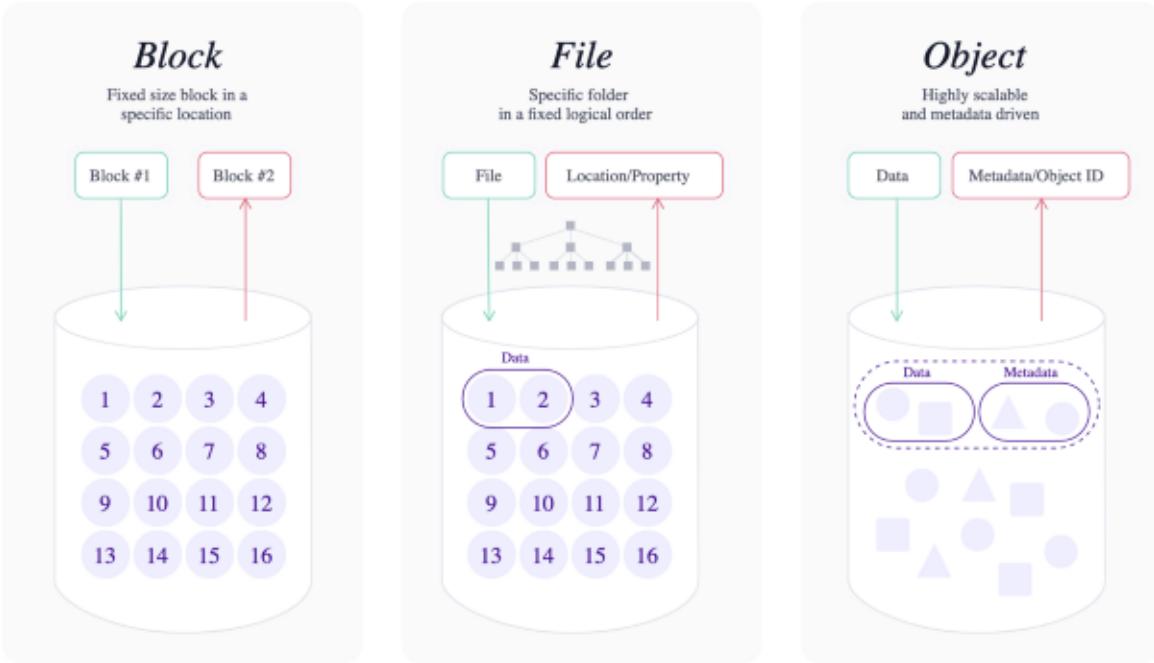
Thin Volume: It must be created inside a **Storage Pool** and allocates space in the storage pool as data is written into the volume. Only the size of the data in the volume is used up from the pool space, and **free space in the volume does not take up any pool space**.

Thick Volume (Flexible): It allocates the total size of the volume upon creation. No matter how much data is actually stored in the volume, the total size of the volume will always be used up in the pool. On the other hand, this space is guaranteed to be available exclusively for this volume, even if other volumes used up all remaining pool free space.

2. File Storage, Block Storage, and Object Storage

Data storage is used for a multitude of reasons. If you're developing an application, you may have users that upload documents, photos, videos, or other files. You'll need somewhere to store user files. If you're a developer, you may use a [content delivery network \(CDN\)](#) and data storage to increase load speed, availability, and reliability. If you're in charge of IT, your main concern may be storage and backup for disaster recovery and business continuity.

Understanding different types of storage is essential to choose the right solution for your business. The main types of storage that are used widely nowadays are **File Storage**, **Block Storage**, and **Object Storage**.



Different types of storage

2.1 File Storage

File storage is a solution to **store data as files and present it to its final users as a hierarchical directories structure**. The main advantage is to provide **a user-friendly solution to store and retrieve files**. To locate a file in file storage, the complete path of the file is required. e.g. /home/images/beach.jpeg. It is economical and easily structured and is usually found on hard drives, which means that they appear exactly the same for the user and on the hard drive.

File Storage is the oldest and most widely used data storage system for direct (DAS) and NAS systems.

File-based storage systems must scale out by adding more systems, rather than scale up by adding more capacity.

Summary: File storage is used for unstructured data and is commonly deployed in [Network Attached Storage](#) (NAS) systems. It uses Network File System (NFS) for Linux, and Common Internet File System (CIFS) or Server Message Block (SMB) protocols for Windows.

2.2 Block Storage

Block storage chops data into blocks (chunks) and stores them as separate pieces. Each block of data is given a unique identifier, which allows a storage system to place the smaller pieces of data wherever it is most convenient. That means that some data can be stored in a Linux environment and some can be stored in a Windows unit.

Block storage is often configured to decouple the data from the user's environment and spread it across multiple environments that can better serve the data. And then, when data is requested, the underlying storage software reassembles the blocks of data from these environments and presents them back to the user.

Block Storage is usually deployed in a storage-area network (SAN) environment and must be tied to a functioning server.

The most common examples of Block Storage are SAN, iSCSI, and local disks.

Block storage is the most commonly used storage type for most applications. It can be either locally or network-attached and are typically formatted with a file system like FAT32, NTFS, EXT3, and EXT4.

Summary: Data is stored in blocks of uniform size, it is ideal for data that needs to be accessed and modified frequently as it provides low-latency. However, it is expensive, complex, and less scalable compared with **File Storage**. It also has limited capability to handle metadata, which means it needs to be dealt with at the application or database level — adding another thing for a developer or systems administrator to worry about.

2.3 Object Storage

Object storage is one of the most recent storage systems. It was created in the cloud computing industry with the requirement of **storing vast amounts of unstructured data**. It is a flat structure in which files are broken into pieces and spread out among hardware. In object storage, the data is broken into discrete units called objects and is kept in a single repository, instead of being kept as files in folders or as blocks on servers.

Object storage volumes work as modular units: each is a self-contained repository that owns:

1. **the data:** images, videos, websites backups
2. **a unique identifier (UID)** that allows the object to be found over a distributed system
3. **the metadata** that describes the data: authors of the file, permissions set on the files, date on which it was created. **The metadata is entirely customizable**

To retrieve the data, the storage operating system uses the **metadata and identifiers**, which distributes the load better and lets administrators apply policies that perform more robust searches.

Object storage requires a simple HTTP API which is used by most clients in all languages. Object storage is cost-efficient: you only pay for what you use. It can scale easily, making it a great choice for public cloud storage. It's a storage system well suited for static data, and its agility and flat nature means it can scale to extremely large quantities of data. The objects have enough information for an application to find the data quickly and are good at storing unstructured data.

Object storage uses erasure coding for data protection. Erasure encoding is a type of algorithm that operates at the object level, spreading data and parity across nodes in a storage cluster. It provides a similar or better level of data redundancy with far less overhead than the HDFS (we will cover it later) three-way replication standard.

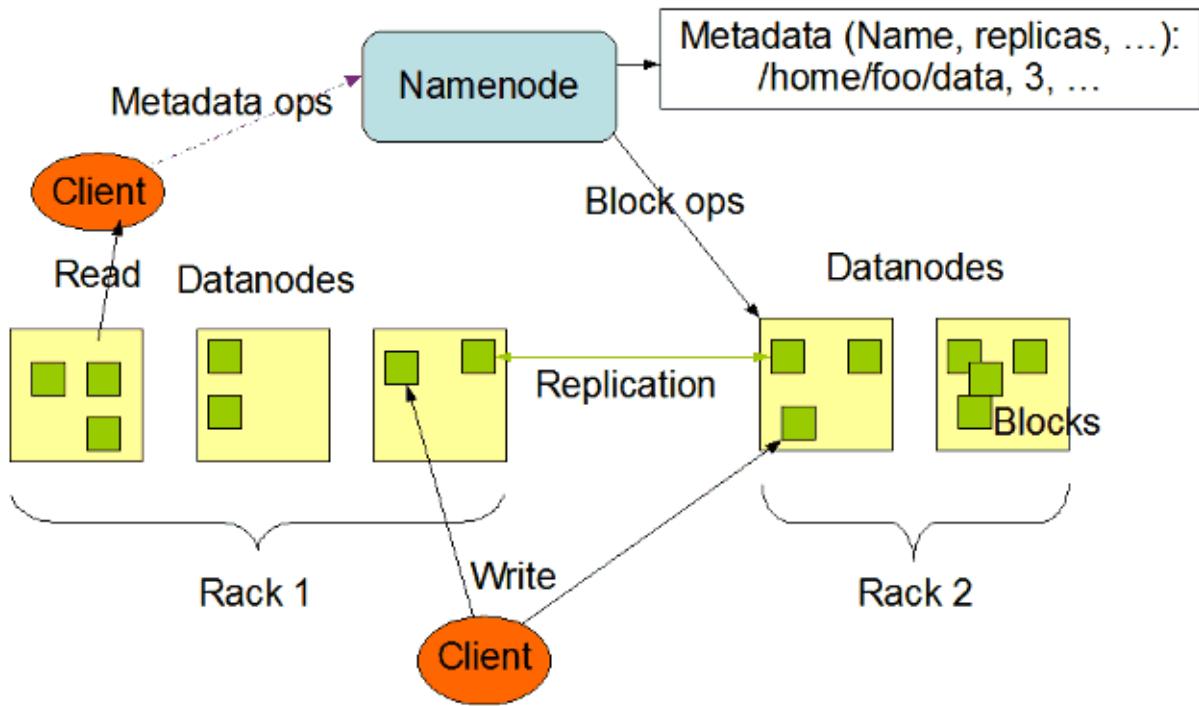
Summary: Data is stored as objects with unique metadata and identifiers. Although, in general, this type of storage is less expensive, but the objects can't be modified — you have to write the object completely at once. Object storage also doesn't work well with traditional databases, because writing objects is a slow process and writing an app to use an object storage API isn't as simple as using file storage.

3. Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. **It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant.** HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is now an Apache Hadoop subproject.

HDFS is designed to reliably store very large files across machines in a large cluster. **It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance ([HDFS requires Block Storage](#)).** The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

HDFS Architecture



HDFS

4. Storage Comparisons

4.1 SAN vs. NAS

NAS stands for [network-attached storage](#). It is a **File Storage** used by enterprises large and small, as well as in SOHO (small office, home office) environments, and by creative professionals and other enthusiasts. NAS allows users to store their files on a centralized appliance or storage array.

These devices are accessible over a network using an ethernet connection and file protocols like **NFS** (Network File System) or **SMB/CIFS** (Server Message Block/Common Internet File System). Often, they contain enterprise-grade NAS drives, hard drives built to withstand operating all-day, every-day, and provide better overall performance relative to their desktop counterparts.

Some small businesses and most enterprise-grade NAS devices ship with RAID support. Typically, the more high-end the NAS system, the more RAID configuration options are available.

A **SAN** is a **block-based storage**, leveraging a high-speed architecture that connects servers to their logical disk units (LUNs). A LUN is a range of blocks provisioned from a pool of shared storage and presented to the server as a logical disk.

Both SAN and NAS are methods of managing storage centrally and sharing that storage with multiple hosts (servers). However, **NAS is Ethernet-based, while SAN can use Ethernet and Fibre Channel**. In addition, while SAN focuses on high performance and low latency, NAS focuses on ease of use, manageability, scalability, and lower total cost of ownership (TCO). Unlike SAN, NAS storage controllers partition the storage and then own the file system. Effectively this makes a NAS server look like a Windows or UNIX/Linux server to the server consuming the storage.

4.2 NAS vs. HDFS

- HDFS is the primary storage system of Hadoop. HDFS designs to store very large files running on a cluster of commodity hardware. While **NAS** is a file-level computer data storage server. NAS provides data access to a heterogeneous group of clients.
- HDFS distributes blocks across all the machines in a [Hadoop cluster](#). While NAS, data stores on dedicated hardware.
- Hadoop HDFS is designed to work with [MapReduce](#) Framework. In MapReduce Framework computation move to the data instead of Data to computation. NAS is not suitable for MapReduce, as it stores data separately from the computations.
- Hadoop HDFS runs on the cluster commodity hardware which is cost-effective. While a NAS is a high-end storage device that includes a high cost.

4.3 Block Storage vs. Object Storage

- Object storage costs about 1/3 to 1/5 as much as block storage. The detailed comparison is as below.

	OBJECT STORAGE	BLOCK STORAGE
PERFORMANCE	Performs best for big content and high stream throughput	Strong performance with database and transactional data
GEOGRAPHY	Data can be stored across multiple regions	The greater the distance between storage and application, the higher the latency
SCALABILITY	Can scale infinitely to petabytes and beyond	Addressing requirements limit scalability
ANALYTICS	Customizable metadata allows data to be easily organized and retrieved	No metadata

5. Choose the right datastore

Choosing the right datastore (storage or database) for your business is not easy. You will need to take the pricing, features, and performance into account. This section summarizes some of the common selections for different use cases.

The general idea is to store metadata in a relational database or Distributed Key-Value store like Dynamo (key-value) or Cassandra (wide-column). Since NoSQL data stores do not support ACID properties in favor of scalability and performance, we need to incorporate the support for ACID properties programmatically in the logic of our services if we choose NoSQL.

To store other contents such as photos, videos, texts, binaries, and messages, we have to choose the right storage based on our requirements.

Photo-sharing services like Instagram, also apply to Twitter

- Store photos in a distributed file storage like [HDFS](#) or [S3](#) (object storage).
- Store data about users, their uploaded photos, and people they follow in RDBMS, but it is difficult to scale. So we may also do below:
 1. Store the schema in a distributed key-value store to enjoy the benefits offered by NoSQL. All the metadata related to photos can go to a table where the **Key** would be the **PhotoID** and the **Value** would be an object containing **PhotoLocation**, **UserLocation**, **CreationTimestamp**, etc.
 2. To store relationships between users and photos and the list of people a user follows, we can use a wide-column datastore like [Cassandra](#). For the **UserPhoto** table, the **Key** would be **UserID** and the **Value** would be the list of **PhotoIDs** the user owns, stored in different columns. We will have a similar scheme for the **UserFollow** table.

URL shortening service like TinyURL

- Since we anticipate storing billions of rows, and we don't need to use relationships between objects — a NoSQL store like [DynamoDB](#) (key-value), [Cassandra](#) (wide-column) or [Riak](#) (key-value) is a better choice.

File hosting service like Dropbox, Google Drive, Onedrive

- The metadata database can be a relational database such as MySQL, or a NoSQL database service such as DynamoDB.
- To store files, we can use Block storage in which files can be stored in small parts or chunks (say 4MB).
- Object Storage is used by Dropbox to store files.

Instant messaging service like Facebook Messenger

- To store messages, we need to have a database that can support a very high rate of small updates and also fetch a range of records quickly. We cannot use RDBMS like MySQL or NoSQL like MongoDB because we cannot afford to read/write a row from the database every time a user receives/sends a message. **Our requirements can**

be easily met with a wide-column database solution like [HBase](#). We can store multiple values against one key into multiple columns.

Video sharing services like Youtube

- Video metadata and user data: RDBMS
- Thumbnails: **Bigtable**, as it combines multiple files into one block to store on the disk and is very efficient in reading a small amount of data.
- Videos can be stored in a distributed file storage system like [HDFS](#) or [GlusterFS](#).
- Spotify uses object storage to store songs.

Real-time suggestion ([auto-complete system](#)) service

- Use the **Trie data structure**. The storage can be an in-memory cache (Redis or Memcached), a database, or even a file.
- Take a snapshot of the trie periodically and store it in a file. This will enable us to rebuild a trie if the server goes down.

Web Crawler

- Use RDBMS to store the meta-data associated with the pages.
- Store URLs on a disk for **frontier**.

[Google Analytics \(GA\) like system](#)

- **Apache Kafka** is used for building real-time streaming data pipelines that reliably get data between many independent systems or applications, it allows:
 1. Publishing and subscribing to streams of records;
 2. **Storing streams of records** in a fault-tolerant, durable way
- The ingested data is read directly from Kafka by **Apache Spark** for stream processing and creates **Timeseries Ignite RDD** (Resilient Distributed Datasets). **Apache Ignite** is a distributed memory-centric database and caching platform that is used by Apache Spark users to achieve true in-memory performance.
- Use **Apache Cassandra** (Column NoSQL based on BigTable) as storage for persistence writes from Ignite. It has **great write and read performance**.

6. Storage options in the Cloud

File Storage

- [Amazon Elastic File System \(EFS\)](#) is ideal for use cases like large content repositories, development environments, media stores, or user home directories.
- [Azure Files](#) offers fully managed file shares in the cloud that are accessible via the industry standard [Server Message Block \(SMB\) protocol](#) or [Network File System \(NFS\) protocol](#).
- Google Cloud [Filestore](#): High-performance file storage for Google Cloud users.

Block Storage

- [Amazon Elastic Block Store \(EBS\)](#) is provisioned with each virtual server and offers the ultra-low latency required for high-performance workloads.
- Azure provides a [managed disks service](#) for block volumes, which you can use with Azure Virtual Machines (VMs).
- Google Cloud: [Zonal persistent disk](#): Efficient, reliable block storage; [Regional persistent disk](#): Regional block storage replicated in two zones; [Local SSD](#): High performance, transient, local block storage.

Object Storage

- [Amazon Simple Storage Service \(Amazon S3\)](#) is ideal for building modern applications from scratch that requires scale and flexibility, and can also be used to import existing data stores for analytics, backup, or archive.
- [Azure Blob Storage](#): For users with large amounts of unstructured data to store in the cloud, Blob storage offers a cost-effective and scalable solution. Every blob is organized into a container with up to a 500 TB storage account capacity limit.
- Google [Cloud Storage buckets](#): Affordable object storage.

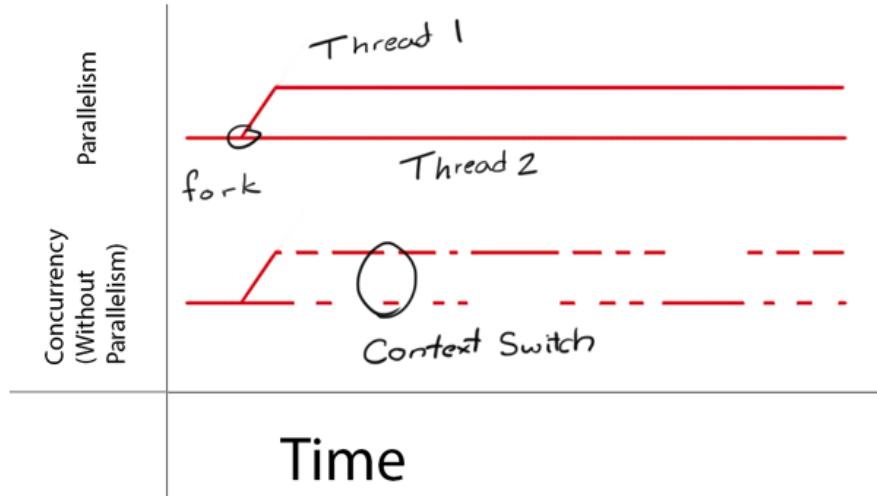
Concurrency

1.1 Threads and processes

- A thread is a basic unit of CPU utilization. It is also referred to as a “lightweight process”.
- A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system, a process may be made up of multiple threads of execution that execute instructions concurrently.
- [CPU is not required when a process is handling the IO request](#).

1.2 Concurrency and parallelism

- Concurrency means executing multiple tasks at the same time but not necessarily simultaneously. In a single-core environment, concurrency is achieved via a process called [context-switching](#). If it's a multi-core environment, concurrency can be achieved through parallelism.
- Parallelism is concerned with utilizing multiple processors/cores to perform two or more tasks simultaneously. Parallel computing in computer science refers to the process of performing multiple calculations simultaneously.
- ***In a single-core environment, concurrency happens*** with tasks executing over the same time period via context switching. i.e at a particular time period, only a single task gets executed.
- ***In a multi-core environment, concurrency can be achieved via parallelism*** in which multiple tasks are executed simultaneously.



1.3 Synchronous and asynchronous

- In the asynchronous programming model, tasks are executed one after another. Each task waits for any previous task to complete and then gets executed.
- In an asynchronous programming model, when one task gets executed, you could switch to a different task without waiting for the previous one to get completed.
- Synchronous & asynchronous programming model helps us to achieve **concurrency**.
- An asynchronous programming model in a multi-threaded environment is a way to achieve **parallelism**.

1.4 Deadlock and starvation

- Starvation and Deadlock are situations that occur when the processes that require a resource are delayed for a long time.
- **Starvation** occurs if a process is indefinitely postponed. This may happen if the process requires a resource for execution that is never allocated or if the process is never provided with a processor for some reason. Some of the common causes of starvation are as follows:
 1. If a process is never provided the resources it requires for execution because of faulty resource allocation decisions, then starvation can occur.
 2. A lower priority process may wait forever if higher priority processes constantly monopolize the processor.
 3. Starvation may occur if there are not enough resources to provide for every process as required.
 4. If a random selection of processes is used then a process may wait for a long time because of non-selection.
- Handle starvation:
 1. An independent manager can be used for the allocation of resources. This resource manager distributes resources fairly and tries to avoid starvation.
 2. Random selection of processes for resource allocation or processor allocation should be avoided as they encourage starvation.

- 3. The priority scheme of resource allocation should include concepts such as aging, where the priority of a process is increased the longer it waits. This avoids starvation.
- A **deadlock** occurs when two or more processes need some resources to complete their execution that is held by the other process.
- A deadlock will only occur if the four Coffman conditions hold true:
 - Mutual Exclusion:** it implies there should be a resource that can only be held by one process at a time. This means that the resources should be non-sharable.
 - Hold and Wait:** A process can hold multiple resources and still request more resources from other processes that are holding them.
 - No preemption:** A resource cannot be preempted from a process by force. A process can only release a resource voluntarily.
 - Circular wait:** A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process, and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain.
- Handle Deadlock
 - Prevention is done by negating one of the above mentioned necessary conditions for deadlock.
 - Let deadlock occur, then do preemption to handle it once occurred.
 - If the deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

[Parallel Algorithm](#)

- A **sequential algorithm** is an algorithm in which some consecutive steps of instructions are executed in a chronological order to solve a problem.
- A **parallel algorithm** is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.
- A **parallel computer** is a computer with many processors. It is not easy to divide a large problem into **sub-problems**. Sub-problems may have data dependency among them. Therefore, the processors have to communicate with each other to solve the problem. The time needed by the processors in communicating with each other is more than the actual processing time.
- There are four models of computation
 - Single Instruction stream, Single Data stream (SISD) computers
 - Single Instruction stream, Multiple Data stream (SIMD) computers
 - Multiple Instruction stream, Single Data stream (MISD) computers
 - Multiple Instruction stream, Multiple Data stream (MIMD) computers
- Parallel algorithms are designed to improve the computation speed of a computer. For analyzing a Parallel Algorithm, we normally consider the following parameters:
 - Time complexity (Execution Time)
 - Total number of processors used
 - Total cost

[Consistency, wiki](#)

- Consistency models are used in distributed systems like distributed shared memory systems or distributed data stores (such as filesystem, databases, optimistic replication systems, or web caching).
- **Strict consistency** is the strongest consistency model. Under this model, a write to a variable by any processor needs to be seen instantaneously by all processors.

Coherence

- **Memory coherence** is an issue that affects the design of computer systems in which two or more processors or cores share a common area of memory.
- **Cache coherence** is the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

Networking

Inter-Process Communication (IPC)

- Interprocess communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. Processes can communicate with each other using these two ways:
 1. Shared Memory: e.g. Producer-Consumer problem using buffer
 2. Message passing

Remote Procedure Call (RPC)

- **Remote Procedure Call (RPC)** is a powerful technique for constructing **distributed, client-server based applications**. It is based on extending the conventional local procedure calling so that the **called procedure need not exist in the same address space as the calling procedure**. The two processes may be on the same system, or they may be on different systems with a network connecting them.

Throughput, latency, and bandwidth

- **Latency** — The *time* taken for a packet to be transferred across a network. You can measure this as a one-way to its destination or as a round trip.
- **Throughput** — The *quantity* of data being sent and received within a unit of time.
- **Bandwidth** — the name given to **the number of packets that can be transferred throughout the network**.
- The **bandwidth** of a network specifies the **maximum number of conversations that the network can support**. Conversations are exchanges of data from one point to another.
- **Latency** is used to measure **how quickly these conversations take place**. The more latency there is, the longer these conversations take to hold.
- The level of latency determines the maximum throughput of a conversation. **Throughput is how much data can be transmitted within a conversation.**

Abstraction

1.1 Cache memory

- Cache memory, also called CPU memory, is high-speed static random access memory (SRAM) that a computer microprocessor can access more quickly than it can access regular random access memory (RAM, or DRAM). This memory is typically integrated directly into the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU.
- Memory cache is also known as the cache-store and random access memory cache (RAM cache). A memory cache is an external cache that is deployed or part of a computer's RAM. It is typically built using static RAM technology, which is much faster than dynamic RAM (DRAM). Despite being faster than RAM, a memory cache is slower than a CPU cache, primarily because it is not in close proximity to the processor.

1.2 Various levels of cache memory

- When you run a program, these instructions have to make their way from the primary storage to the CPU. The data first gets loaded up into the RAM and is then sent to the CPU. CPUs these days are capable of carrying out *a gigantic number of instructions per second*. To make full use of its power, the CPU needs access to super-fast memory. This is where the cache comes in.
- A CPU cache is divided into three main 'Levels', L1, L2, and L3. The hierarchy here is according to the speed, and thus, the size of the cache.
- **L1 Cache** has the data the CPU is most likely to need while completing a certain task. As far as the size goes, the L1 cache typically goes up to 256KB (1–2MB in Some server chipsets), it is also usually split two ways, into the instruction cache and the data cache.
- **L2 Cache** is slower than the L1 cache, but bigger in size. Its size typically varies between 256KB to 8MB. In most modern CPUs, the L1 and L2 caches are present on the CPU cores themselves, with each core getting its own cache.
- **L3 Cache** is the largest cache memory unit, and also the slowest one. It can range between 4MB to upwards of 50MB. Modern CPUs have dedicated space on the CPU die for the L3 cache, and it takes up a large chunk of the space. L3 is usually double the speed of RAM.
- The data flows from the RAM to the L3 cache, then the L2, and finally L1. When the processor is looking for data to carry out an operation, it first tries to find it in the L1 cache. If the CPU is able to find it, the condition is called a cache hit. It then proceeds to find it in L2, and then L3.

Real-World Performance and Estimation

1.1 Performance of RAM, disk, SSD, and network

- Memory speed is in nanoseconds (100 thousand times faster than disk)
- 10GbE Network speed is in microseconds (~50)

- Flash speed is in microseconds (between 20–500+)
- Disk speed is in milliseconds (between 4–7)
- Speed: Processor > DRAM > Network > Flash > Hard drive
- A machine can often execute a couple of millions of instructions per second. $5 * 10^6$

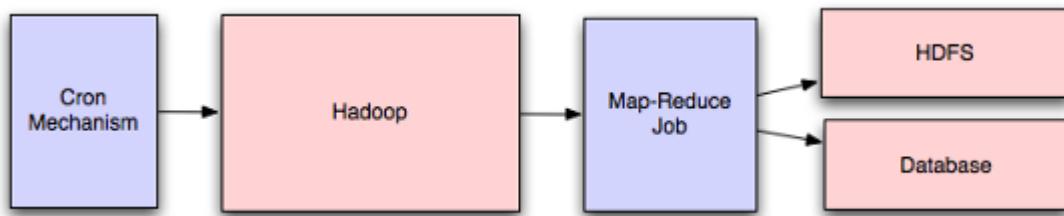
1.2 Speed of everything

- https://medium.com/@wizzard_harshit/back-of-the-envelope-calculations-stay-one-step-ahead-while-designing-system-architectures-cfe78691c79e

Map-Reduce

1.1 Concepts

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](#). Hadoop MapReduce is a software framework for easily writing applications that process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.



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

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

A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then inputted to the **reduce tasks**. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them, and re-executes the failed tasks.

Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System (see [HDFS Architecture Guide](#)) are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master ResourceManager, one worker NodeManager per cluster-node, and MRAppMaster per application (see [YARN Architecture Guide](#)).

Minimally, applications specify the input/output locations and supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract classes. These, and other job parameters, comprise the *job configuration*.

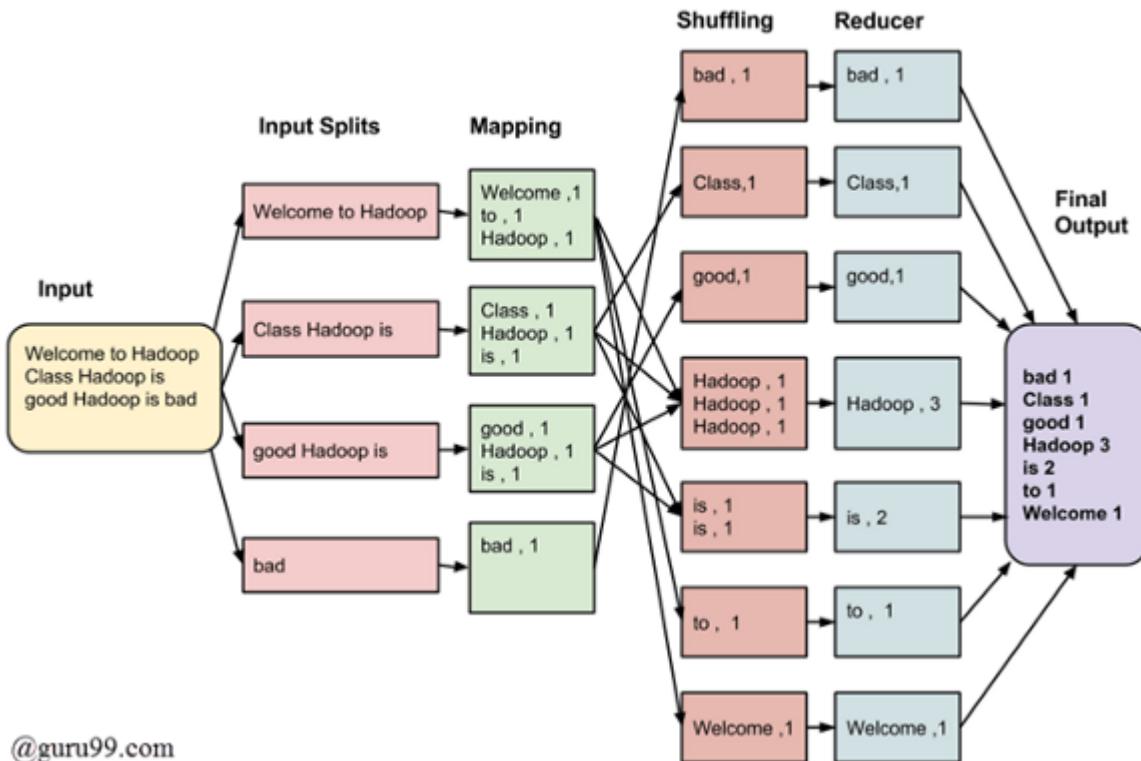
The Hadoop *job client* then submits the job (jar/executable etc.) and configuration to the ResourceManager which then assumes the responsibility of distributing the software/configuration to the workers, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

1.2 How does it work

A MapReduce framework (or system) is usually composed of three operations (or steps):

1. **Map:** each worker node applies the map function to the local data, and writes the output to temporary storage. A master node ensures that only one copy of the redundant input data is processed.
2. **Shuffle:** worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node.
3. **Reduce:** worker nodes now process each group of output data, per key, in parallel.

1.3 Example



1.4 Use case

Consider an e-commerce system that receives a million requests every day to process payments. There may be several exceptions thrown during these requests such as “payment declined by a payment gateway,” “out of inventory,” and “invalid address.” A developer wants to analyze the last four days’ logs to understand which exception is thrown how many times.

To perform analysis on logs that are bulky, with millions of records, MapReduce is an apt programming model. Multiple mappers can process these logs simultaneously: one mapper could process a day’s log or a subset of it based on the log size and the memory block available for processing in the mapper server.

We can assume the Hadoop framework runs just four mappers. Mapper 1, Mapper 2, Mapper 3, and Mapper 4. The value input to the mapper is one record of the log file. The key could be a text string such as “filename + line number.” The mapper, then, processes each record of the log file to produce key-value pairs, and the final output may look something like this:

Reducer 1: <Exception A, 9>

Reducer 2: <Exception B, 5>

Reducer 3: <Exception C, 4>

Hadoop and Spark

Spark and Hadoop often work together with Spark processing data that sits in HDFS, Hadoop’s file system. But, they are distinct and separate entities, each with its own pros and cons and specific business use cases.

1.1 Hadoop

It’s a general-purpose form of distributed processing that has several components: the Hadoop Distributed File System (HDFS), which stores files in a Hadoop-native format and parallelizes them across a cluster; YARN, a schedule that coordinates application runtimes; and MapReduce, the algorithm that actually processes the data in parallel.

In addition to these basic components, Hadoop also includes Sqoop, which moves relational data into HDFS; Hive, a SQL-like interface allowing users to run queries on HDFS; and Mahout, for machine learning. In addition to using HDFS for file storage, Hadoop can also now be configured to use **S3 buckets or Azure blobs** as input.

1.2 Spark

It’s also a top-level Apache project focused on processing data in parallel across a cluster, but the **biggest difference is that it works in memory**.

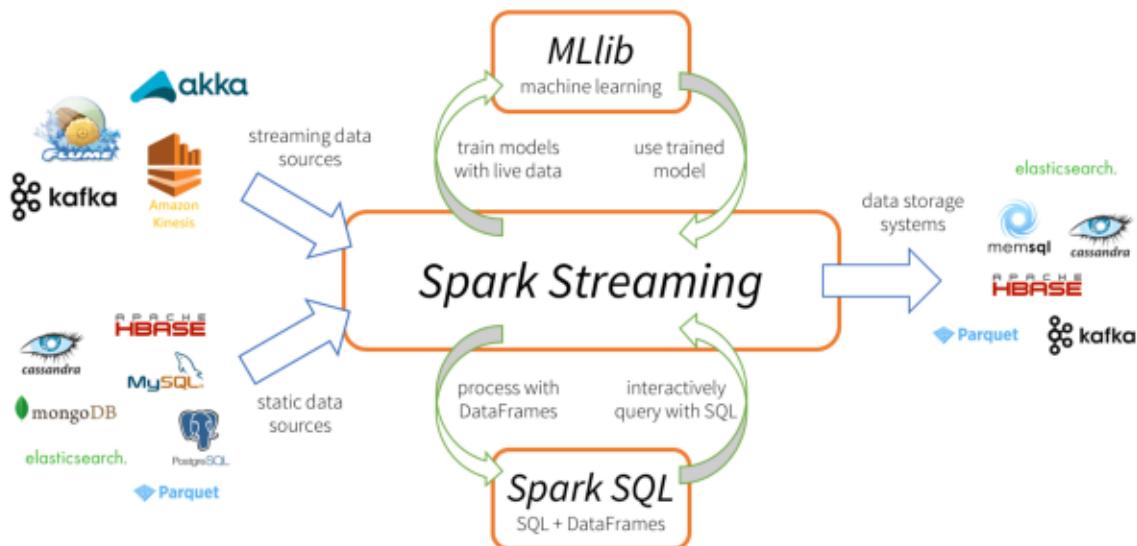
Whereas **Hadoop reads and writes files to HDFS, Spark processes data in RAM using a concept known as an RDD, Resilient Distributed Dataset**. Spark can run either in

stand-alone mode, with a Hadoop cluster serving as the data source, or in conjunction with Mesos. In the latter scenario, the Mesos master replaces the Spark master or YARN for scheduling purposes.

One use case of Spark is described in this [blog](#). The ingested data is read directly from Kafka by Apache Spark for stream processing and creates Timeseries Ignite RDD. Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Spark streaming process Kafka data streams; create and share Ignite RDDs across Apache Ignite which is a distributed memory-centric database and caching platform.



Top Interview Question

1. Designing a URL Shortening service like TinyURL

Solution 1:

A URL shortener service creates a **short url/aliases/tiny url** against a long url. Moreover, when a user clicks on the tiny url, he gets redirected to the original url.

Features

There can be a myriad of features even in seemingly small services like URL shortening. Hence it's always a good idea to ask the interviewer what features are expected in the service. This will then become the basis for functional requirements.

How long would a tiny url be ? Will it ever expire ?

Assume once a url is created it will remain forever in the system.

Can a customer create a tiny url of his/her choice or will it always be service generated ? If a user is allowed to create customer shortened links, what would be the maximum size of custom url ?

Yes, a user can create a tiny url of his/her choice. Assume maximum character limit to be 16.

How many url shortening requests are expected per month ?

Assume 100 million new URL shortenings per month

Do we expect service to provide metrics like most visited links ?

Yes. Services should also aggregate metrics like number of URL redirections per day and other analytics for targeted advertisements.

System Design goals

Functional Requirements:

- Service should be able to create shortened url/links against a long url
- Click to the short URL should redirect the user to the original long URL
- Shortened link should be as small as possible
- Users can create custom url with maximum character limit of 16
- Service should collect metrics like most clicked links
- Once a shortened link is generated it should stay in system for lifetime

Non-Functional Requirements:

- Service should be up and running all the time
- URL redirection should be fast and should not degrade at any point of time (Even during peak loads)
- Service should expose REST API's so that it can be integrated with third party applications

Traffic and System Capacity

Traffic

Assuming 200:1 *read/write* ratio

Number of unique shortened links generated per month = 100 million

Number of unique shortened links generated per seconds = 100 million /(30 days * 24 hours * 3600 seconds) ~ **40 URLs/second**

With 200:1 *read/write* ratio, number of redirections = 40 URLs/s * 200 = **8000 URLs/s**

Storage

Assuming lifetime of service to be 100 years and with 100 million shortened links creation per month, total number of data points/objects in system will be = 100 million/month * 100 (years) * 12 (months) = 120 billion

Assuming size of each data object (*Short url, long url, created date* etc.) to be 500 bytes long, then total require storage = 120 billion * 500 bytes =**60TB**

Memory

Following Pareto Principle, better known as the 80:20 rule for caching. (80% requests are for 20% data)

Since we get 8000 read/redirection requests per second, we will be getting 700 million requests per day:

8000/s * 86400 seconds =**~700 million**

To cache 20% of these requests, we will need ~70GB of memory.

$$0.2 * 700 \text{ million} * 500 \text{ bytes} = \sim 70\text{GB}$$

Estimates

Shortened URLs generated : **40/s**

Shortened URLs generated in 100 years : **120 billion**

Total URL requests/redirections : **8000/s**

Storage for 100 years : **60TB**

Cache memory : **70GB**

High Level Design

Following is the high level design of our URL service. This is a rudimentary design. We will optimize this further as we move along.

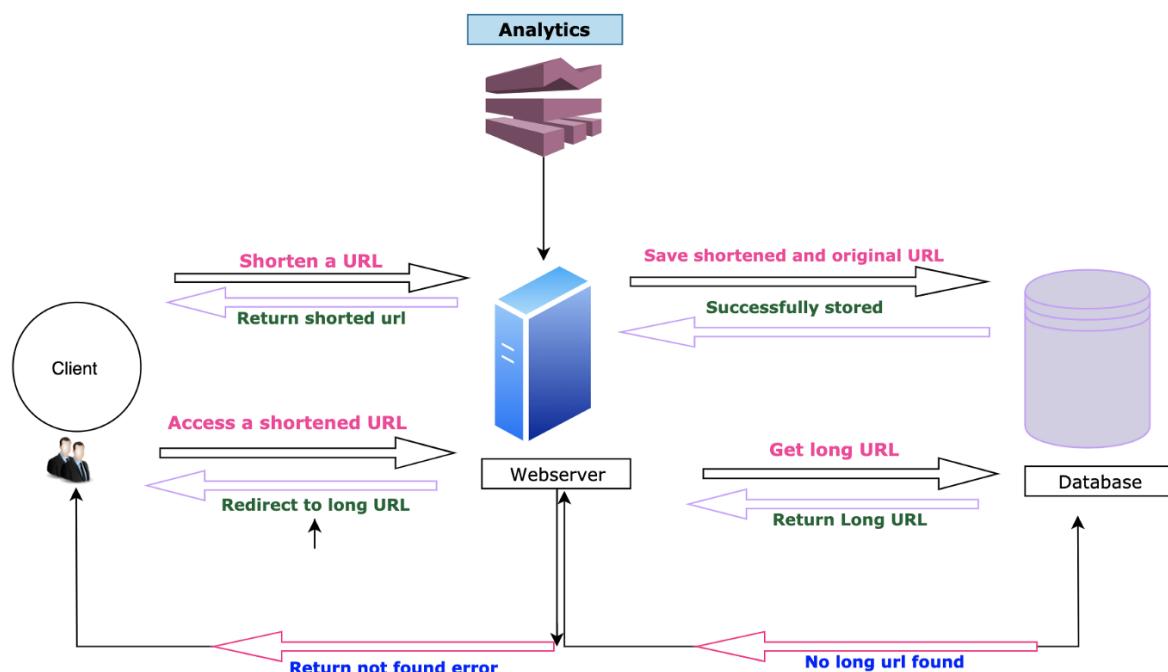


Fig. 1 A Rudimentary design for URL service

Problems with above design :

1. There is only one WebServer which is single point of failure (SPOF)
2. System is not scalable
3. There is only single database which might not be sufficient for 60 TB of storage and high load of 8000/s read requests

To cater above limitations we :

1. Added a load balancer in front of WebServers

2. Sharded the database to handle huge object data
3. Added cache system to reduce load on the database.

We will delve further into each component when we will go through the algorithms in later sections

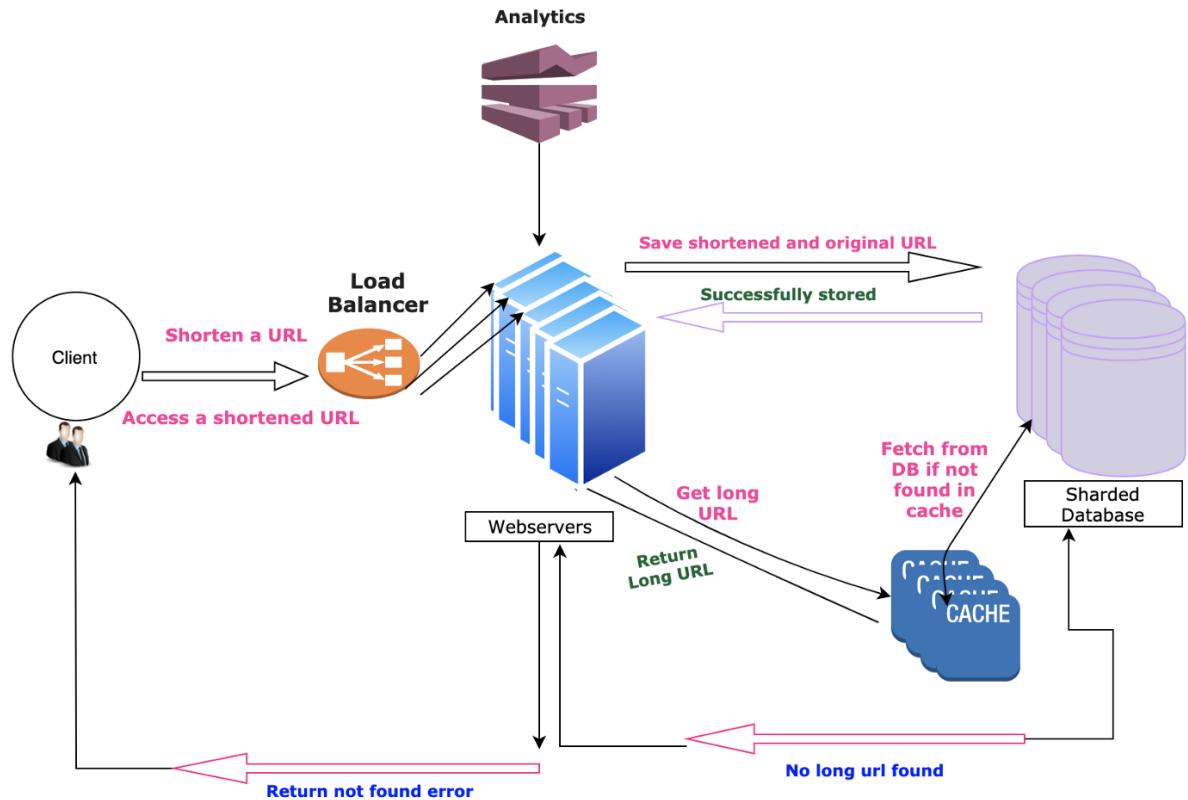


Fig. 2 Scalable high level design

Algorithm REST Endpoints

Let's starts by making two functions accessible through REST API:

create(long_url, api_key, custom_url)

POST

Tinyrl : POST : <https://tinyurl.com/app/api/create>

Request Body: {url=long_url}

Return OK (200), with the generated short_url in data

long_url: A long URL that needs to be shortened.

api_key: A unique API key provided to each user, to protect from the spammers, access, and resource control for the user, etc.

`custom_url(optional)`: The custom short link URL, user want to use.

Return Value: The short Url generated, or error code in case of the inappropriate parameter.

GET: `/{short_url}`

Return a http redirect response(302)

Note : “HTTP 302 Redirect” status is sent back to the browser instead of “HTTP 301 Redirect”. A **301** redirect means that the page has permanently moved to a new location. A **302** redirect means that the move is only temporary. Thus, returning 302 redirect will ensure all requests for redirection reaches to our backend and we can perform analytics (Which is a functional requirement)

`short_url`: The short URL generated from the above function.

Return Value: The original long URL, or invalid URL error code.

Database Schema

Let's see the data we need to store:

Data Related to user

1. **User ID:** A unique user id or API key to make user globally distinguishable
2. **Name:** The name of the user
3. **Email:** The email id of the user
4. **Creation Date:** The date on which the user was registered

Data Related to ShortLink

1. **Short Url:** 6/7 character long unique short URL
2. **Original Url:** The original long URL
3. **UserId:** The unique user id or API key of the user who created the short URL

Shortening Algorithm

For shortening a url we can use following two solutions (URL encoding and Key Generation service). Let's walk through each of them one by one.

a.) **URL Encoding**

- i.) URL encoding through **base62**
- ii) URL encoding through **MD5**

b.) **Key Generation Service (KGS)**

URL encoding through base62

A base is a number of digits or characters that can be used to represent a particular number.

Base 10 are digits [0–9], which we use in everyday life and

base 62 are [0–9][a-z][A-Z]

Let's do a back of the envelope calculation to find out how many characters shall we keep in our tiny url.

URL with length 5, will give $62^5 = \sim 916$ Million URLs

URL with length 6, will give $62^6 = \sim 56$ Billion URLs

URL with length 7, will give $62^7 = \sim 3500$ Billion URLs

Since we required to produce **120 billion** URLs, with **7 characters** in base62 we will get **~3500 Billion URLs**. Hence each of tiny url generated will have 7 characters

How to get unique '7 character' long random URLs in base62

Once we have decided number of characters to use in Tiny URL (7 characters) and also the base to use (base 62 [0–9][a-z][A-Z]), then the next challenge is how to generate unique URLs which are 7 characters long.

Technique 1 —Short url from random numbers:

We could just make a **random choice for each character** and check if this tiny url **exists** in DB or not. If it doesn't exist return the tiny url else continue rolling/retrying. As more and more 7 characters short links are generated in Database, we would require 4 rolls before finding non-existing one short link which will slow down tiny url generation process.

Java code for randomly generating 7 characters long tiny url

```
private static final int NUM_CHARS_SHORT_LINK = 7;
private static final String ALPHABET =
"ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
private Random random = new Random();
public String generateRandomShortUrl() {
    char[] result = new char[NUM_CHARS_SHORT_LINK];
    while (true) {
        for (int i = 0; i < NUM_CHARS_SHORT_LINK; i++) {
            int randomIndex = random.nextInt(ALPHABET.length() - 1);
            result[i] = ALPHABET.charAt(randomIndex);
        }
        String shortLink = new String(result);
        // make sure the short link isn't already used
        if (!DB.checkShortLinkExists(shortLink)) {
            return shortLink;
        }
    }
}
```

Technique 2 — Short urls from base conversion:

Think of the seven-bit short url as a hexadecimal number (0–9, a-z, A-Z) (For e.g. **aKc3K4b**). Each short url can be mapped to a decimal integer by using base conversion and vice versa.

How do we do the base conversion? This is easiest to show by an example.

Take the number 125 in base 10.

It has a 1 in the 100s place, a 2 in the 10s place, and a 5 in the 1s place. In general, the places in a base-10 number are:

- 10^0
- 10^1
- 10^2
- 10^3
- etc

The places in a base-62 number are:

- 62^0
- 62^1
- $62^2=3,844$
- $62^3=238,328$
- etc.

So to convert 125 to base-62, we distribute that 125 across these base-62 “places.” The highest “place” that can take some is 62^1 , which is 62. $125/62$ is 2, with a remainder of 1. So we put a 2 in the 62’s place and a 1 in the 1’s place. So our answer is 21.

What about a higher number — say, 7,912?

Now we have enough to put something in the 3,844’s place (the 62^2 ’s place). $7,912 / 3,844$ is 2 with a remainder of 224. So we put a 2 in the 3,844’s place, and we distribute that remaining 224 across the remaining places — the 62’s place and the 1’s place. $224 / 62$ is 3 with a remainder of 38. So we put a 3 in the 62’s place and a 38 in the 1’s place. We have this three-digit number: 23- 38.

Now, that “38” represents one numeral in our base-62 number. So we need to convert that 38 into a specific choice from our set of numerals: a-z, A-Z, and 0–9.

Let’s number each of our 62 numerals, like so:

- 0: 0,
- 1: 1,
- 2: 2,
- 3: 3,
- ...
- 10: a,
- 11: b,
- 12: c,

...
36: A,
37: B,
38: C,
...
61: Z

As you can see, our “38th” numeral is “C.” So we convert that 38 to a “C.” That gives us **23C**.

So we can start with a counter (A Large number **100000000000** in base 10 which **1L9zO9O** in base 62) and increment counter every-time we get request for new short url (**100000000001**, **100000000002**, **100000000003 etc.**) .This way we will always get a unique short url.

100000000000 (Base 10) ==> 1L9zO9O (Base 62)

Similarly, when we get tiny url link for redirection we can convert this base62 tiny url to a integer in base10

1L9zO9O (Base 62) ==>100000000000 (Base 10)

Java code for generating 7 characters long tiny url with a counter

```
public class URLService {  
    HashMap<String, Integer> Itos;  
    HashMap<Integer, String> stol;  
    static int COUNTER=100000000000;  
    String elements;  
    URLService() {  
        Itos = new HashMap<String, Integer>();  
        stol = new HashMap<Integer, String>();  
        COUNTER = 100000000000;  
        elements =  
        "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"; }  
    public String longToShort(String url) {  
        String shorturl = base10ToBase62(COUNTER);  
        Itos.put(url, COUNTER);  
        stol.put(COUNTER, url);  
        COUNTER++;  
        return "http://tiny.url/" + shorturl;  
    }  
    public String shortToLong(String url) {  
        url = url.substring("http://tiny.url/".length());  
        int n = base62ToBase10(url);  
        return stol.get(n);  
    }  
    public int base62ToBase10(String s) {  
        int n = 0;
```

```

        for (int i = 0; i < s.length(); i++) {
            n = n * 62 + convert(s.charAt(i));
        }
        return n;
    }

    public int convert(char c) {
        if (c >= '0' && c <= '9')
            return c - '0';
        if (c >= 'a' && c <= 'z') {
            return c - 'a' + 10;
        }
        if (c >= 'A' && c <= 'Z') {
            return c - 'A' + 36;
        }
        return -1;
    }

    public String base10ToBase62(int n) {
        StringBuilder sb = new StringBuilder();
        while (n != 0) {
            sb.insert(0, elements.charAt(n % 62));
            n /= 62;
        }
        while (sb.length() != 7) {
            sb.insert(0, '0');
        }
        return sb.toString();
    }
}

```

Technique 3 — MD5 hash :

The **MD5 message-digest algorithm** is a widely used [hash function](#) producing a 128-bit hash value(or 32 [hexadecimal](#) digits). We can use these 32 hexadecimal digit for generating 7 characters long tiny url.

- Encode the long URL using the MD5 algorithm and take only the first 7 characters to generate TinyURL.
- The first 7 characters could be the same for different long URLs so check the DB to verify that TinyURL is not used already
- Try next 7 characters of previous choice of 7 characters already exist in DB and continue until you find a unique value

```

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
public class MD5Utils {
    private static int SHORT_URL_CHAR_SIZE=7;
    public static String convert(String longURL) {
        try {
            // Create MD5 Hash

```

```

MessageDigest digest = MessageDigest.getInstance("MD5");
digest.update(longURL.getBytes());
byte messageDigest[] = digest.digest();           // Create Hex String
StringBuilder hexString = new StringBuilder();
for (byte b : messageDigest) {
    hexString.append(Integer.toHexString(0xFF & b));
}
return hexString.toString();
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(e);
}
}

public static String generateRandomShortUrl(String longURL) {
    String hash=MD5Utils.convert(longURL);
    int numberofCharsInHash=hash.length();
    int counter=0;
    while(counter < numberofCharsInHash-SHORT_URL_CHAR_SIZE){
        if(!DB.exists(hash.substring(counter, counter+SHORT_URL_CHAR_SIZE))){
            return hash.substring(counter, counter+SHORT_URL_CHAR_SIZE);
        }
        counter++;
    }
}
}
}

```

With all these shortening algorithms, let's revisit our design goals!

1. Being able to store a *lot* of short links (**120 billion**)
2. Our TinyURL should be as short as possible (**7 characters**)
3. Application should be resilient to load spikes (For both url redirections and short link generation)
4. Following a short link should be *fast*

All the above techniques we discussed above will help us in achieving goal (1) and (2). But will fail at step (3) and step (4). Let's see how ?

1. A single web-server is a single point of failure (SPOF). If this web-server goes down, none of our users will be able to generate tiny urls/or access original (long) urls from tiny urls. This can be handled by adding more web-servers for redundancy and then bringing a load balancer in front but even with this design choice next challenge will come from database
 2. With database we have two options :
- a. Relational databases (RDBMs) like MySQL and Postgres
 - b. "NoSQL"-style databases like BigTable and Cassandra

If we choose RDBMs as our database to store the data then it is efficient to check if an URL exists in database and handle concurrent writes. But RDBMs are difficult to scale (We will use RDBMS for scaling in one of our Technique and will see how to scale using RDBMS)

If we opt for “NOSQL”, we can leverage scaling power of “NOSQL”-style databases but these systems are eventually consistent (Unlike RDBMs which provides ACID consistency)

We can use putIfAbsent(TinyURL, long URL) or INSERT-IF-NOT-EXIST condition while inserting the tiny URL but this requires support from DB which is available in RDBMS but not in NoSQL. Data is eventually consistent in NoSQL so putIfAbsent feature support might not be available in the NoSQL database(We can handle this using some of the features of NOSQL which we have discussed later on in this article). This can cause consistency issue (Two different long URLs having the same tiny url).

Scaling Technique 1 — Short url from random numbers

We can use a relational database as our backend store but since we don't have joins between objects and we require huge storage size (**60TB**) along with high writes (**40 URL's per seconds**) and read (**8000/s**) speed, a NoSQL store (MongoDB, Cassandra etc.) will be better choice.

We can certainly scale using SQL (By using custom partitioning and replication which is by default available in MongoDB and Cassandra) but this difficult to develop and maintain

Let's discuss how to use MongoDB to scale database for shortening algorithm given in Technique — 1

MongoDB supports distributing data across multiple machines using shards. Since we have to support large data sets and high throughput, we can leverage sharding feature of MongoDB.

Once we generate 7 characters long tiny url, we can use this tinyURL as shard key and employ sharding strategy as hashed sharding. MongoDB automatically computes the hashes (Using tiny url) when resolving queries. Applications do **not** need to compute hashes. Data distribution based on hashed values facilitates more even data distribution, especially in data sets where the shard key changes monotonically.

We can scale reads/writes by increasing number of shards for our collection containing tinyURL data(Schema having three fields **Short Url,Original Url,UserId**).

Schema for collection tiny URL :

```
{  
  _id: <ObjectId102>,  
  shortUrl: "https://tinyurl.com/3sh2ps6v",  
  originalUrl: "https://medium.com/@sandeep4.verma",  
  userId: "sandeepv",  
}
```

Since MongoDB supports transaction for a single document, we can maintain consistency and because we are using hash based shard key, we can efficiently use putIfAbsent (As a hash will always be corresponding to a shard)

To speed up reads (checking whether a Short URL exists in DB or what is Original url corresponding to a short URL) we can create indexing on ShortURL.

We will also use cache to further speed up reads (We will discuss caching in later part of this article)

Scaling Technique 2 — Short urls from base conversion

We used a counter (A large number) and then converted it into a base62 **7 character tinyURL**. As counters always get incremented so we can get a new value for every new request (Thus we don't need to worry about getting same tinyURL for different long/original urls)

Scaling with SQL sharding and auto increment

Sharding is a scale-out approach in which database tables are partitioned, and each partition is put on a separate RDBMS server. For SQL, this means each node has its own separate SQL RDBMS managing its own separate set of data partitions. This data separation allows the application to distribute queries across multiple servers simultaneously, creating parallelism and thus increasing the scale of that workload. However, this data and server separation also creates challenges, including sharding key choice, schema design, and application rewrites. Additional challenges of sharding MySQL include data maintenance, infrastructure maintenance, and business challenges.

Before an RDBMS can be sharded, several design decisions must be made. Each of these is critical to both the performance of the sharded array, as well as the flexibility of the implementation going forward. These design decisions include the following:

- Sharding key must be chosen
- Schema changes
- Mapping between sharding key, shards (databases), and physical servers

We can use sharding key as auto-incrementing counter and divide them into ranges for example from 1 to 10M, server 2 ranges from 10M to 20M, and so on.

We can start the counter from **100000000000**. So counter for each SQL database instance will be in range **100000000000+1 to 100000000000+10M , 100000000000+10M to 100000000000+20M** and so on.

We can start with 100 database instances and as and when any instance reaches maximum limit (10M), we can stop saving objects there and spin up a new server instance. In case one instance is not available/or down or when we require high throughput for write we can spawn multiple new server instances.

Schema for collection tiny URL For RDBMS :

```
CREATE TABLE tinyUrl (
    id BIGINT          NOT NULL, AUTO_INCREMENT
    shortUrl VARCHAR(7)    NOT NULL,
    originalUrl VARCHAR(400) NOT NULL,
```

```

userId VARCHAR(50)    NOT NULL,
automatically on primary-key column
    -- INDEX (shortUrl)
    -- INDEX (originalUrl)
);

```

Where to keep which information about active database instances ?

Solution: We can use a distributed service [Zookeeper](#) to solve the various challenges of a distributed system like a race condition, deadlock, or particle failure of data. Zookeeper is basically a distributed coordination service that manages a large set of hosts. It keeps track of all the things such as the naming of the servers, active database servers, dead servers, configuration information (*Which server is managing which range of counters*) of all the hosts. It provides coordination and maintains the synchronization between the multiple servers.

Let's discuss how to maintain a counter for distributed hosts using Zookeeper.

- From 3500 Billion URLs combinations take 1st billion combinations.
- In Zookeeper maintain the range and divide the 1st billion into 100 ranges of 10 million each i.e. range 1->(1-1,000,0000), range 2->(1,000,0001-2,000,0000).... range 1000->(999,000,0001-1,000,000,0000) (Add **100000000000** to each range for counter)
- When servers will be added these servers will ask for the unused range from Zookeepers. Suppose the W1 server is assigned range 1, now W1 will generate the tiny URL incrementing the counter and using the encoding technique. Every time it will be a unique number so there is no possibility of collision and also there is no need to keep checking the DB to ensure that if the URL already exists or not. We can directly insert the mapping of a long URL and short URL into the DB.
- In the worst case, if one of the servers goes down then only that range of data is affected. We can replicate data of master to its slave and while we try to bring master back, we can divert read queries to its slaves
- If one of the database reaches its maximum range or limit then we can move that database instance out from active database instances which can accept write and add a new database with a new a new fresh range and add this to Zookeeper. This will only be used for reading purpose
- The Addition of a new database is also easy. Zookeeper will assign an unused counter range to this new database.
- We will take the 2nd billion when the 1st billion is exhausted to continue the process.

How to check whether a short URL is present in the database or not ?

Solution : When we get a tiny url (For example 1L9zO9O) we can use the base62ToBase10 function to get the counter value (100000000000). Once we have this values we can get which database this counter ranges belongs to from zookeeper(Let's say database instance 1) . Then we can send SQL query to this server (Select * from tinyUrl where id=100000000000111).This will provide us sql row data (*if present)

How to get the original url corresponding to tiny url ?

Solution : We can leverage the above solution to get back the sql row data. This data will have shortUrl, originalURL and userId.

Scaling Technique 3— MD5 hash

We can leverage the scaling Technique 1 (Using MongoDB). We can also use Cassandra in place of MongoDB. In Cassandra instead of using shard key we will use partition key to distribute our data.

Technique 4 — Key Generation Service (KGS)

We can have a standalone **Key Generation Service (KGS)** that generates random seven-letter strings beforehand and stores them in a database (let's call it key-DB). Whenever we want to shorten a URL, we will take one of the already-generated keys and use it. This approach will make things quite simple and fast. Not only are we not encoding the URL, but we won't have to worry about duplications or collisions. KGS will make sure all the keys inserted into key-DB are unique

Can concurrency cause problems? As soon as a key is used, it should be marked in the database to ensure that it is not used again. If there are multiple servers reading keys concurrently, we might get a scenario where two or more servers try to read the same key from the database. How can we solve this concurrency problem?

Servers can use KGS to read/mark keys in the database. KGS can use two tables to store keys: one for keys that are not used yet, and one for all the used keys. As soon as KGS gives keys to one of the servers, it can move them to the used keys table. KGS can always keep some keys in memory to quickly provide them whenever a server needs them.

For simplicity, as soon as KGS loads some keys in memory, it can move them to the used keys table. This ensures each server gets unique keys. If KGS dies before assigning all the loaded keys to some server, we will be wasting those keys—which could be acceptable, given the huge number of keys we have.

KGS also has to make sure not to give the same key to multiple servers. For that, it must synchronize (or get a lock on) the data structure holding the keys before removing keys from it and giving them to a server.

Isn't KGS a single point of failure? Yes, it is. To solve this, we can have a standby replica of KGS. Whenever the primary server dies, the standby server can take over to generate and provide keys.

Can each app server cache some keys from key-DB? Yes, this can surely speed things up. Although, in this case, if the application server dies before consuming all the keys, we will end up losing those keys. This can be acceptable since we have 68B unique six-letter keys.

How would we perform a key lookup? We can look up the key in our database to get the full URL. If its present in the DB, issue an “HTTP 302 Redirect” status back to the browser, passing the stored URL in the “Location” field of the request. If that key is not present in our system, issue an “HTTP 404 Not Found” status or redirect the user back to the homepage.

Should we impose size limits on custom aliases? Our service supports custom aliases. Users can pick any ‘key’ they like, but providing a custom alias is not mandatory. However, it is reasonable (and often desirable) to impose a size limit on a custom alias to ensure we have a consistent URL database. Let's assume users can specify a maximum of 16 characters per customer key.

Cache

We can cache URLs that are frequently accessed. We can use some off-the-shelf solution like Memcached, which can store full URLs with their respective hashes. Before hitting backend storage, the application servers can quickly check if the cache has the desired URL.

How much cache memory should we have? We can start with 20% of daily traffic and, based on clients' usage patterns, we can adjust how many cache servers we need. As estimated above, we need 70GB memory to cache 20% of daily traffic. Since a modern-day server can have 256GB memory, we can easily fit all the cache into one machine. Alternatively, we can use a couple of smaller servers to store all these hot URLs.

Which cache eviction policy would best fit our needs? When the cache is full, and we want to replace a link with a newer/hotter URL, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system. Under this policy, we discard the least recently used URL first. We can use a Linked Hash Map or a similar data structure to store our URLs and Hashes, which will also keep track of the URLs that have been accessed recently.

To further increase the efficiency, we can replicate our caching servers to distribute the load between them.

How can each cache replica be updated? Whenever there is a cache miss, our servers would be hitting a backend database. Whenever this happens, we can update the cache and pass the new entry to all the cache replicas. Each replica can update its cache by adding the new entry. If a replica already has that entry, it can simply ignore it.

Load Balancer (LB)

We can add a Load balancing layer at three places in our system:

1. Between Clients and Application servers
2. Between Application Servers and database servers
3. Between Application Servers and Cache servers

Initially, we could use a simple Round Robin approach that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is that if a server is dead, LB will take it out of the rotation and will stop sending any traffic to it.

A problem with Round Robin LB is that we don't take the server load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load and adjusts traffic based on that.

Customer provided tiny URLs

A customer can also provide a tiny url of his/her choice. We can allow customers to choose just alphanumerics and the “special characters” \$-_.+!*’(),. in tiny url (With let's say 8 minimum characters). When a customer provides a custom url we can save it to some other instance of the database (Different one from which the system generates a tiny url against the original url) and treat these tiny URLs as special URLs. When we get a redirection request we can divert these request to special instances of WebServer . Since this will be a paid service we can expect very few tiny URLs to be of this kind and we don't need to worry about scaling WebServers/Database in this case.

Analytics

How many times a short URL has been used ? How would we store these statistics? Since we used “HTTP 302 Redirect” status to the browser instead of “HTTP 301 Redirect”, thus each redirection for tiny url will reach service's backend. We can push this data(tiny url, users etc.) to a Kafka queue and perform analytics in real time

Purging or DB cleanup

Should entries stick around forever, or should they be purged? If a user-specified expiration time is reached, what should happen to the link?

If we chose to continuously search for expired links to remove them, it would put a lot of pressure on our database. Instead, we can slowly remove expired links and do a lazy cleanup. Our service will ensure that only expired links will be deleted, although some expired links can live longer but will never be returned to users.

- Whenever a user tries to access an expired link, we can delete the link and return an error to the user.
- A separate Cleanup service can run periodically to remove expired links from our storage and cache. This service should be very lightweight and scheduled to run only when the user traffic is expected to be low.
- We can have a default expiration time for each link (e.g., two years).
- After removing an expired link, we can put the key back in the key-DB to be reused.
- Should we remove links that haven't been visited in some length of time, say six months? This could be tricky. Since storage is getting cheap, we can decide to keep links forever.

Security and Permissions

Can users create private URLs or allow a particular set of users to access a URL?

We can store the permission level (public/private) with each URL in the database. We can also create a separate table to store UserIDs that have permission to see a specific URL. If a user does not have permission and tries to access a URL, we can send an error (HTTP 401) back. Given that we are storing our data in a NoSQL wide-column database like Cassandra, the key for the table storing permissions would be the 'Hash' (or the KGS generated 'key'). The columns will store the UserIDs of those users that have permission to see the URL.

2. Designing Youtube or Netflix

Design a video streaming service like Youtube/Netflix where users can upload/view/search videos. The service should be scalable where a large number of users can watch and share the videos simultaneously. It will be storing and transmitting petabytes and petabytes of data.

Things to discuss and analyze:

- Approach to record stats about videos e.g the total number of views, up-votes/down-votes, etc.
- Adding comments on videos in real-time.

Components:

1. **OC** – Clouds like [AWS](#), and OpenConnect act as a content delivery network.
2. **Backend-Database**
3. **Client**- Any device to use Youtube/Netflix

Solution 1:

A video streaming service, such as Netflix or YouTube, is a platform where content creators can upload videos and viewers can search and play videos. Additionally, it's also able to record video statistics, such as the number of views, number of likes/dislikes, number of minutes watched, etc.

While users pay a monthly subscription fee to enjoy the services of Netflix, YouTube is an advertisement-based service that users enjoy free of cost. Nevertheless, both are video streaming services with similar design basics. Let's see how a basic video streaming service is designed.

System Requirements

Their architecture has several components to enhance customer experience. AI-based recommendation systems, billing, popular videos and watch later are all part of such a service, but we will focus on the core features.

Core Features

Our design for a video streaming service will support the following features:

1. Content creators can upload videos.
2. Viewers can watch videos on different devices (mobile, TV, etc.).
3. Users can search videos by their titles.
4. Users can like/dislike or comment on videos.
5. The system can store likes, dislikes and number of views to display these stats to the users.

Goals Of The System

1. There should be no buffering so the viewers can have a real-time experience when watching videos.
2. The system should have low latency and high availability. Consistency is of secondary importance in this case, since it's acceptable if a newly uploaded video is not available to a user for a while.
3. Video storage should be reliable. Uploaded videos should not be lost.
4. The system should be able to scale with the increasing number of users.

How Much Capacity Should The System Support

Since the system is expected to handle a heavy traffic on a daily basis, a single server cannot support the volume of data. The system will be served by a cluster of servers. Even if one server goes down, no performance issues should be perceived at the client end.

Uploading Video

Our system will be read-heavy. We'll allocate resources in a way that can retrieve and play videos quickly for the users to enjoy a real-time experience. For each new video upload, there may be 100 views. With this assumption, the read : write ratio is 100:1.

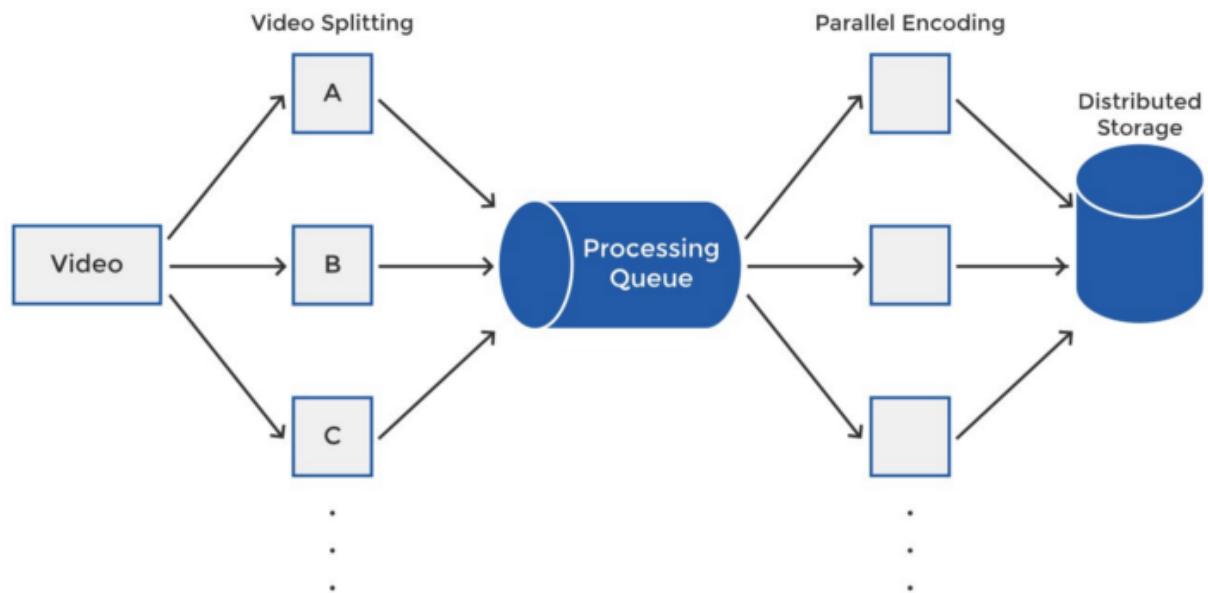
As soon as a new video is uploaded by a client, it's routed by the load balancer to the upload service. The upload service is simply a microservice for the system that handles uploading videos.

Video Storage

The upload service stores the video in a distributed file storage, such as Amazon S3, HDFS, or Gloster FS. All the video metadata information, such as likes, dislikes, views, size, description and uploader will be stored in the metadata database. It will also carry the file path for the stored video.

How To Onboard A Video

The primary function of YouTube is to onboard a movie or a video. There are several challenges that a video streaming service caters before making a video available to its users.



Storing In Chunks

Instead of being stored as a **single large file**, each uploaded video will be **stored in several chunks**. This is necessary because a content creator can upload a large video. **Processing or streaming a single heavy file can be time-consuming**. When the video is stored and available to the viewer in **chunks**, the viewer will not need to download the entire video before playing it. It will request the **first chunk from the server**, and while that chunk is playing, the client **requests for the next chunk** so there's minimum lag between the chunks and the user can have a seamless experience while watching the video.

Consider the splitting of videos as an independent microservice handled by the server, video splitter. It will divide the video into smaller chunks and put them in the processing queue. As they are de-queued, the chunks will be encoded.

Processing Queue

A processing queue is needed because there are several chunks for each video and Netflix will use several parallel workers to process them. This is made easier by pushing them into the queue. The workers (or the encoders which we will discuss next) will pick up the tasks from the processing queue, encode them into different formats and store them in the distributed file storage.

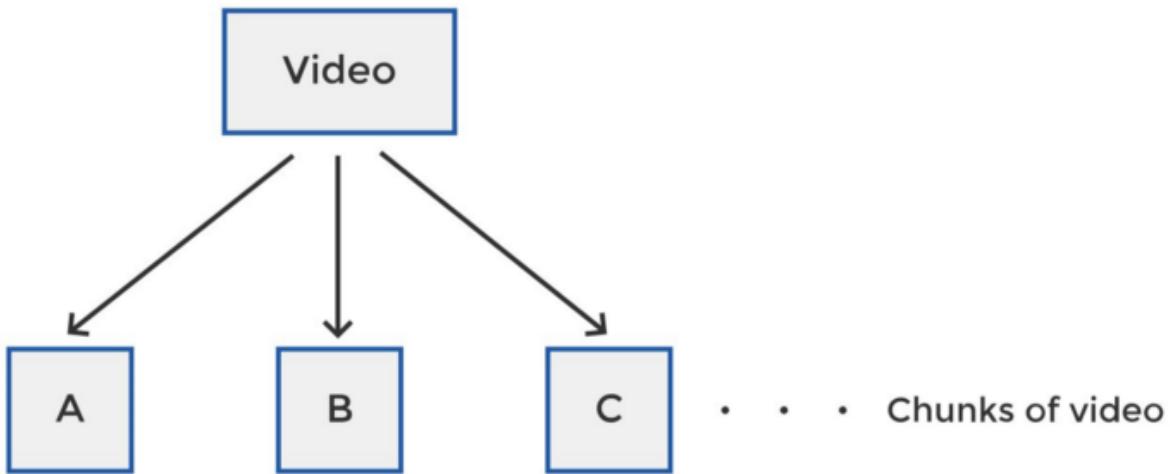
Video Encoding

An important step is to convert and store the video chunks into different formats, so different viewers can view it in the format that best suits their device and internet connection. Viewers may be watching the video through their laptop, phone, TV or some other device. Different devices have different formats that work best for them.

Similarly, different viewers may be connected to the internet through different bandwidths. Depending on the speed of their internet connection or bandwidth, some viewers may be able to stream high resolution videos easily, while those with a lower bandwidth will be able to stream low quality videos much more easily.

The original video is a high definition format that can have a size in terabytes. Netflix supports several devices and different network speeds. To make this possible, the system generates and stores over 1200 formats for each movie or show.

You will need to convert each video into different formats (mp4, avi, flv, etc), and different resolutions for each format (e.g. 1080p, 480p, 240p etc). If you have i number of formats to support and j number of resolutions, the system will generate $i*j$ number of videos for each video uploaded to the platform.



Set 1: A. mp4.1080p , B. mp4.1080p , C. mp4.1080p . . .

Set 2: A. mp4.480p , B. mp4.480p , C. mp4.480p . . .

.

.

- Sets in different formats and resolutions

For each of the video chunks, A, B, C, and so on, the video encoder will first convert them to, say, .mp4.1080p to store the first set of video chunks for the video. For the same video, another set of video chunks will be generated in .mp4.480p format and so on. The task for this microservice will complete once the sets for all the supported formats are generated.

An entry for each of these sets of chunks will be made into the metadata database and a path will be provided for the distributed storage where the actual files are saved.

Once the video splitting and encoding is complete, the processing of the video file is complete. The client who uploaded the video will be notified and the video will be available on the platform for sharing and viewing.

Open Connect For Improved User Experience

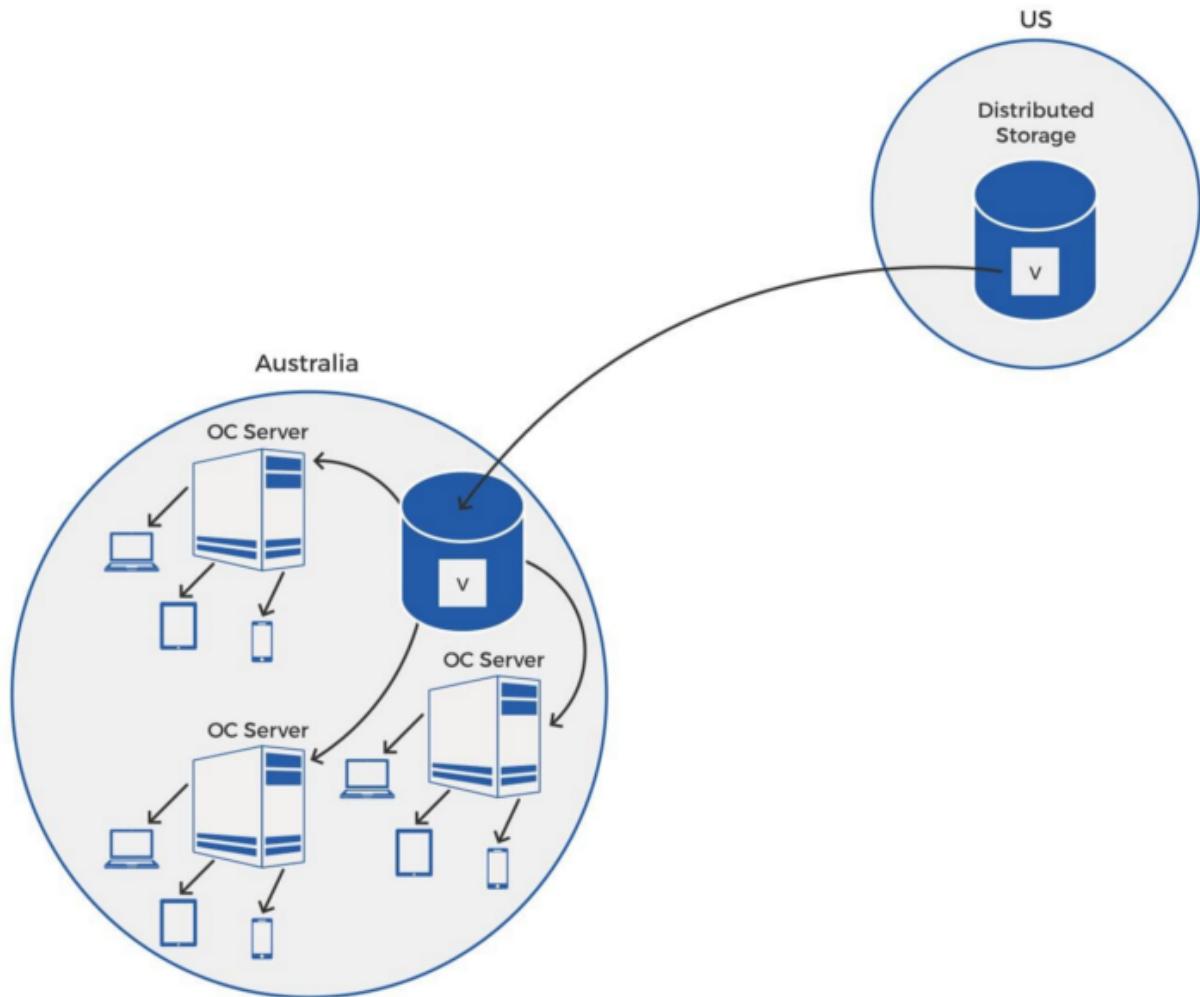
When you request for Netflix.com on your browser, you are actually asking the ISP (Internet Service Provider) to connect you to the Netflix server. The ISP contacts the Netflix IP address for you and returns you the response. However, these servers are concentrated in the US, so for the audience in a far-off country such as Tokyo, there will be a lot of delay in sending and receiving signals. Delays are even more of a problem with videos since a large amount of data needs to reach the viewer and if it's slow, the streaming will be slow and the user experience will be poor.

Netflix uses an intelligent solution to overcome the problem. It is called Open Connect (OC). Open Connect is Netflix's customized **CDN** (Content Delivery Network). CDN is a network of

distributed servers and their data centers to cache web content and deliver it quickly to the users by decreasing the physical distance between the user and the content.

Netflix's Open Connect has partnered with many ISPs across the globe to cache popular Netflix videos so that they may be delivered to the nearby viewers locally, eliminating the need for connection with Netflix's US-based servers. The Netflix videos are available in the data centers of local ISPs through Open Connect.

How Does A Newly Uploaded Video Reach The Viewer?



So once the video is processed (split and encoded) and ready for viewing, it is stored in Amazon S3. It is also pushed to all the Open Connect servers in the world. If a new video is uploaded on Netflix and is to be made available to Australian audiences, it will need to reach the continent via undersea cables.

Instead of directing all the Netflix traffic through this expensive route, Netflix copies the video file (denoted by V in the diagram above) from US-based storage to a storage location in Australia **once** during off-peak hours. Once the video has reached the continent, it's copied to all the Open Connect servers present in the ISP networks.

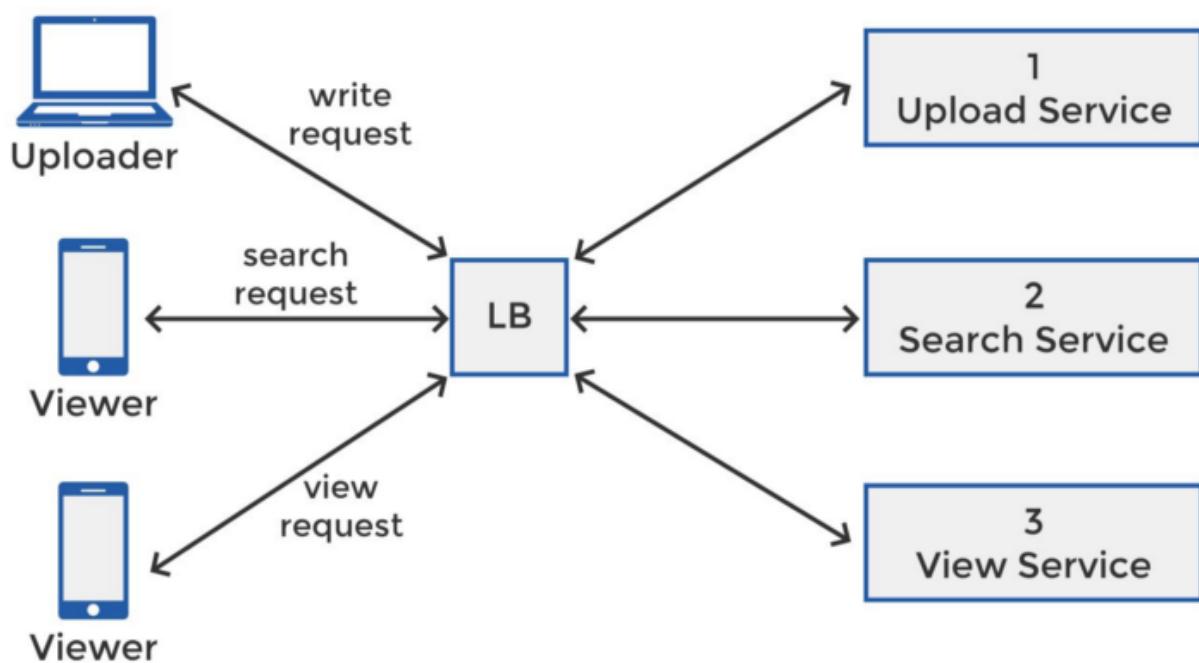
When the viewer presses the play button, the video is streamed from the nearest Open Connect Appliance (OCA) installed at the local ISP and displayed on the viewer's device.

Load Balancing

Since there are several requests (uploading, search, and viewing requests) coming in every second, a single application server cannot have the capacity to serve all the requests. Since multiple servers are involved, there needs to be a load balancer in place that can redirect requests to suitable servers and efficiently balance load among the servers.

Netflix uses consistent hashing to balance loads among servers since it can effectively manage any failure of servers and incorporate the addition of new servers. Since each video has a different popularity, it may result in an uneven load on the physical servers that handle these videos. To resolve this issue, we can use dynamic HTTP redirections that enable a busy server to redirect a new request to an available server.

YouTube/Netflix Architecture — System Design Diagram



Now that we have discussed the individual components, let's put them all together to create a complete system, as shown in the system design diagram above.

There are three basic types of requests that can be made to a video streaming application:

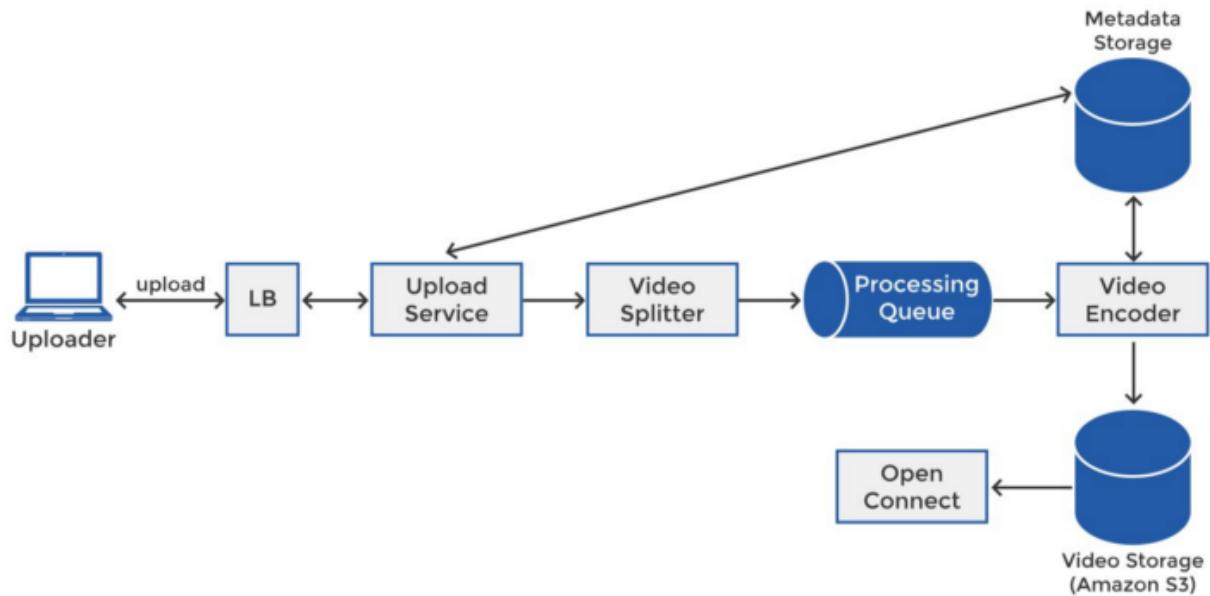
- Upload (write)
- Search (read)
- View (read)

Each of these is handled by an independent cluster of servers, keeping in mind that read requests (search and view) will be several times greater in number than write (upload).

requests. Since it's a read-heavy application, you will need to allocate more resources (servers) to serve read requests than uploads.

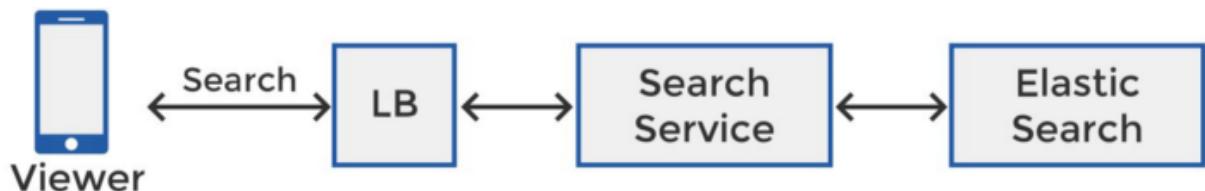
Each request from a client reaches the load balancer, from where it is redirected to the appropriate microservice.

- **Upload Service**



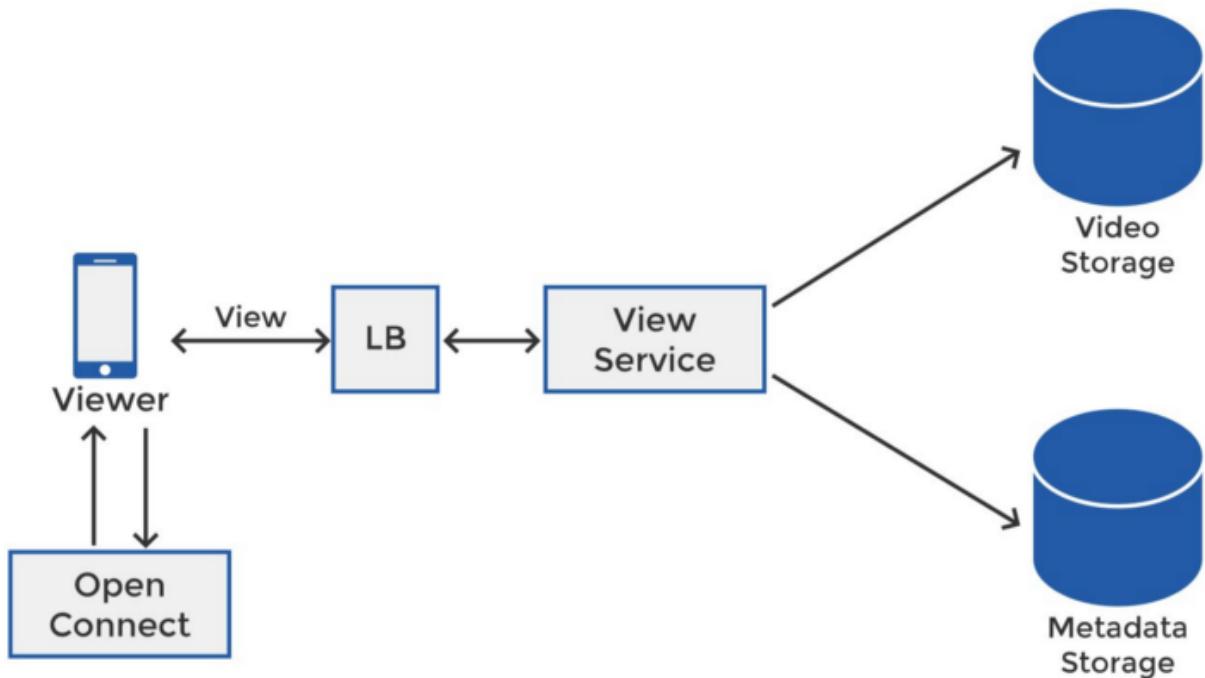
An upload request is served by the upload service that processes the video, uploads it to Open Connect servers and makes it available to all the users.

- **Search Service**



Search request is forwarded by the load balancer to the search microservice and onwards to Netflix's Elastic search. The response of the Elastic search is returned to the client. Elastic search is a highly scalable full-text open-source search engine that handles searching for millions of videos for Netflix. Netflix also relies on **Elastic search** for analyzing customer services operations.

- **View Service**

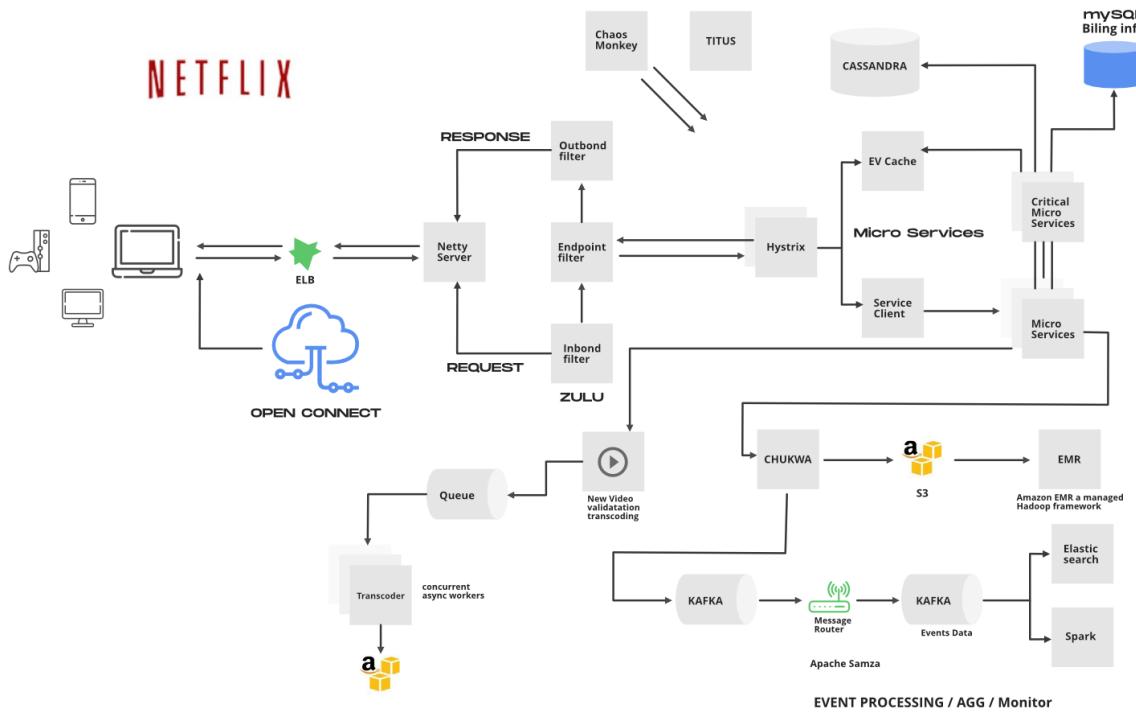


Most of the view requests will not be directed to the load balancer and Netflix's servers. Instead, they'll reach the local ISPs and will be served from the nearest Open Connect server directly. If the requested video isn't available, however, it will be forwarded to the load balancer and the view microservice. From there, the video is searched in the metadata database and fetched from the path stored in the metadata and sent to the client. Of course, this approach involves delays, which is why any view request is almost always served via Open Connect.

Solution 2:

Netflix High-Level System Architecture

We all are familiar with Netflix services. It handles large categories of movies and television content and users pay the monthly rent to access these contents. Netflix has **180M+** subscribers in **200+** countries.



Netflix works on two clouds...**AWS** and **Open Connect**. These two clouds work together as the backbone of Netflix and both are highly responsible for providing the best video to the subscribers.

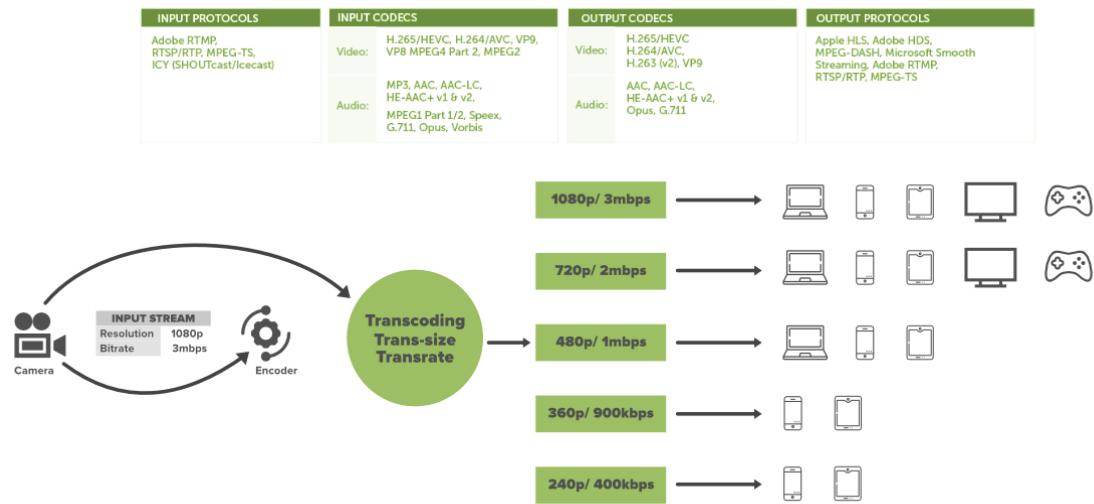
The application has mainly 3 components...

- **Client:** Device (User Interface) which is used to browse and play Netflix videos. TV, XBOX, laptop or mobile phone, etc
- **OC (Open connect) or Netflix CDN:** CDN is the network of distributed servers in different geographical locations, and Open Connect is Netflix's own custom global CDN (Content delivery network). It handles everything which involves video streaming. It is distributed in different locations and once you hit the play button the video stream from this component is displayed on your device. So if you're trying to play the video sitting in North America, the video will be served from the nearest open connect (or server) instead of the original server (faster response from the nearest server).
- **Backend (Database):** This part handles everything that doesn't involve video streaming (before you hit the play button) such as onboarding new content, processing videos, distributing them to servers located in different parts of the world, and managing the network traffic. Most of the processes are taken care of by Amazon Web Services.

Netflix frontend is written in **ReactJS** for mainly three reasons...startup speed, runtime performance, and modularity. Let's discuss the components and working of Netflix.

How does Netflix Onboard a Movie/Video?

Netflix receives very high-quality videos and content from the production houses, so before serving the videos to the users it does some preprocessing. Netflix supports more than 2200 devices and each one of them requires different resolutions and formats. To make the videos viewable on different devices, Netflix performs **transcoding or encoding**, which involves finding errors and converting the original video into different formats and resolutions.



Netflix also creates file optimization for different network speeds. The quality of a video is good when you're watching the video at high network speed. Netflix creates multiple replicas (approx 1100-1200) for the same movie with different resolutions. These replicas require a lot of transcoding and preprocessing. Netflix breaks the original video into different smaller chunks and using parallel workers in AWS it converts these chunks into different formats (like mp4, 3gp, etc) across different resolutions (like 4k, 1080p, and more).

After transcoding, once we have multiple copies of the files for the same movie, these files are transferred to each and every Open Connect server which is placed in different locations across the world.

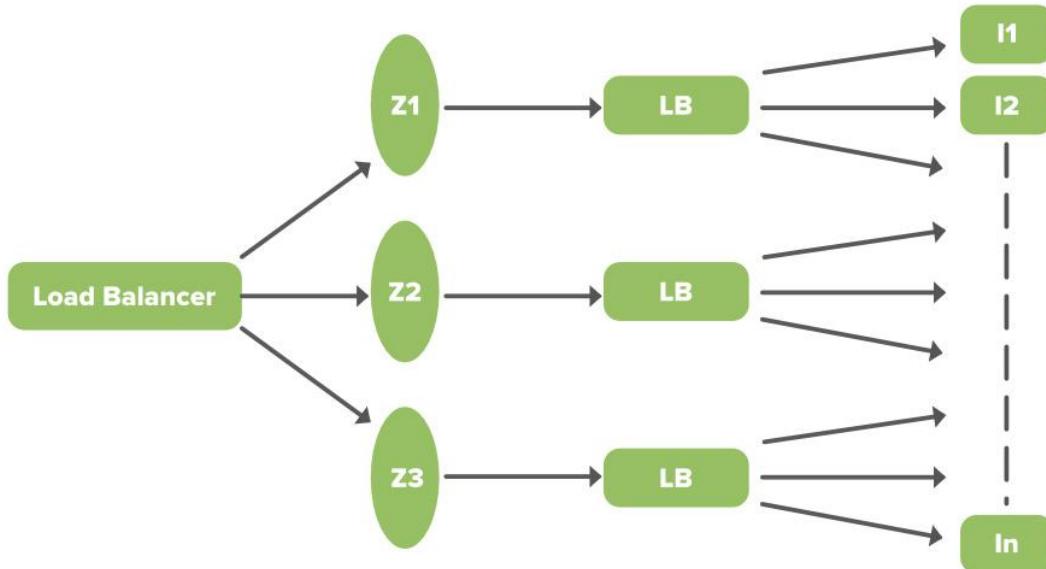
When the user loads the Netflix app on his/her device firstly AWS instances come into the picture and handle some tasks such as login, recommendations, search, user history, the home page, billing, customer support, etc. After that, when the user hits the play button on a video, Netflix analyzes the network speed or connection stability, and then it figures out the best Open Connect server near to the user. Depending on the device and screen size, the right video format is streamed into the user's device. While watching a video, you might have noticed that the video appears pixelated and snaps back to HD after a while. This happens because the application keeps checking the best streaming open connect server and switches between formats (for the best viewing experience) when it's needed.

User data is saved in AWS such as searches, viewing, location, device, reviews, and likes, Netflix uses it to build the movie recommendation for users using the Machine learning model or Hadoop.

Open Connect Advantages:

- Less Expensive
- Better Quality
- More Scalable

Elastic Load Balancer



ELB in Netflix is responsible for routing the traffic to frontend services. ELB performs a two-tier load-balancing scheme where the load is balanced over zones first and then instances (servers).

- The First-tier consists of basic DNS-based Round Robin Balancing. When the request lands on the first load balancing (see the figure), it is balanced across one of the zones (using round-robin) that your ELB is configured to use.
- The second tier is an array of load balancer instances, and it performs the Round Robin Balancing technique to distribute the request across the instances that are behind it in the same zone.

ZUUL

ZUUL is a gateway service that **provides dynamic routing, monitoring, resiliency, and security**. It provides easy routing based on query parameters, URL, and path. Let's understand the working of its different parts...

- **The Netty server** takes the responsibility to handle the network protocol, web server, connection management, and proxying work. When the request hits the Netty server, it will proxy the request to the inbound filter.
- **The inbound filter** is responsible for authentication, routing, or decorating the request. Then it forwards the request to the endpoint filter.
- **The endpoint filter** is used to return a static response or to forward the request to the backend service (or origin as we call it). Once it receives the response from the backend service, it sends the request to the outbound filter.

- An **outbound filter** is used for zipping the content, calculating the metrics, or adding/removing custom headers. After that, the response is sent back to the Netty server and then it is received by the client.

Advantages:

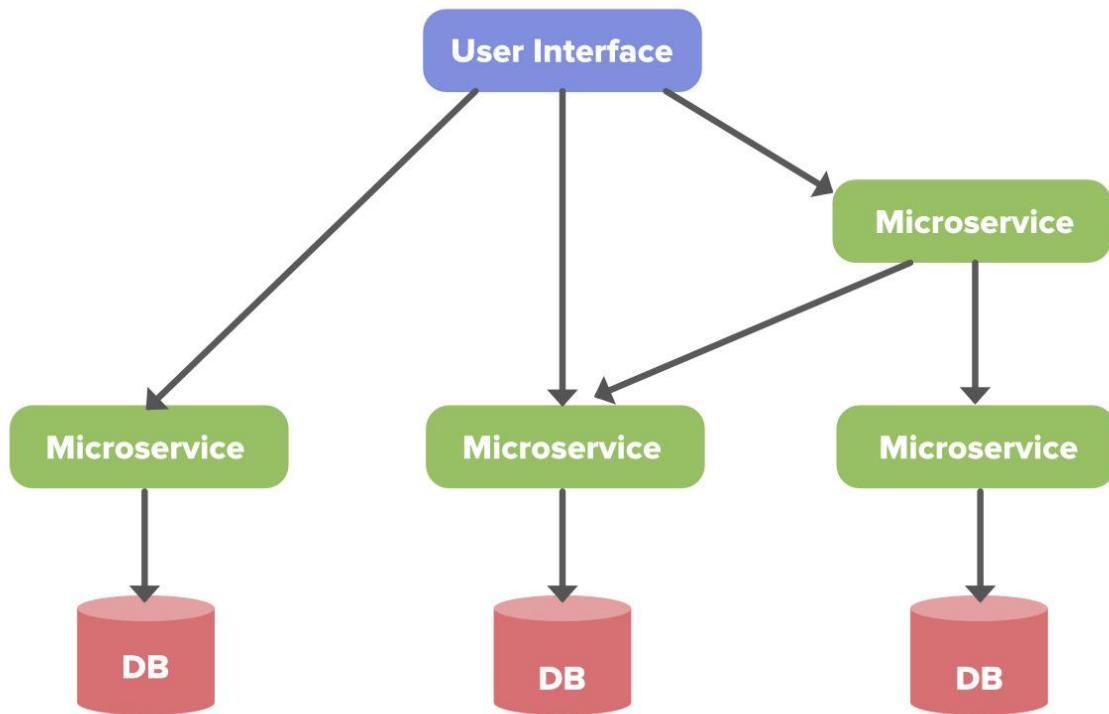
- You can create some rules and **share the traffic** by distributing the different parts of the traffic to different servers.
- Developers can also do **load testing** on newly deployed clusters in some machines. They can route some existing traffic on these clusters and check how much load a specific server can bear.
- You can also **test new services**. When you upgrade the service and you want to check how it behaves with the real-time API requests, in that case, you can deploy the particular service on one server and you can redirect some part of the traffic to the new service to check the service in real-time.
- We can also **filter the bad request** by setting the custom rules at the endpoint filter or firewall.

Hystrix

In a complex distributed system a server may rely on the response of another server. Dependencies among these servers can create latency and the entire system may stop working if one of the servers will inevitably fail at some point. To solve this problem we can isolate the host application from these external failures. Hystrix library is designed to do this job. It helps you to control the interactions between these distributed services by adding latency tolerance and fault tolerance logic. Hystrix does this by isolating points of access between the services, remote system, and 3rd party libraries. The library helps in.

- Stop cascading failures in a complex distributed system.
- Control over latency and failure from dependencies accessed (typically over the network) via third-party client libraries.
- Fail fast and rapidly recover.
- Fallback and gracefully degrade when possible.
- Enable near real-time monitoring, alerting, and operational control.
- Concurrency-aware request caching. Automated batching through request collapsing

Microservice Architecture of Netflix



Netflix's architectural style is built as a collection of services. This is known as microservices architecture and this powers all of the APIs needed for applications and Web apps. When the request arrives at the endpoint it calls the other microservices for required data and these microservices can also request the data from different microservices. After that, a complete response for the API request is sent back to the endpoint.

In a microservice architecture, services should be independent of each other, for example, the video storage service would be decoupled from the service responsible for transcoding videos. Now, let's understand how to make it reliable...

How to make microservice architecture reliable?

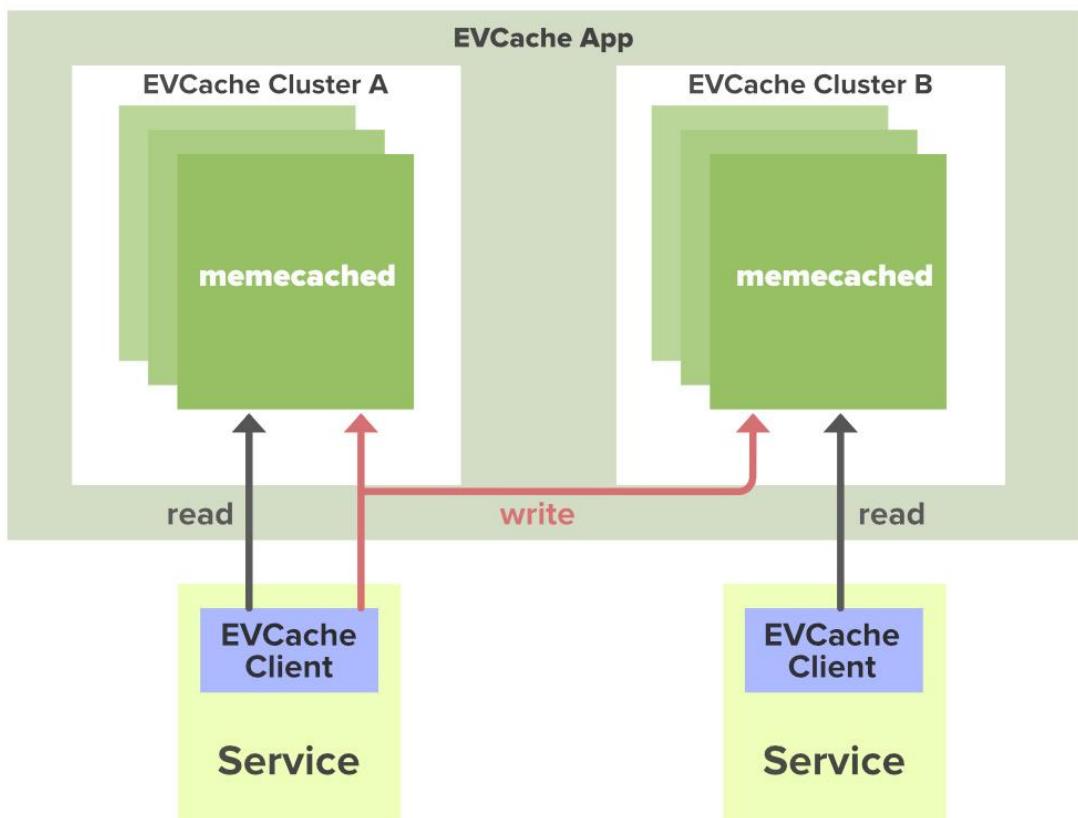
- Use Hystrix (Already explained)
- **Separate Critical Microservices:** We can separate out some critical services (or endpoint or APIs) and make it less dependent or independent of other services. You can also make some critical services dependent only on other reliable services. While choosing the critical microservices you can include all the basic functionalities, like searching a video, navigating to the videos, hitting and playing the video, etc. This way you can make the endpoints highly available and even in worst-case scenarios at least a user will be able to do the basic things.
- **Treat Servers as Stateless:** This may sound funny to you but to understand this concept think of your servers like a herd of cows and you care about how many gallons of milk you get every day. If one day you notice that you're getting less milk from a cow then you just need to replace that cow (producing less milk) with another cow. You don't need to be dependent on a specific cow to get the required amount of

milk.

We can relate the above example with our application. The idea is to design the service in such a way that if one of the endpoints is giving the error or if it's not serving the request in a timely fashion then you can switch to another server and get your work done. Instead of relying on a specific server and preserving the state in that server, you can route the request to another service instance and you can automatically spin up a new node to replace it. If a server stops working, it will be replaced by another one.

EV Cache

In most applications, some amount of data is frequently used. For faster response, these data can be cached in so many endpoints and it can be fetched from the cache instead of the original server. This reduces the load from the original server but the problem is if the node goes down all the cache goes down and this can hit the performance of the application. To solve this problem Netflix has built its own custom caching layer called EV cache. EV cache is based on **Memcached** and it is actually a wrapper around Memcached.



3 Node Memcached Cluster in 2 Availability Zones With a Client in Each Zone

Netflix has deployed a lot of clusters in a number of AWS EC2 instances and these clusters have so many nodes of Memcached and they also have cache clients. The data is shared across the cluster within the same zone and multiple copies of the cache are stored in sharded nodes. Every time when write happens to the client all the nodes in all the clusters

are updated but when the read happens to the cache, it is only sent to the nearest cluster (not all the cluster and nodes) and its nodes. In case, a node is not available then read from a different available node. This approach increases performance, availability, and reliability.

Database

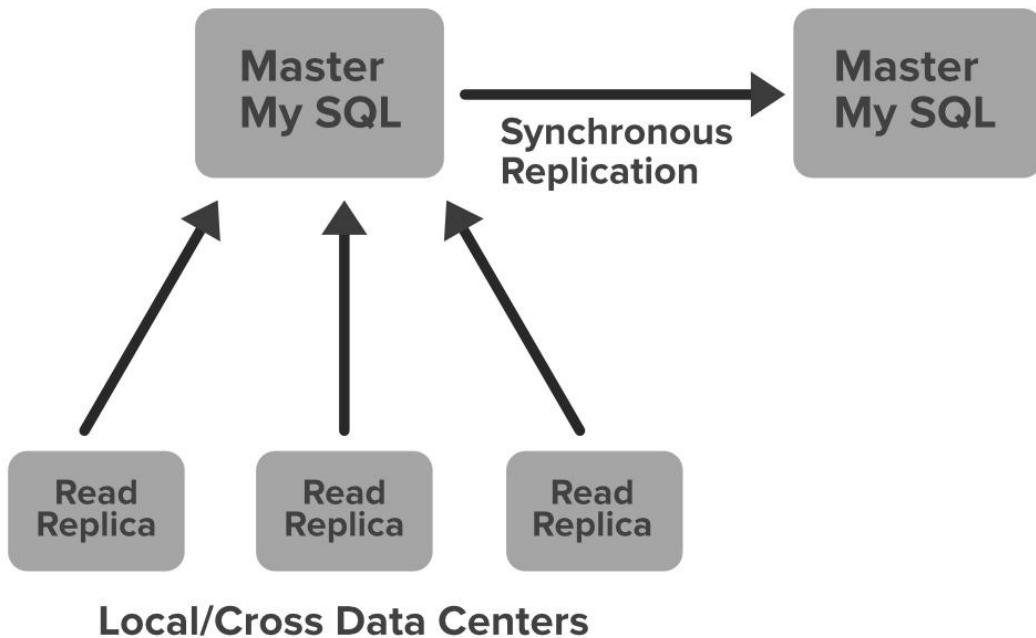
Netflix uses two different databases i.e. MySQL(RDBMS) and Cassandra(NoSQL) for different purposes.

EC2 Deployed MySQL

Netflix saves data like billing information, user information, and transaction information in MySQL because it needs ACID compliance. Netflix has a master-master setup for MySQL and it is deployed on Amazon large EC2 instances using InnoDB.

The setup follows the “**Synchronous replication protocol**” where if the writer happens to be the primary master node then it will be also replicated to another master node. The acknowledgment will be sent only if both the primary and remote master nodes’ write have been confirmed. This ensures the high availability of data.

Netflix has set up the read replica for each and every node (local, as well as cross-region). This ensures high availability and scalability.

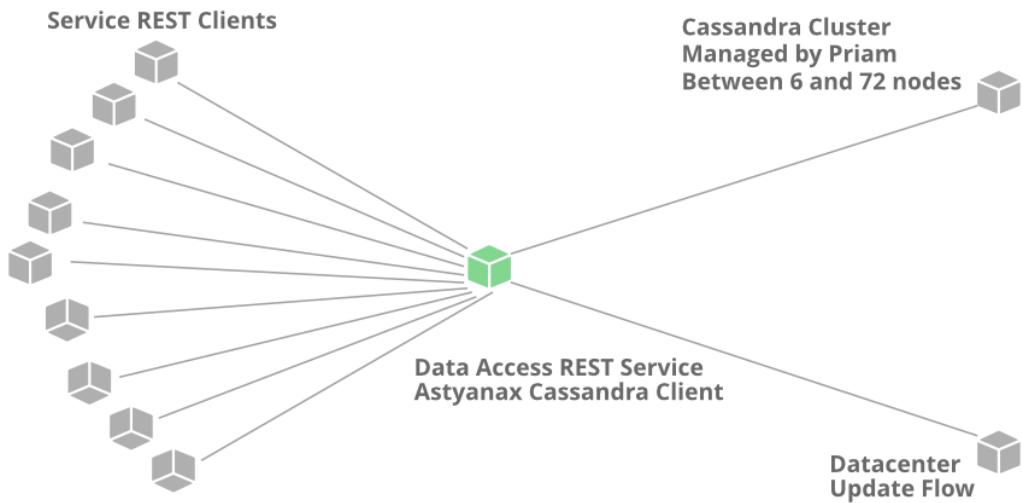


All the read queries are redirected to the read replicas and only the write queries are redirected to the master nodes. In the case of a primary master MySQL failure, the secondary master node will take over the primary role and the route53 (DNS configuration) entry for the database will be changed to this new primary node. This will also redirect the write queries to this new primary master node.

Cassandra

Cassandra is a NoSQL database that can handle large amounts of data and it can also handle heavy writing and reading. When Netflix started acquiring more users, the viewing history data for each member also started increasing. This increases the total number of viewing history data and it becomes challenging for Netflix to handle this massive amount of data. Netflix scaled the storage of viewing history data-keeping two main goals in their mind...

- Smaller Storage Footprint.
- Consistent Read/Write Performance as viewing per member grows (viewing history data write to read ratio is about 9:1 in Cassandra).



Total Denormalized Data Model

- Over 50 Cassandra Clusters
- Over 500 Nodes
- Over 30TB of daily backups
- The biggest cluster has 72 nodes.
- 1 cluster over 250K writes/s

Initially, the viewing history was stored in Cassandra in a single row. When the users started increasing on Netflix the row sizes as well as the overall data size increased. This resulted in high storage, more operational cost, and slow performance of the application. The solution to this problem was to compress the old rows...

Netflix divided the data into two parts...

- **Live Viewing History (LiveVH):** This section included the small number of recent viewing historical data of users with frequent updates. The data is frequently used for the ETL jobs and stored in uncompressed form.

- **Compressed Viewing History (CompressedVH):** A large amount of older viewing records with rare updates is categorized in this section. The data is stored in a single column per row key, also in compressed form to reduce the storage footprint.

Data Processing in Netflix Using Kafka And Apache Chukwa

When you click on a video Netflix starts processing data in various terms and it takes less than a nanosecond. Let's discuss how the evolution pipeline works on Netflix.

Netflix uses Kafka and Apache Chukwe to ingest the data which is produced in a different part of the system. Netflix provides almost **500B** data events that consume **1.3 PB/day** and 8 million events that consume 24 GB/Second during peak time. These events include information like...

- Error logs
- UI activities
- Performance events
- Video viewing activities
- Troubleshooting and diagnostic events.

Apache Chukwe is an open-source data collection system for collecting logs or events from a distributed system. It is built on top of HDFS and Map-reduce framework. It comes with Hadoop's scalability and robustness features. Also, it includes a lot of powerful and flexible toolkits to display, monitor, and analyze the result. Chukwe collects the events from different parts of the system and from Chukwe you can do monitoring, and analysis or you can use the dashboard to view the events. Chukwe writes the event in the Hadoop file sequence format (S3). After that Big Data team processes these S3 Hadoop files and writes Hive in Parquet data format. This process is called batch processing which basically scans the whole data at the hourly or daily frequency.

To upload online events to EMR/S3, Chukwa also provide traffic to Kafka (the main gate in real-time data processing). Kafka is responsible for moving data from fronting Kafka to various sinks: S3, Elasticsearch, and secondary Kafka. Routing of these messages is done using the **Apache Samza** framework. Traffic sent by the Chukwe can be full or filtered streams so sometimes you may have to apply further filtering on the Kafka streams. That is the reason we consider the router to take from one Kafka topic to a different Kafka topic.

Elastic Search

In recent years we have seen massive growth in using Elasticsearch within Netflix. Netflix is running approximately **150** clusters of elastic search and **3, 500** hosts with instances.

Netflix is using elastic search for data visualization, customer support, and for some error detection in the system. For example, if a customer is unable to play the video then the customer care executive will resolve this issue using elastic search. The playback team goes to the elastic search and searches for the user to know why the video is not playing on the user's device. They get to know all the information and events happening for that particular user. They get to know what caused the error in the video stream. Elastic search is also

used by the admin to keep track of some information. It is also used to keep track of resource usage and to detect signup or login problems.

Apache Spark For Movie Recommendation

Netflix uses Apache Spark and Machine learning for Movie recommendation. Let's understand how it works with an example. When you load the front page you see multiple rows of different kinds of movies. Netflix personalizes this data and decides what kind of rows or what kind of movies should be displayed to a specific user. This data is based on the user's historical data and preferences. Also, for that specific user, Netflix performs sorting of the movies and calculates the relevance ranking (for the recommendation) of these movies available on their platform.

In Netflix, Apache Spark is used for content recommendations and personalization. A majority of the machine learning pipelines are run on these large spark clusters. These pipelines are then used to do row selection, sorting, title relevance ranking, and artwork personalization among others.

Artwork Personalization

When you open the Netflix front page you might have noticed the images for each video...these images are called header images (thumbnail). Netflix wants maximum clicks for the videos from the users and these clicks are dependent on the header images. Netflix has to choose the right compelling header image for a specific video. To do that Netflix creates multiple artworks for a specific movie and they display these images to the users randomly. For the same movie, images can be different for different users. Based on your preferences and viewing history Netflix predicts what kind of movies you like best or which actors you like the most in a movie. According to users' tastes, the images will be displayed to them.

For example, suppose you see 9 different images for your favorite movie Good will hunting in three rows (If you like comedies then images of Robin Williams for this movie will be shown. If you like romantic movies then Netflix will show you the image of Matt Damon and Minnie Driver). Now, Netflix calculates the number of clicks a certain image receives. If the clicks for the center image of the movie are 1, 500 times and the other images have fewer clicks then Netflix will make the center image a header image for the movie Good Will Hunting forever. This is called **data-driven** and Netflix performs the data analytics with this approach. To make the right decision data is calculated based on the number of views associated with each picture.

Video Recommendation System

If a user wants to discover some content or video on Netflix, the recommendation system of Netflix helps users to find their favorite movies or videos. To build this recommendation system Netflix has to predict the user interest and it gathers different kinds of data from the users such as...

- User interaction with the service (viewing history and how user rated other titles)
- Other members with similar tastes and preferences.

- Metadata information from the previously watched videos for a user such as titles, genre, categories, actors, release year, etc.
- The device of the user, at what time a user is more active, and for how long a user is active.

Netflix uses two different algorithms to build a recommendation system

1. Collaborative filtering: The idea of this filtering is that if two users have similar rating history then they will behave similarly in the future. For example, consider there are two persons. One person liked the movie and rated the movie with a good score. Now, there is a good chance that the other person will also have a similar pattern and he/she will do the same thing that the first person has done.

2. Content-based filtering: The idea is to filter those videos which are similar to the video a user has liked before. Content-based filtering is highly dependent on the information from the products such as movie title, release year, actors, the genre. So to implement this filtering it's important to know the information describing each item and some sort of user profile describing what the user likes is also desirable.

Solution 3:



Photo by [Thibault Penin](#) on [Unsplash](#)

Audience

This article is the next in my series of how I would design popular applications. It is recommended (although not entirely necessary) to read the previous posts I've helpfully compiled in a list [here](#).

We will expect a basic familiarity with architecture principles and AWS, but hopefully, this post is approachable for most engineers.

Argument

Initially, let's look at our problem statement.

The System to Design

We are hoping to design a video on demand platform such as [YouTube](#) or [Netflix](#). It's unlikely you haven't encountered them before, but in case you haven't, the premise is that a user can upload or view videos online. The exact requirements are:

1. We should be able to upload videos.
2. We should be able to view videos.
3. We should be able to perform searches based on video titles.

We'll ignore the fact you can't upload videos to Netflix unless you're a production studio. Let's imagine you're Quentin Tarantino.

The Approach

We have a standard approach to system design which is explained more thoroughly in the article [here](#). However, the steps are summarised below:

1. **Requirements clarification:** Making sure we have all the information before starting. This may include how many requests or users we are expecting.
2. **Back of the envelope estimation:** Doing some quick calculations to gauge the necessary system performance. For example, how much storage or bandwidth do we need?
3. **System interface design:** What will our system look like from the outside, how will people interact with it? Generally this is the API contract.
4. **Data model design:** What our data will look like when we store it. At this point we could be thinking about relational vs non-relational models.
5. **Logical design:** Fitting it together in a rough system! At this point I'm thinking at a level of 'how would I explain my idea to someone who knows nothing about tech?'
6. **Physical design:** Now we start worrying about servers, programming languages and the implementation details. We can superimpose these on top of the logical design.
7. **Identify and resolve bottlenecks:** At this stage we will have a working system! We now refine the design.

With that said, let's get stuck in!

Requirements Clarification

Let's think of some initial questions. Initially, I'd be wondering what the maximum and average sizes of a video are. I'd then be thinking about the number of users and their read/write/ search ratio.

Additionally, what devices/ connection speeds are clients using? We may need to optimize our file formats and sizes based on this.

Back of the Envelope Estimation

Let's assume we have 100 million users (realistically it's probably more!), reading and writing to a daily ratio of 100:1. The average file size is 1GB for HD, and maximum file size is 50GB (doesn't matter how accurate these are).

This means we will have a rough capacity of $100,000,000 * 1\text{GB} = 100\text{PB/d}$ just for uploading! We can work this out in requests per second given an average video length, but it's safe to say we're dealing with a lot of data!

This is also a global system, so we can assume we will be using lots of different devices, from mobile to embedded devices, over a range of networks, so we will need to design defensively for this.

System Interface Design

Having learnt a little bit more about our system, we can decide how we would like to interact with it. There are three main points of interaction.

1. Upload video
2. Download video
3. Search

Uploading a video will be done over HTTP. One thing we need to be aware of is the size of the file. In a different scenario we may use [FTP](#), but given we have random users on the internet this doesn't really make sense here.

What I would recommend doing is creating an initial form to submit the metadata to do with the file (title, tags, anything that is not the video itself) which would return a GGUID associated with the file.

From there we could use the JavaScript [File](#) and [Blob](#) APIs to [cut up the video file and send partial requests](#) with a [Content-Range](#) header representing the range of bytes this request represents. Our content type would be application/octet-stream and the server is responsible for reassembling the parts.

This means we need an endpoint for creating a file, POST at endpoint /file which creates a new file object with the metadata, then another endpoint at POST file/{id}/chunk which allows a user to post a chunk.

Something we could also think about would be hashing the file on the client, then comparing the hash after we reassemble the file on the server, to ensure it matches.

Downloading can make use of an [HTTP Range Request](#). This allows us to request parts of a video at a time (returning a 206), meaning we can dynamically request video segments as we need them. Using GET file/{id}/chunk with the Content-Range header could be one solution.

Luckily searching is a little more simple, we can GET to /search?title=<search title> and receive the usual response codes.

Data Model Design

Now we have an idea for how we'd interact with our system, let's think about the type of data we want to store. Some of it will be based around video bytes, and some of it will be based around metadata.

Initially, let's think of a file. A file can have a bunch of information stored around it: the title, the created date, the user who uploaded it, the number of hits etc.

File

```
id      BIGINT  PRIMARY KEY  
title   VARCHAR
```

We now need to think about our file chunks. Remember we need to serve to lots of networks and devices! This means we will want to store chunks in lots of different qualities and file formats. With the advent of object storage like [AWS S3](#) we can reference where in S3 our file sits.

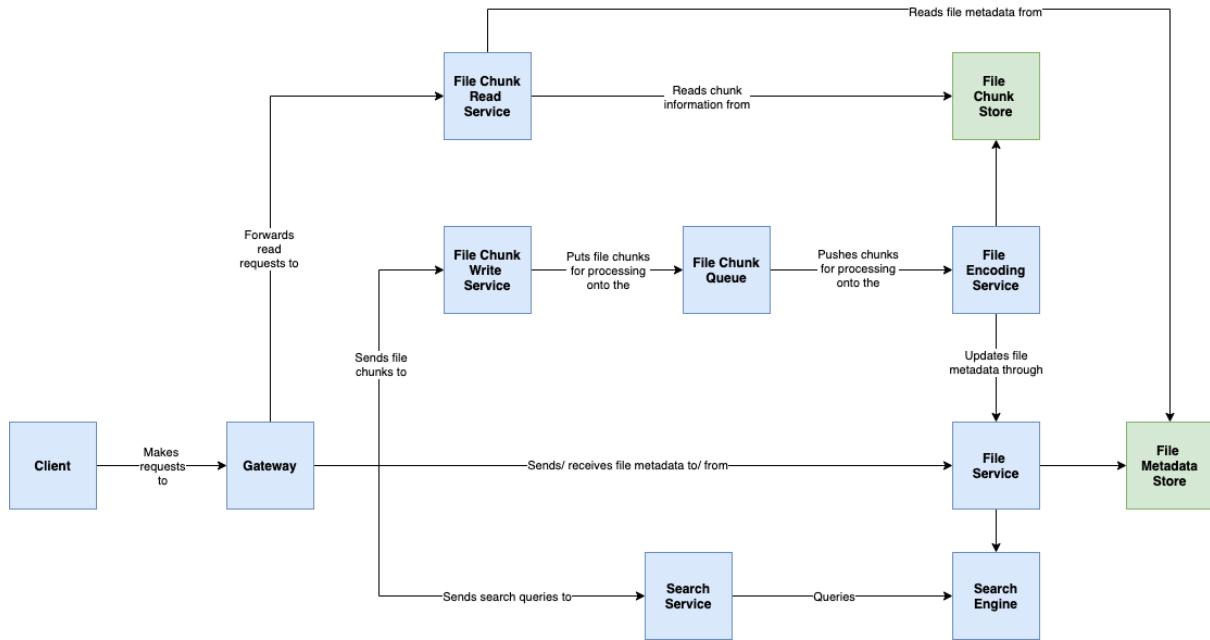
Chunk

```
id      BIGINT  PRIMARY KEY  
file_id  BIGINT  FOREIGN KEY REFERENCES file(id)  
format   VARCHAR  
quality  BIGINT  
order    BIGINT  
location  VARCHAR
```

Using the above we can identify which file a chunk belongs to, the format of the chunk, the size of a chunk and which bit of the file it represents.

This forms the core of our data model. Let's move onto the logical design!

Logical Design



A basic logical design

We can walk through the logical design to bring the various components together. When a user wants to upload a file they send a request to the file service to create the new file metadata. Their browser then breaks the file into chunks and uploads it to the file chunk write service.

This file chunking service is responsible for assembling the chunks from the client into a single file, verifying it, and splitting it down into different chunks more usable by the media player.

From here we persist onto a queue to give us a layer of resiliency in case the file encoding service is down. An acceptance message can be returned to the user now, we have the file, we just need to process it.

The file encoding service takes the chunks of the file and converts it into different formats and qualities for different services. An interesting point to note is there is a difference between encoding and transcoding.

- **Transcoding:** Creating a file in different sizes.
- **Encoding:** Re-encoding a file in a different format.

Encoding encompasses transcoding, which is why we use the term here.

This service is responsible for writing our chunks to storage, as well as updating our file metadata with their location.

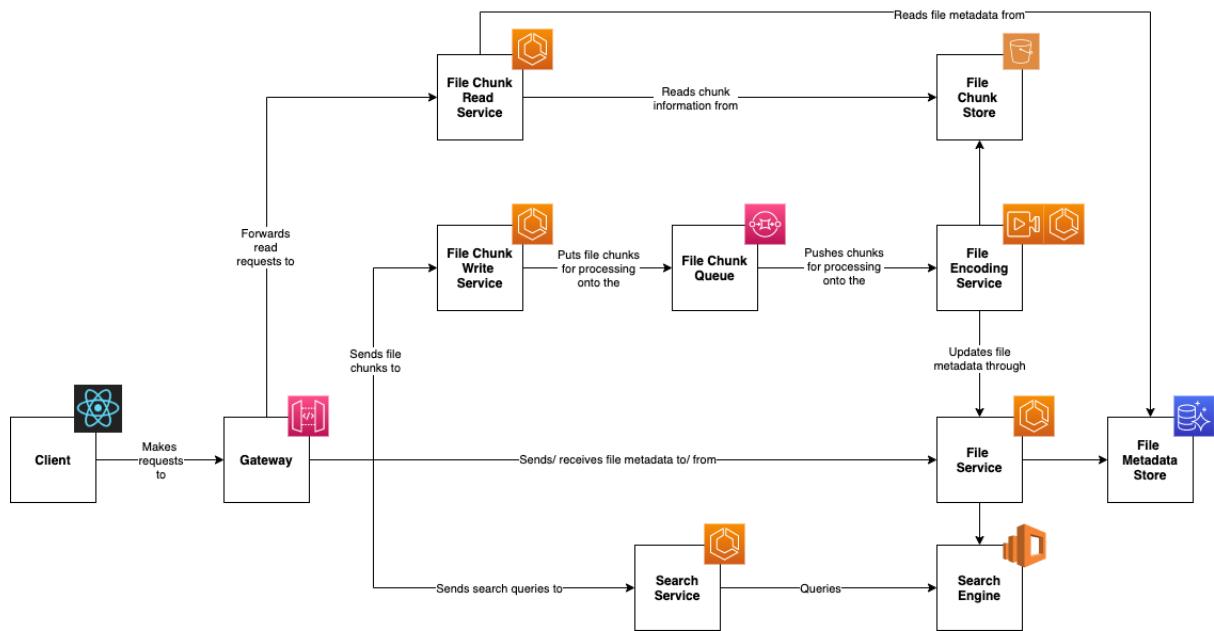
Read requests go through the API Gateway as well, but instead are redirected to either our file chunk read service if they are requesting videos to be streamed, or our file service if they are looking for metadata.

The client will be responsible for determining which chunks of the file they need and sending out the necessary requests in order to load the next part.

The search function should be comparably straight forward. The file service will have added the file metadata to the search engine, and so the search service is responsible for converting from a URL query to something we can use with the search engine, then marshalling the response.

Physical Design

Now we've covered the rough idea of how we'd put together our platform, let's flesh it out into something tangible in the real world.



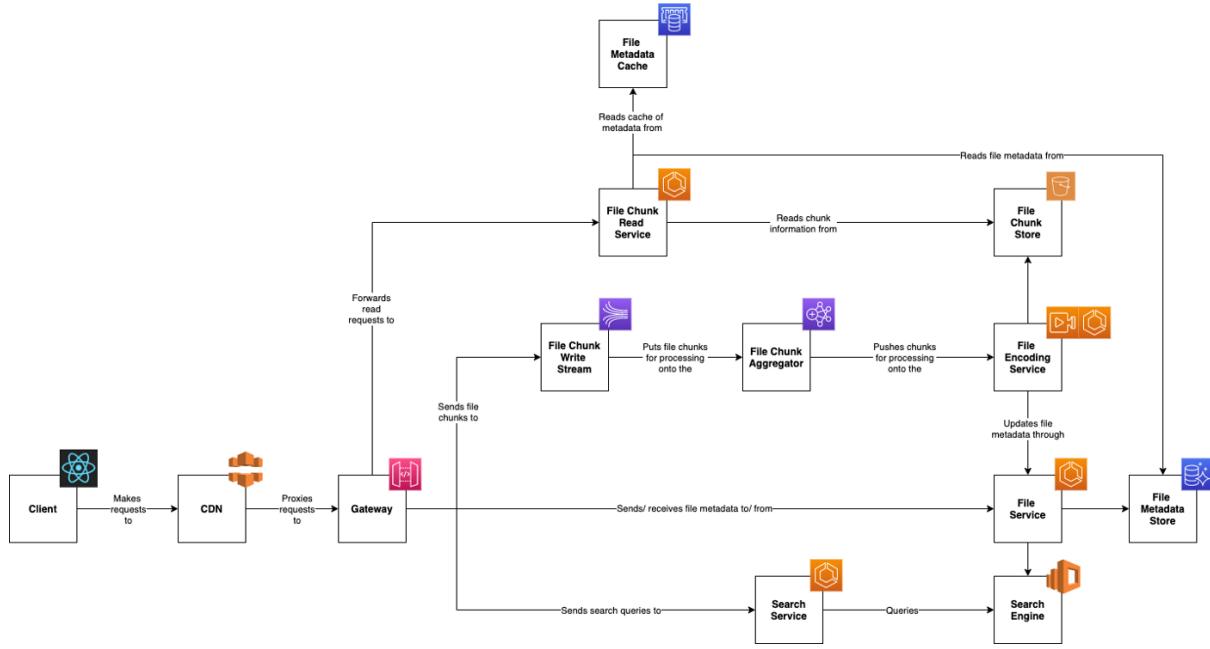
A naive physical design

Even as I was writing it I could see a number of issues with this design. However, let's explore it and we can refine it in the next section.

Initially, all of our static services use [AWS ECS](#). There's an argument we could go totally serverless, but my feeling is we don't want to worry about cold start times. Our Gateway is an [AWS API Gateway](#) and our queues use [SQS](#).

We store our chunks in [S3](#), and use [AWS Elemental MediaConvert](#) for our encoding. The search engine itself is [Amazon OpenSearch](#).

Identifying and Resolving Bottlenecks



A revised diagram

We mentioned there were some performance improvements we could make. Initially, I'm concentrating on the amount of data we're receiving. We need to have a resilient system capable of dealing with lots of information in real time. This sounds like a job for streaming.

From the diagram you can see that our API Gateway now writes to a [Kinesis stream](#), which in turn pushes to [AWS EMR](#) (probably with Spark Streaming on) to process and re-chunk our files.

As our file metadata will rarely change, we can also add a cache layer to our read service.

We could also implement [adaptive bitrate streaming](#), which is covered very well in the linked article. In our encoder we made chunks of the same file of varying quality. Using this technique we can reactively send different quality chunks depending on the current state of the streamer's connection.

The final part of the optimisation is adding a CDN. As we will have users all over the world, we don't want them all trying to access their videos from servers in the UK. It makes sense to try and distribute our content closer to where they are. Note, Netflix actually has its own solution to this problem called [Open Connect](#), which is worth a look!

Solution 4:

Youtube is one of the most popular video sharing websites in the world. Users of the service can upload, view, share, rate, and report videos as well as add comments on videos.

Requirements and Goals of the System

- For simplicity, we plan to design a simpler version of Youtube with following requirements:
- Functional Requirements:
 - Users should be able to upload videos.
 - Users should be able to share and view videos.
 - Users can perform searches based on video titles.
 - Our services should be able to record stats of videos, e.g., likes/dislikes, total number of views, etc.
 - Users should be able to add and view comments on videos.
- Non-Functional Requirements:
 - The system should be highly reliable, any video uploaded should not be lost.
 - The system should be highly available. Consistency can take a hit (in the interest of availability), if a user doesn't see a video for a while, it should be fine.
 - Users should have real time experience while watching videos and should not feel any lag.
- Not in scope:
 - Video recommendation, most popular videos, channels, and subscriptions, watch later, favorites, etc.

Capacity Estimation and Constraints

- Our service will be read heavy.
- Let's assume our **upload:view ratio is 1:200** i.e., for every video upload we have 200 video viewed.

Traffic Estimates:

- Let's assume we have **1.5 billion total users**, of them there are **800 million daily active users**.
- If, on the **avg a user views 5 videos per day**, then **Total video-views per second**: $800M * 5 / 86400 \text{ sec} \Rightarrow 46K \text{ videos/sec}$.
- **Total videos uploaded per second** would be $46K / 200 \Rightarrow 230 \text{ videos/sec}$.

Storage Estimates:

- Let's assume that **every minute 500 hours worth of videos are uploaded to Youtube**.
- If on **avg one minute of video needs 50MB of storage** (videos need to be stored in multiple formats), **total storage needed for videos uploaded in a minute** would be: $500 \text{ hours} * 60 \text{ min} * 50\text{MB} \Rightarrow 1500 \text{ GB/min (25 GB/sec)}$
- These numbers are estimated, ignoring video compression and replication, which would change our estimates.

Bandwidth estimates:

- With 500 hours of video uploads per minute, assuming each video upload takes a bandwidth of 10MB/min, we would be getting 300GB of uploads every minute.
Incoming data per second: 500 hours * 60 mins * 10MB => 300GB/min (**5GB/sec**)
- Assuming an upload:view ratio of 1:200, **Outgoing data per second: 1TB/s** .

System APIs

Upload Video API

```
uploadVideo(api_dev_key,      video_title,      vide_description,      tags[],      category_id,
default_language, recording_details, video_contents)
```

- Parameters:
 - api_dev_key (string):** The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.
 - video_title (string):** Title of the video.
 - video_description (string):** Optional description of the video.
 - tags (string[]):** Optional tags for the video.
 - category_id (string):** Category of the video, e.g., Film, Song, People, etc.
 - default_language (string):** For example English, Mandarin, Hindi, etc.
 - recording_details (string):** Location where the video was recorded.
 - video_contents (stream):** Video to be uploaded.
- Returns: (string)
 - A successful upload will return HTTP 202 (request accepted).
 - Once the video encoding is completed, the user is notified through email with a link to access the video.
 - We can also expose a queryable API to let users know the current status of their uploaded video.

Search Video API

```
searchVideo(api_dev_key,      search_query,      user_location,      maximum_videos_to_return,
page_token)
```

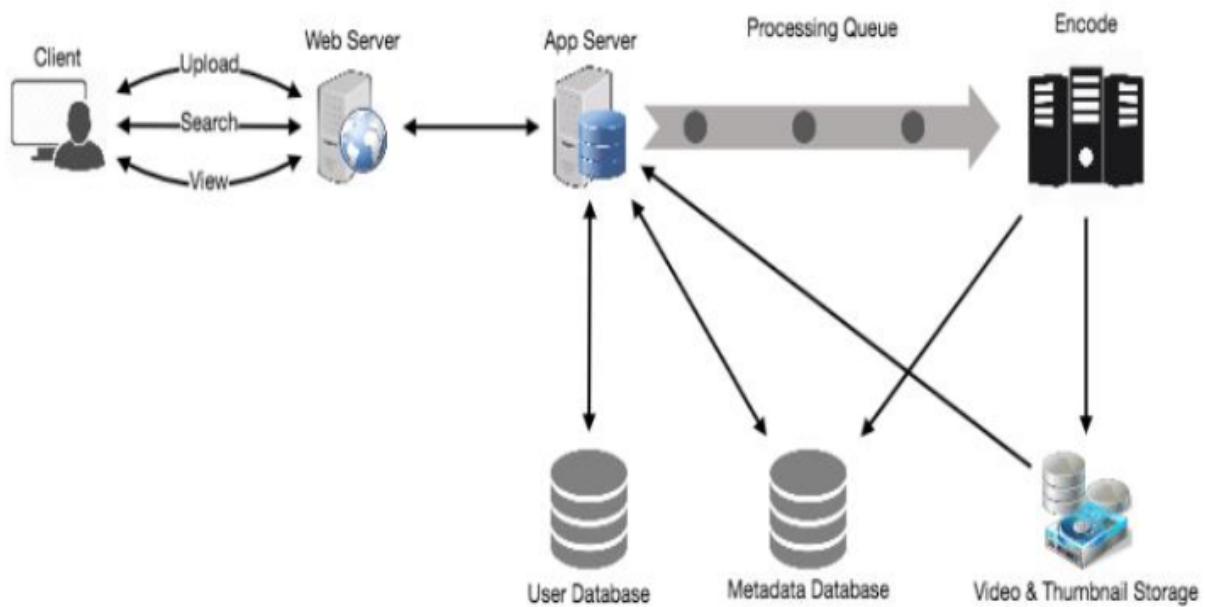
- Parameters:
 - api_dev_key (string):** The API developer key of a registered account of our service.
 - search_query (string):** A string containing the search terms.
 - user_location (string):** Optional location of the user performing the search.
 - maximum_videos_to_return (number):** Maximum number of results returned in one request.

- **page_token (string)**: This token will specify a page in the result set that should be returned.
- Returns: (JSON)
 - A JSON containing information about the list of video resources matching the search query.
 - Each video resource will have a video title, a thumbnail, a video creation date and how many views it has.

High Level Design

At a high-level we would need following components:

1. Processing Queue: Each uploaded video will be pushed to a processing queue, to be de-queued later for encoding, thumbnail generation, and storage.
2. Encoder: To encode each uploaded video into multiple formats.
3. Thumbnails generator: We need to have a few thumbnails for each video.
4. Video and Thumbnail storage: We need to store video and thumbnail files in some distributed file storage.
5. User Database: We would need some database to store user's information, e.g., name, email, address, etc.
6. Video metadata storage: Metadata database will store all the information about videos like title, file path in the system, uploading user, total views, likes, dislikes, etc. Also, it will be used to store all the video comments.



Database Schema

Video metadata storage - MySql

- Video metadata can be stored in a SQL database.
- **Following information should be stored with each video:**
 - Videoid
 - Title
 - Description
 - Size
 - Thumbnail
 - Uploader/User
 - Total number of likes
 - Total number of dislikes
 - Total number of views
- **For each video comment, we need to store following information:**
 - CommentID
 - Videoid
 - UserID
 - Comment
 - TimeOfCreation

User data storage - MySql

- For user we need to store following details:
 - UserID
 - Name
 - email
 - address
 - age
 - registration details etc.

Detailed Component Design

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

Where would videos be stored ?

- Videos can be stored in a distributed file storage system like **HDFS** or GlusterFS.

How should we efficiently manage read traffic?

- We should segregate our read traffic from write.
- Since we will be having multiple copies of each video, we can distribute our read traffic on different servers.
- For metadata, we can have master-slave configurations, where writes will go to master first and then replayed at all the slaves.
- Such configurations can cause some staleness in data, e.g. when a new video is added, its metadata would be inserted in the master first, and before it gets replayed

at the slave, slaves would not be able to see it and therefore will be returning stale results to the user.

- This staleness might be acceptable in our system, as it would be very short lived and the user will be able to see the new videos after a few milliseconds.

Where would thumbnails be stored ?

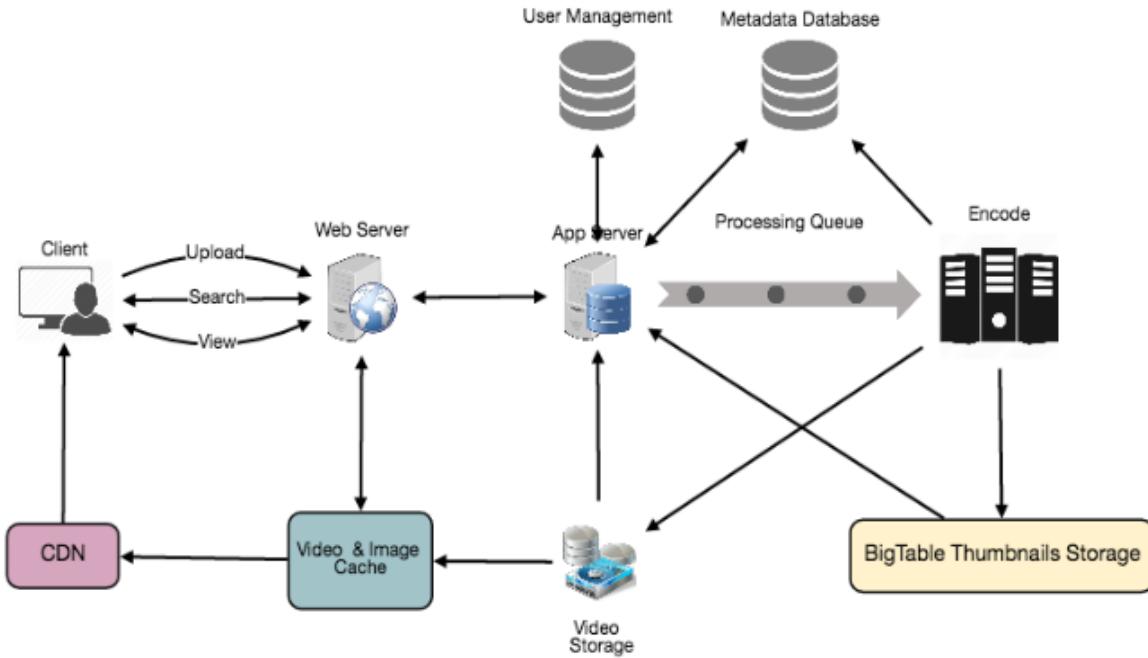
- There will be a lot more thumbnails than videos.
- If we assume that every video will have 5 thumbnails, we need to have a very efficient storage system that can serve a huge read traffic.
- There will be two considerations before deciding which storage system will be used for thumbnails:
 - Thumbnails are small files, say maximum 5KB each.
 - Read traffic for thumbnails will be huge compared to videos. Users will be watching one video at a time, but they might be looking at a page that has 20 thumbnails of other videos.
- Let's evaluate storing all the thumbnails on disk:
 - Given that we have a huge number of files; to read these files we have to perform a lot of seeks to different locations on the disk.
 - This is quite inefficient and will result in higher latencies.
- BigTable to store thumbnail ?
 - Can be a reasonable choice here, as it combines multiple files into one block to store on the disk and is very efficient in reading a small amount of data.
 - Both of these are the two biggest requirements of our service.
 - Keeping hot thumbnails in the cache will also help in improving the latencies, and given that thumbnails files are small in size, we can easily cache a large number of such files in memory.

Video Uploads:

- Since videos could be huge, if while uploading, the connection drops, we should support resuming from the same point.

Video Encoding:

- Newly uploaded videos are stored on the server, and a new task is added to the processing queue to encode it into multiple formats.
- Once all the encoding is completed; uploader is notified, and video is made available for view/sharing.



Metadata Sharding

- Since we have a huge number of new videos every day and our read load is extremely high too, we need to distribute our data onto multiple machines so that we can perform read/write operations efficiently.
- We have many options to shard our data. Let's go through different strategies of sharding this data one by one:

Sharding based on UserID Hash:

- We can try storing all the data for a particular user on one server.
- While storing, we can pass the UserID to our hash function which will map the user to a database server where we will store all the metadata for that user's videos.
- While querying for videos of a user, we can ask our hash function to find the server holding user's data and then read it from there.
- To search videos by titles, we will have to query all servers, and each server will return a set of videos.
- A centralized server will then aggregate and rank these results before returning them to the user.
- Issues with this Approach:
 - What if a user becomes popular ? There could be a lot of queries on the server holding that user, creating a performance bottleneck. This will affect the overall performance of our service.
 - Over time, some users can end up storing a lot of videos compared to others. Maintaining a uniform distribution of growing user's data is quite difficult.
- To recover from these situations either we have to repartition/redistribute our data or used **consistent hashing to balance the load between servers**.

Sharding based on Videoid hash:

- Our hash function will map each Videoid to a random server where we will store that Video's metadata.
- To find videos of a user we will query all servers, and each server will return a set of videos.
- A centralized server will aggregate and rank these results before returning them to the user.
- Issue with this Approach:
 - This approach solves our problem of popular users but shifts it to popular videos.
- We can further improve our performance by introducing cache to store hot videos in front of the database servers.

Video Deduplication

- With a huge no. of users, uploading a massive amount of video data, our service will have to deal with widespread video duplication.
- Duplicate videos often differ in aspect ratios or encodings, can contain overlays or additional borders, or can be excerpts from a longer, original video.
- **The proliferation of duplicate videos can have an impact on many levels:**
 1. **Data Storage:** We could be wasting storage space by keeping multiple copies of the same video.
 2. **Caching:** Duplicate videos would result in degraded cache efficiency by taking up space that could be used for unique content.
 3. **Network usage:** Increasing the amount of data that must be sent over the network to in-network caching systems.
 4. **Energy consumption:** Higher storage, inefficient cache, and network usage will result in energy wastage.
- For end users, these inefficiencies will cause duplicate search results, longer video startup times, and interrupted streaming.
- For our service, deduplication makes most sense early, when a user is uploading a video; as compared to post-processing it to find duplicate videos later.
- **Inline deduplication** will save us a lot of resources that can be used to encode, transfer and store the duplicate copy of the video.
- As soon as any user starts uploading a video, our service can run video matching algorithms (e.g., Block Matching, Phase Correlation, etc.) to find duplications.
- If we already have a copy of the video being uploaded, we can either stop the upload and use the existing copy or use the newly uploaded video if it is of higher quality.
- If the newly uploaded video is a subpart of an existing video or vice versa, we can intelligently divide the video into smaller chunks, so that we only upload those parts that are missing.

Load Balancing

- We should use **Consistent Hashing among our cache servers**, which will also help in balancing the load between cache servers.
- Since we will be using a static hash-based scheme to map videos to hostnames, it can lead to uneven load on the logical replicas due to the different popularity for each video.
- For instance, if a video becomes popular, the logical replica corresponding to that video will experience more traffic than other servers.
- These uneven loads for logical replicas can then translate into uneven load distribution on corresponding physical servers.
- To resolve this issue, any busy server in one location can redirect a client to a less busy server in the same cache location.
- We can use dynamic HTTP redirections for this scenario.
- **However, the use of redirections also has its drawbacks:**
 - As service tries to load balance locally, leads to multiple redirections if the host that receives the redirection can't serve the video.
 - Also, each redirection requires a client to make additional HTTP request leading higher delays before the video starts playing back.
 - Moreover, inter-tier (or cross data-center) redirections lead a client to a distant cache location because the higher tier caches are only present at a small number of locations.

Cache

- To serve globally distributed users, our service needs a massive-scale video delivery system.
- Our service should push its content closer to the user using a large number of geographically distributed video cache servers.
- We need to have a strategy that would maximize user performance and also evenly distributes the load on its cache servers.
- We can introduce a cache for metadata servers to cache hot database rows.
- Using Memcache to cache the data and Application servers before hitting database can quickly check if the cache has the desired rows.
- Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.
- **How can we build more intelligent cache ?**
 - If we go with 80-20 rule, i.e., 20% of daily read volume for videos is generating 80% of traffic.
 - Meaning that certain videos are so popular that the majority of people view them.
 - It follows that we can try caching 20% of daily read volume of videos and metadata.

Content Delivery Network (CDN)

- A CDN is a system of distributed servers that deliver web content to a user based on the geographic locations of the user, the origin of the web page and a content delivery server.
- **Our service can move most popular videos to CDNs:**
 - CDNs replicate content in multiple places. There's a better chance of videos being closer to the user and with fewer hops, videos will stream from a friendlier network.
 - CDN machines make heavy use of caching and can mostly serve videos out of memory.
 - Less popular videos (1-20 views per day) that are not cached by CDNs can be served by our servers in various data centers.

Fault Tolerance

- We should use Consistent Hashing for distribution among database servers.
- Consistent hashing will not only help in replacing a dead server but also help in distributing load among servers.

3. Designing Facebook Messenger

Things to discuss and analyze:

- Approach for one-on-one text messaging between users.
- Approach for extending the design to support group chats.
- Delivered and read status
- What action needs to be taken if the user is not connected to the internet?
- Push notifications
- Sending media like images or other documents
- Approach for providing end-to-end message encryption.

Solution 1:

Almost everyone has at least one of the popular chat apps installed on their phones. Messaging services like Facebook Messenger, WhatsApp, Discord and Slack allow you to send and receive messages from other users. Different apps have different functions, but there are certain key features that are common in all of them.

Let's design an instant messaging service that supports one-on-one chat between users through mobile and web interfaces. Depending on your requirements, you may design a system that supports group chats too.

How To Design A Messaging App

Designing a chat service is a popular interview question asked by companies like Facebook, Amazon and Twitter. As with any system design question, designing a messaging app can be a broad topic so it's important not to address a single approach and start discussing it from the beginning.

Take a few minutes to explore what the interviewer wants you to design. List down the key features that your messaging app should have and then generate a customized system based on the requirements.

Outline The Key Features

Our messaging app should at the minimum support the following key features:

1. One-on-one-chat and group chats between users.
2. Allow users to send text, pictures, videos and other files.
3. Store chat history.
4. Show the online/offline status of users.
5. Sent/delivered/read notifications.

List Down The Requirements

As of January 2021, WhatsApp supports 2 billion monthly active users according to Statistica. To build a system that can support such a large number of users, there are some additional requirements you will have to incorporate:

1. The service should be able to handle 2B users, with around 10B messages sent each day.
2. Real-time chatting with minimum latency.
3. The messaging service should be highly available, but consistency is of higher priority than availability in this case.
4. The system needs to be highly consistent. In other words, all users should see messages in the same order through any device they login from.

Expected Capacity

Estimating Message Storage

Estimating around 10B messages sent each day, with an average size of 140 bytes for each message, we'll need around 1.4 TB of storage capacity for daily messages.

$$10 \text{ billion messages} * 140 \text{ bytes} = 1.4 \text{ terabytes}$$

The above estimation assumes there's no meta data and a single copy of messages is saved. In actual messaging applications, however, metadata along with other information are also stored. Messages are replicated across multiple caches for scalability and reduced latency.

Keeping our design simple, we'll assume 1.4 TB of daily message storage. Now, assuming we store chat history for 10 years, we will need:

$$10 \text{ years} * 365 \text{ days} * 1.4 \text{ TB} \approx 5 \text{ petabytes}$$

Storing Online/Offline Status

Messages aren't the only data that is stored. In the case of Facebook Messenger, for example, the users' statuses are also stored. However, these will not need to be saved in the database, just the memory cache.

Let's assume that Facebook Messenger also has 2 billion monthly active users. Considering the real-world scenario, they won't all be active at the same time. Let's assume that half of them, i.e. 1 billion are active at a time.

If each entry takes 1 KB, you'll need:

$$1 \text{ B} * 1 \text{ KB} = 1 \text{ terabyte of memory space in the distributed cache.}$$

For a scalable service such as this one 1 TB of memory cache is reasonably affordable. The messaging service sends heartbeat timestamps to the cache. A user's online presence will be confirmed by his/her presence in the cache with a recent timestamp.

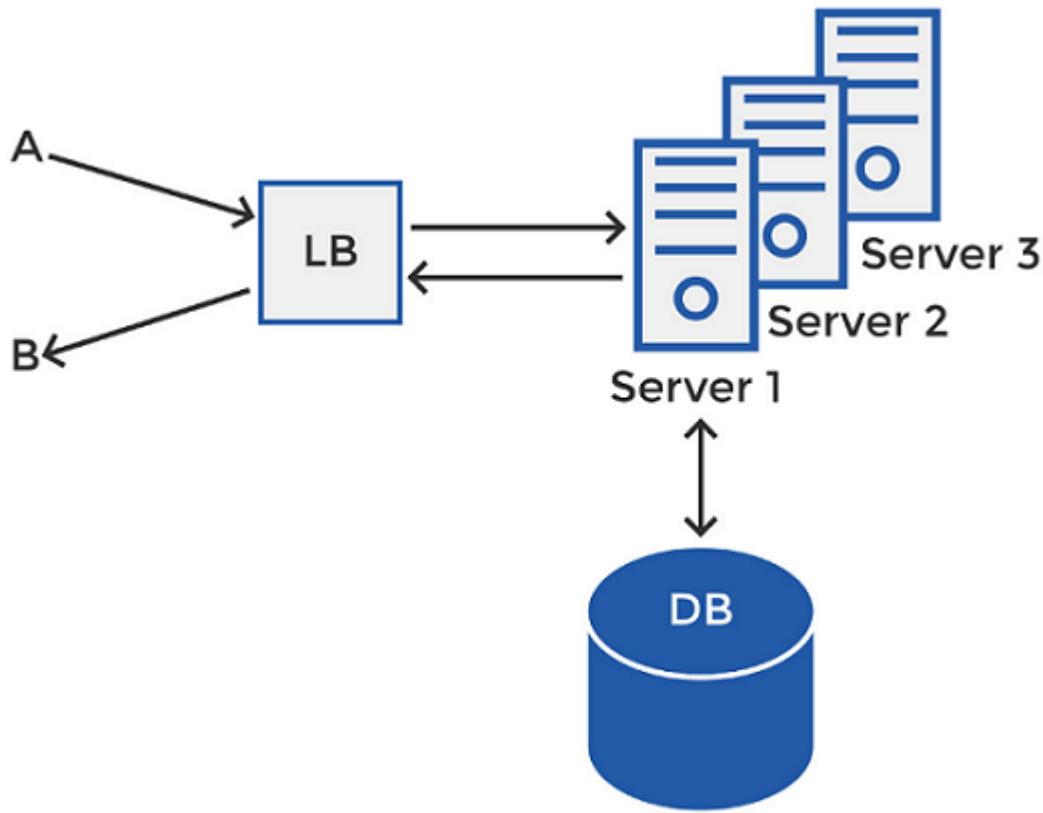
Basic Architecture

For the most elementary architecture of the system, assume that there are two users, A and B who want to communicate with each other. In the ideal scenario, where A knows the address of B and B knows the address of A, they can each send information to the other at the known IP address.



High Level Architecture

Coming closer to the real-world scenario, consider clients A and B are part of a bigger network, which also facilitates millions of other users for the exchange of messages. Now, in such cases, it's not feasible that the users know the IP addresses of other users. So instead of direct connection between the users, there has to be a server in between to manage the connections.



Load balancer connects A to an available server

In a horizontally scaled system, there will be multiple servers that are a part of the messaging system that A and B want to communicate through. If A wants to send a message to B, it will connect to the server that's free and has the capacity to serve the communication. Here's where we'll need a **load balancer**.

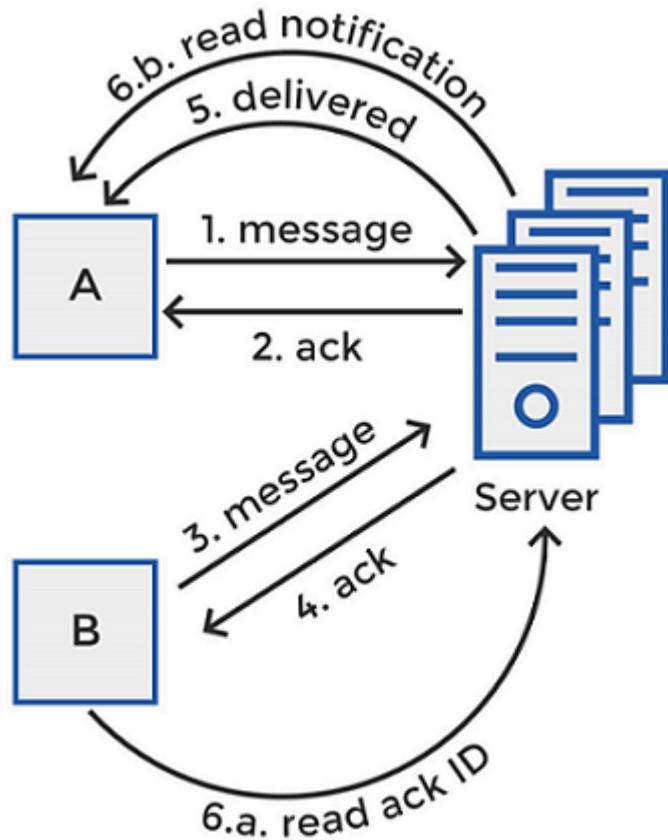
In a horizontally scaled system with a cluster of several servers, client A will connect to the load balancer, and the load balancer will decide which server will serve the client at that instant. Suppose A sends the message for B through server 2. Server 2 will store the message in the database and also relay it to B through the load balancer.

At a time, server 2 may be connected to multiple clients, coordinating the sending and receiving of their messages. Now if A wants to communicate with B, it does not need to know the IP address of B. It will simply connect to a server, server 2 in this case, and the server will know the open connection portal for B (or the process that handles B's messages). Server 2 will send the message to B via that process (more on this later in the post).

Sent/Delivered/Read Notifications

The little tick marks you see below a message you sent on WhatsApp notify you whether your message has been sent, delivered and read by the receiver. Most of you would already be familiar with what each of those signs mean. But how does WhatsApp manage these notifications for you?

Let's see how the system delivers sent, delivered and read notifications to the sender of the message.



Message notification cycle

1. A sends a message for B via a server.
2. Because the connection will follow some messaging protocol, for example TCP, an acknowledgment will be sent to A when the server receives the message. This is the **sent** notification as shown by a single grey tick on WhatsApp.
3. The server sends the message to the database for storage and also to the open connection for user B.
4. When user B receives the message, it sends an acknowledgment to the server.
5. The server sends a notification to A that the message has been delivered to B. This is the **delivered** notification as shown by a double grey tick on WhatsApp.
6. As soon as B opens his/her phone and sees the message, an acknowledgment will need to be sent to the server. Now, this acknowledgment does not need to be sent to the same server through which the message was sent. The **read** notification may be treated as an independent message with a unique message ID. The notification will be sent over a server to A. A sees this message as a double blue tick on WhatsApp, i.e. a **read** notification.

How To Handle Messages — Two Alternatives

When A establishes a connection with the server to send a message to B, the receiver B has two options for getting this message from the server:

Pull Approach

With this approach, the receiving client, B in this case, periodically requests the server for any new messages. The server maintains a queue of messages for the client and as soon as the client makes a get request, the server will send all the pending messages in the queue for that client.

To minimize delay in receiving messages, B needs to make requests to the server at a high frequency, even if many of these requests are returned with no new messages for the client. With requests and responses at such high frequencies sent for each client, the system will ask for a lot of resources, making it infeasible for a messaging app that serves billions of users. Alternatively, let's look at a different approach.

Push Approach

Through the push approach, the active clients will maintain a connection with the server in the form of a thread or a process and will wait for the server to deliver them any new messages that appear for the client through that open connection.

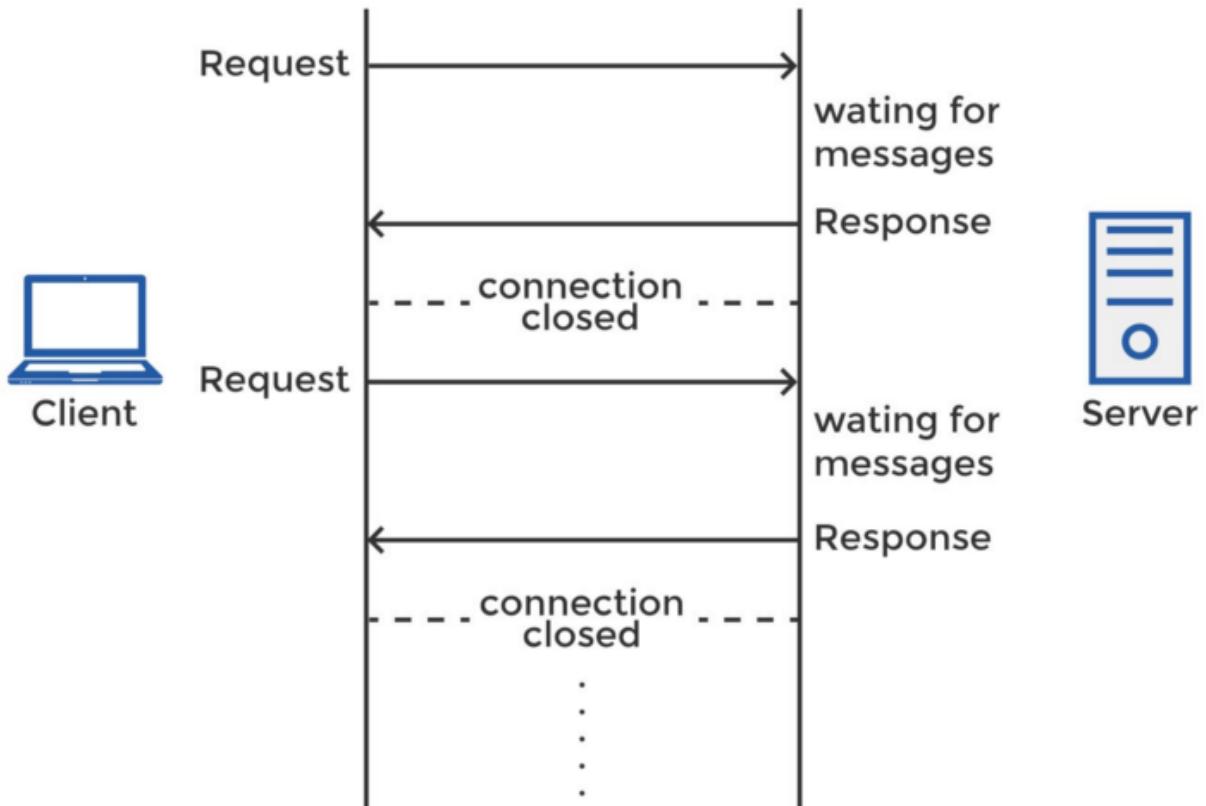
Since the server immediately relays all incoming messages to the clients, the server will not need to handle any pending messages and the overheads involved in establishing a new connection will be eliminated. This approach lowers the latency to a minimum, allowing for a faster chat between clients, which is exactly what we want in your messaging system.

Note: Since we've established that of the two approaches discussed above to handle messages, push approach is a more fitting option since it involves minimum resources and minimum latency, that's the one we'll move forward with. In the next section, we'll see how the server and client exchange information using the push approach.

Server-Client Connection

Moving forward with the push approach for the communication between the server and the client, we again have two choices: Long Polling and WebSockets.

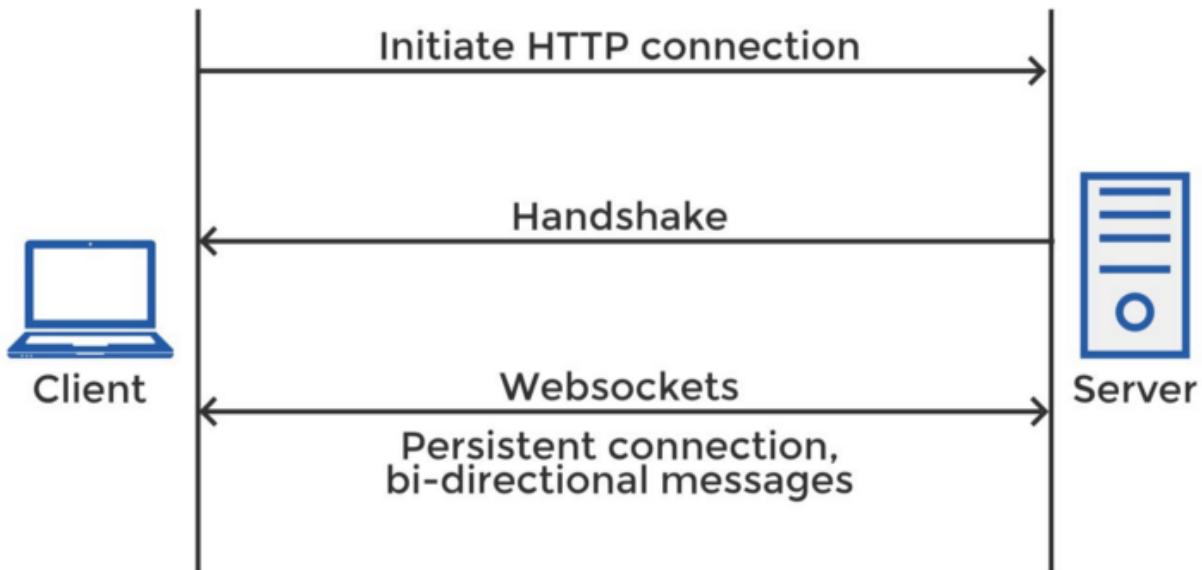
Long Polling



When a client requests the server for information, the server holds the connection open until there are new messages for the client or the connection times out. When the server responds with the new messages, the connection ends. Once the connection ends, the client immediately initiates another request to the server for messages and the cycle repeats. This approach lowers the latencies and resources involved in the pull approach (polling).

WebSockets

Alternatively, the server can send alerts to the clients via WebSocket connections. An HTTP connection set up by a client with a server can be established into a WebSocket connection via a ‘handshake’. This is a bidirectional and consistent connection between the server and client and can be used for both sending and receiving messages.



Where To Send Messages?

When using the push method, the server maintains an open connection with several active users at the same time. How will it pick which process to send an incoming message to? It will need a hash table to map the clients onto the process or thread that's open for them. This hash table will map the user ID as 'key' onto the connection ID. Once the server scans the table to locate the connection ID for the intended User ID for an incoming message, it sends the message through that open connection.

What If The Intended User Is Offline?

If a message is received for a user, let's say B, that has gone offline and is not connected with the server anymore, our messaging system will not find an entry for B in the hash table discussed above. This is because B does not have an open connection with the server, so there is no connection ID either. So where will the server save the message that came for B?

This message will be saved in the database against B's User ID. As soon user B connects to the server and an open connection is generated for B, the database is checked for any new messages against B's User ID. The message that was waiting in the database for B is relayed through the open connection for B and B receives the message.

Online/Offline Status Or Last Seen

Facebook messenger shows online/offline status of users. WhatsApp shows 'Last Seen'. Though slightly different, both these features can be implemented using the same basics. As discussed earlier in the post, the information for the user's status is saved in the memory cache rather than the database.

So, to know if the user is online or offline or when the user was 'Last Seen', we need a heartbeat that the online users periodically send to the server. The server continues updating

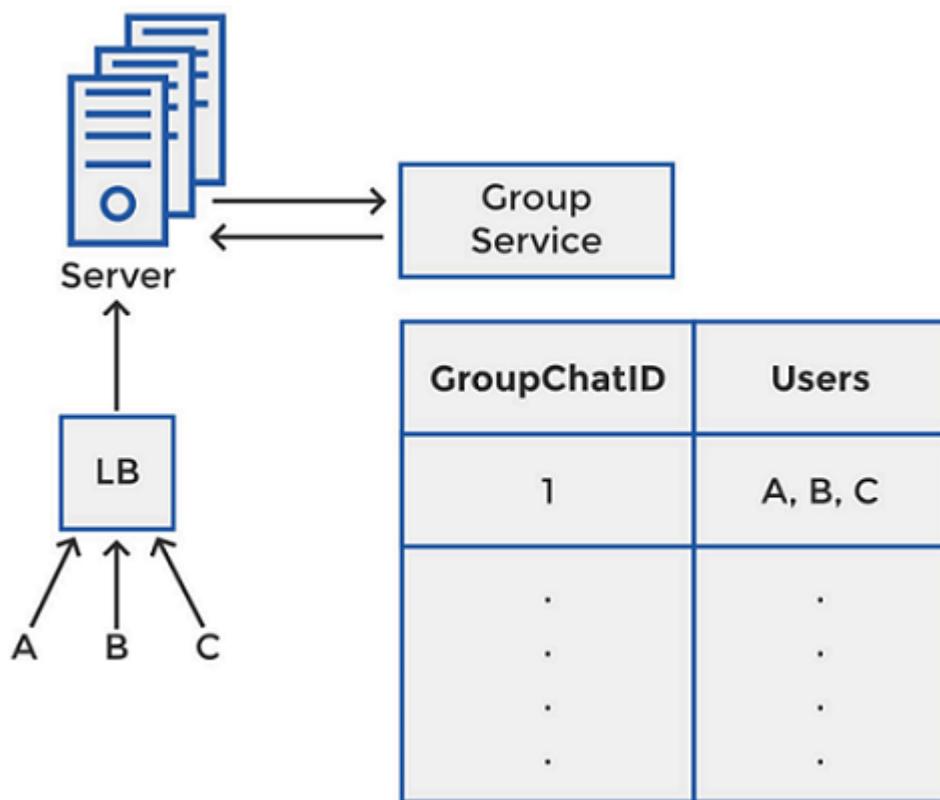
the heartbeat timestamp in a table stored in the memory cache. All the active users will have an entry in this table with a timestamp value against their user ID.

Suppose the system is configured such that the online clients send a heartbeat every 5 seconds. So a client, let's say, A, sends a heartbeat every 5 seconds, letting the server know that A is using the application. As the messaging server receives the heartbeat, it updates the time in the table.

A will be online if he/she has an entry in the table and the timestamp is recent. If another user, say B, wants to know A's status, the server can read A's timestamp in the table to check if he/she is online or offline and display it to B. Similarly, in case of WhatsApp, if B wants to know when A was last available, the server will read the timestamp against A's user ID in the table and display it to B.

Group Chat

We can extend our design for the messaging system to accommodate group chats. A message sent to a group chat by one of the members in the chat needs to be received by all the members of the chat.



Each group chat is considered as a unique object and is assigned a GroupChatID. Suppose A sends a message in a group with 1 as the GroupChatID. The load balancer will look up the server that handles group 1 and sends the message to that server. The server has a service that handles group messages and it queries this service for the other users that are also a part of the group chat. The group service maintains a table that stores all the members of each GroupChatID as you can see in the picture.

The group service returns all the users, A, B and C, that are part of the group chat 1. The server then sends the message to the different connections handling the messages for A, B and C (they can be handled by the same server or different). Through the unique open connections for A, B and C, the group message is delivered to each of the members of the chat.

Solution 2:

Problem Statement:

- Let's design an instant messaging service like Facebook Messenger.
- Users can send text messages to each other through web and mobile interfaces.
- **Difficulty:** Medium

What is Facebook Messenger ?

- Facebook Messenger is a software application which provides text-based instant messaging service to its users.
- Messenger users can chat with their Facebook friends both from cell-phones and their website.

Requirements and Goals of the System

- Our Messenger should meet the following requirements:
- Functional Requirements:
 1. Messenger should support one-on-one conversations between users.
 2. Messenger should keep track of online/offline statuses of its users.
 3. Messenger should support persistent storage of chat history.
- Non-functional Requirements:
 1. Users should have real-time chat experience with minimum latency.
 2. Our system should be highly consistent; users should be able to see the same chat history on all their devices.
 3. Messenger's high availability is desirable; we can tolerate lower availability in the interest of consistency.
- Extended Requirements:
 1. **Group Chats:** Messenger should support multiple people talking to each other in a group.
 2. **Push notifications:** Messenger should be able to notify users of new messages when they are offline.

Capacity Estimation and Constraints

- Let's assume we have **500M daily active users** and on average each user sends **40 messages daily**.
- **Total messages** for a day we have: **20B messages / day**.

Storage Estimation:

- Let's assume that on average a **message is 100 bytes**.

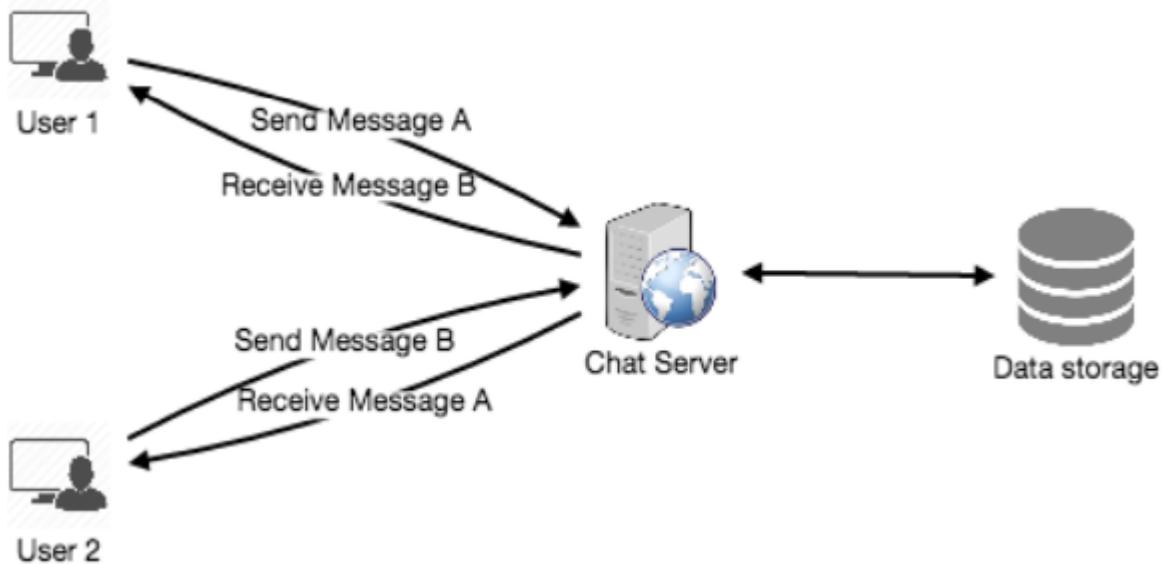
- **Total storage required** to store **all the messages for one day**: 20 Billion messages * 100 bytes => **2TB/day**
- Although Facebook Messenger stores all previous chat history.
- But just for estimation to save 5 years of chat history, **Total Storage needed**: 2 TB * 365 days * 5 years ~ = **3.6 PB**
- Other than the chat messages, we would also need to store users' information, messages' metadata (ID, Timestamp, etc.).
- Also, above calculations didn't keep data compression and replication in consideration.

Bandwidth Estimation:

- If our service is getting 2TB of data every day, then total **incoming data**: 2 TB / 86400 sec ~ = **25 MB/s**
- Since each incoming message needs to go out to another user, we need the same amount of bandwidth for both upload & download.
- Hence **outgoing data**: *****25MB/s**

High Level Design

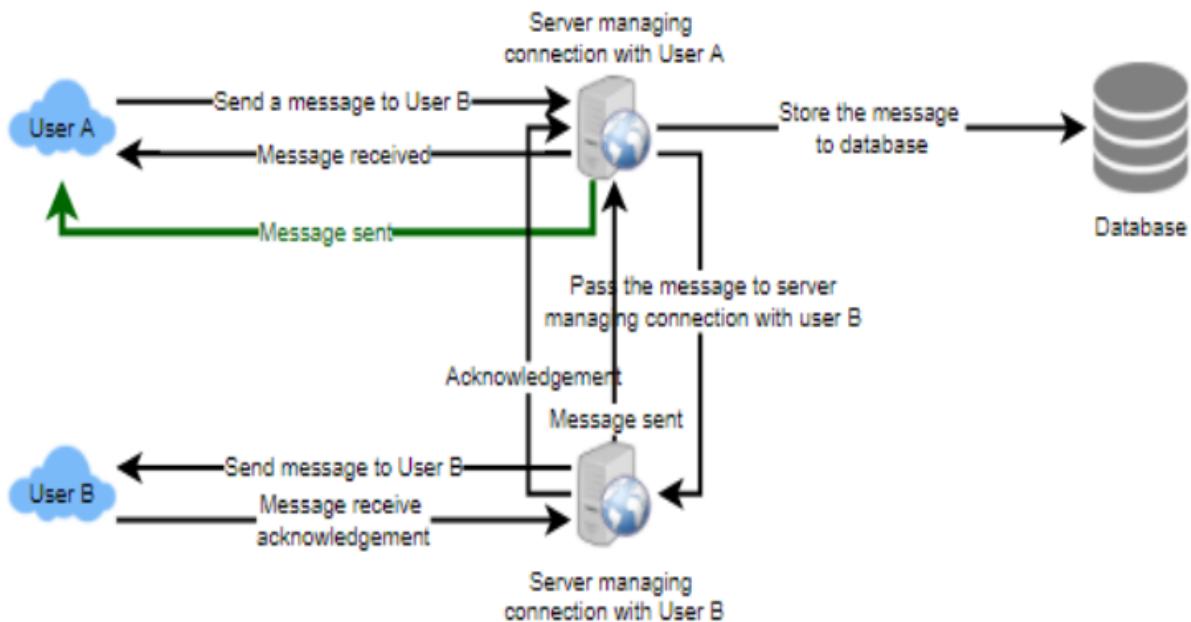
- At a high level, we will need a chat server that would be the central piece orchestrating all the communications between users.
- When a user wants to send a message to another user, they will connect to the chat server and send the message to the server.
- The server then passes that message to the other user and also stores it in the database.



Detailed Workflow

1. User-A sends a message to User-B through the chat server.
2. The server receives the message and sends an acknowledgment to User-A.
3. The server stores the message in its database and sends the message to User-B.

4. User-B receives the message and sends the acknowledgment to the server.
5. The server notifies User-A that the message has been delivered successfully to User-B.



Detailed Component Design

- Let's try to build a simple solution first where everything runs on one server.
- At the high level our system needs to handle following use cases:
 1. Receive incoming messages and deliver outgoing messages.
 2. Store and retrieve messages from the database.
 3. Keep a record of which user is online or has gone offline and notify all the relevant users about these status changes.

a) Messages Handling

How would we efficiently send/receive messages ?

- To send messages, a user needs to connect to the server and post messages for the other users.
- To get a message from the server, the user has two options:
 1. Pull model: Users can periodically ask the server if there are any new messages for them.
 - Server needs to keep track of messages that are still waiting to be delivered.

- As soon as the receiving user connects to the server to ask for any new message, the server returns all the pending messages.
 - To minimize latency, frequent check to server needed which mostly returns empty response if no pending messages.
 - This will waste a lot of resources and does not look like an efficient solution.
2. Push model: Users can keep a connection open with the server and depend on the server to notify when new messages come.
- As soon as the server receives a message it can immediately pass the message to the intended user.
 - This way, the server does not need to keep track of pending messages, and we will have minimum latency.
 - Coz the messages are delivery instantly on the opened connection.

How will clients maintain an open connection with the server ?

- Using HTTP Long Polling, where clients request information from server with the expectation that server may not respond immediately.,
- If the server has no new data for the client when the poll is received, instead of sending an empty response, the server holds the request open and waits for response information to become available.
- Once it does have new information, the server immediately sends the response to the client, completing the open request.
- Upon receipt of the server response, the client can immediately issue another server request for future updates.
- This gives a lot of improvements in latencies, throughputs, and performance.
- The long polling request can timeout or can receive a disconnect from the server, in that case, the client has to open a new request.

How can server keep track of all opened connection to efficiently redirect messages to the users ?

- Server can maintain a hash table, where "**key**" would be the **UserID** and "**value**" would be the **connection object**.
- Whenever the server receives a message for a user, it looks up for that user in hash table, finds the connection object and sends the message on the open request.

What will happen when the server receives a message for a user who has gone offline ?

- If the receiver has disconnected, the server can notify the sender about the delivery failure.
- If it is a temporary disconnect, e.g., the receiver's long-poll request just timed out, then we should expect a reconnect from the user.
- In that case, we can ask the sender to retry sending the message.
- This retry could be embedded in the client's logic so that users don't have to retype the message.
- The server can also store the message for a while and retry sending it once the receiver reconnects.

How many chat servers we need ?

- Let's plan for **500 Million connections** at any time and assuming a modern server can handle 50K concurrent connections at any time.
- **Total Servers** needed: we would need 10K such servers.

How to know which server holds the connection to which user ?

- We can introduce a software load balancer in front of our chat servers.
- It maps each UserID to a server to redirect the request.

How should the chat server process a 'deliver message' request ?

- The server needs to do following things upon receiving a new message:
 1. Store the message in the database.
 2. Send the message to the receiver.
 3. Send an acknowledgment to the sender.
- Chat server will first find the server that holds the connection for the receiver and pass the message to that server to send to receiver.
- The chat server can then send the acknowledgment to the sender.
- Don't need to wait for storing the message in the database, this can happen in the background.

How does the messenger maintain the sequencing of the messages ?

- We can store a timestamp with each message, which would be the time when the message is received at the server.
- But this will still not ensure correct ordering of messages for clients.
- The scenario where the server timestamp cannot determine the exact ordering of messages would look like this:
 - User-1 sends a message M1 to the server for User-2.
 - The server receives M1 at T1.
 - Meanwhile, User-2 sends a message M2 to the server for User-1.
 - The server receives the message M2 at T2, such that T2 > T1.
 - The server sends message M1 to User-2 and M2 to User-1.
- So User-1 will see M1 first and then M2, whereas User-2 will see M2 first and then M1.
- To resolve this, we need to keep a sequence number with every message for each client.
- This sequence number will determine the exact ordering of messages for EACH user.
- With this solution, both clients will see a different view of message sequence, but this view will be consistent for them on all devices.

b) Storing and Retrieving Messages from DB

- Whenever the chat server receives a new message, it needs to store it in the database, 2 options to do this.

1. Start a separate thread, which will work with the database to store the message.
 2. Send an asynchronous request to the database to store the message.
- We have to **keep certain things in mind while designing our database:**
 1. How to efficiently work with database connection pool.
 2. How to retry failed requests ?
 3. Where to log those requests that failed even after certain retries ?
 4. How to retry these logged requests (that failed after the retry) when issues are resolved?

Which storage system we should use ?

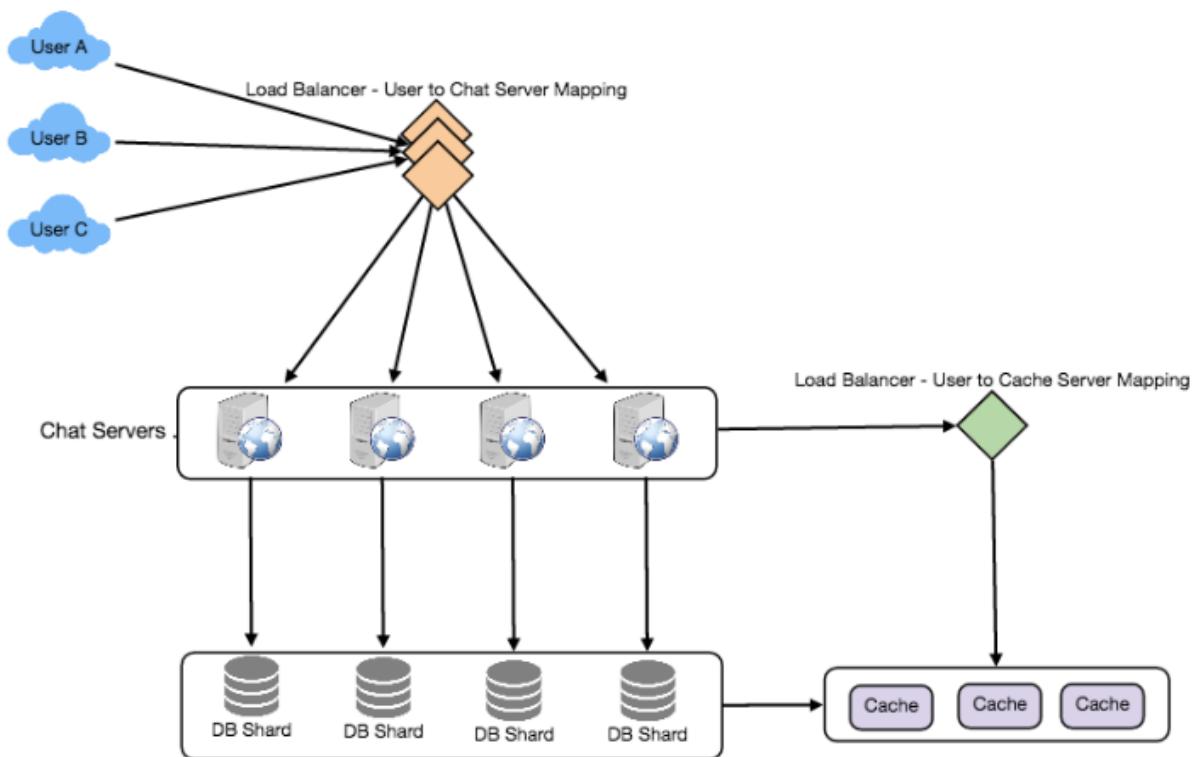
- We need to have a database that can:
 - Support a very high rate of small updates and also
 - Fetch a range of records quickly.
- This is required because we have a huge number of small messages that need to be inserted in the database and while querying a user is mostly interested in accessing the messages in a sequential manner.
- We cannot use RDBMS like MySQL or NoSQL like MongoDB because we cannot afford to read/write a row from the database every time a user receives/sends a message.
- This will make not only basic operations of our service to run with high latency but also create a huge load on databases.
- Both of our requirements can be easily met with a wide-column database solution like **HBase**.
- What is HBase ?
 - HBase is a column-oriented key-value NoSQL database that can store multiple values against one key into multiple columns.
 - HBase is modeled after **Google's BigTable** and runs on top of Hadoop Distributed File System (**HDFS**).
 - HBase groups data together to store new data in a memory buffer and once the buffer is full, it dumps the data to the disk.
 - This way of storage not only helps storing a lot of small data quickly but also fetching rows by the key or scanning ranges of rows.
 - HBase is also an efficient database to store variable size data, which is also required by our service.

How should clients efficiently fetch data from the server ?

- Clients should paginate while fetching data from the server.
- Page size could be different for different clients, e.g., cell phones have smaller screens, so we need lesser number of messages or conversations in the viewport.

c) Managing user's status

- We need to keep track of user's online/offline status and notify all the relevant users whenever a status change happens.
- Since we are maintaining a connection object on the server for all active users, we can easily figure out user's current status from this.
- With **500M active users at any time**, to broadcast each status change to all the relevant active users, it will consume a lot of resources.
- We can do the following optimization around this:
 1. Whenever a client starts the app, it can pull current status of all users in their friends' list.
 2. Whenever a user sends a message to another user that has gone offline, we can send a failure to the sender and update the status on the client.
 3. Whenever a user comes online, the server can always broadcast that status with a delay of few seconds to see if the user does not go offline immediately.
 4. Client's can pull the status from the server about those users that are being shown on the user's viewport. This should not be a frequent operation, as server is broadcasting online status of users and we can live with the stale offline status of users for a while.
 5. Whenever the client starts a new chat with another user, we can pull the status at that time.



Design Summary:

- Clients will open a connection to the chat server to send a message; the server will then pass it to the requested user.
- All the active users will keep a connection open with the server to receive messages.

- Whenever a new message arrives, the chat server will push it to the receiving user on the long poll request.
- Messages can be stored in HBase, which supports quick small updates, and range based searches.
- The servers can broadcast the online status of a user to other relevant users.
- Clients can pull status updates for users who are visible in the client's viewport on a less frequent basis.

Data partitioning

- Since we will be storing a lot of data (3.6PB for five years), we need to distribute it onto multiple database servers.
- What would be our partitioning scheme ?

Partitioning based on UserID:

- Let's assume we partition based on the hash of the UserID, so that we can keep all messages of a user on the same database.
- Assuming **1 DB shard is 4TB**, then Total Shards: $3.6\text{PB}/4\text{TB} \approx 900 \text{ shards for 5 years}$.
- For simplicity, let's assume we keep **1K shards**.
- We will find the shard number by "**hash(UserID) % 1000**" and then store/retrieve the data from there.
- This partitioning scheme will also be very quick to fetch chat history for any user.
- In the beginning, we can start with fewer database servers with multiple shards residing on one physical server.
- Since we can have multiple database instances on a server, we can easily store multiple partitions on a single server.
- Hash function needs to understand this logical partitioning scheme so that it can map multiple logical partitions on one physical server.
- Since we will store an infinite history of messages, we can start with a big number of logical partitions, which would be mapped to fewer physical servers, and as our storage demand increases, we can add more physical servers to distribute our logical partitions.

Partitioning based on MessageID:

- If we store different messages of a user on separate database shard, fetching a range of messages of a chat would be very slow, so we should not adopt this scheme.

Cache

- We can cache a few recent messages (say last 15) in a few recent conversations that are visible in user's viewport (say last 5).

- Since we decided to store all of the user's messages on one shard, cache for a user should completely reside on one machine too.

Load balancing

- We will need a load balancer in front of our chat servers; that can map each UserID to a server that holds the connection for the user and then direct the request to that server.
- Similarly, we would need a load balancer for our cache servers.

Fault tolerance and Replication

What will happen when a chat server fails ?

- Our chat servers are holding connections with the users.
- If a server goes down, should we devise a mechanism to transfer those connections to some other server ?
- It's extremely hard to failover TCP connections to other servers.
- An easier approach can be to have clients automatically reconnect if the connection is lost.

Should we store multiple copies of user messages ?

- We can't have only one copy of user's data, because if the server holding the data crashes or is down permanently, we don't have any mechanism to recover that data.
- For this either we have to store multiple copies of the data on different servers or use techniques like [Reed-Solomon](#) encoding to distribute and replicate it.

Extended Requirements

a) Group chat

- We can have separate group-chat objects in our system that can be stored on the chat servers.
- A group-chat object is identified by **GroupChatID** and will also maintain a list of people who are part of that chat.
- Our load balancer can direct each group chat message based on GroupChatID and the server handling that group chat can iterate through all the users of the chat to find the server handling the connection of each user to deliver the message.
- In databases, we can store all the group chats in a separate table partitioned based on GroupChatID.

b) Push notifications

- In current design user's can only send messages to active users and if the receiving user is offline, we send a failure to the sending user.
- Push notifications will enable our system to send messages to offline users.
- Push notifications only work for mobile clients.
- Each user can opt-in from their device to get notifications whenever there is a new message or event.
- Each mobile manufacturer maintains a set of servers that handles pushing these notifications to the user's device.
- To have push notifications in our system, we would need to set up a Notification server.
- It will take the messages for offline users and send them to mobile manufacture's push notification server, which will then send them to the user's device.

Solution 3:

1.1. Requirements

First thing first, we must figure out the exact requirements of the target system. For Slack (or other apps you use), the following features can be expected:

- Direct messaging: two users can chat with each other
- Group chat: users can participate in group conversations
- Join/leave groups, (add/delete friends not important for Slack)
- Typing indicator: when typing, the recipient gets notified
- User status: whether you are online or offline
- When the user is offline, try send notifications to the user's mobile device if a new message arrives.

Besides the above features, the system should obviously be scalable and highly available.

1.2. Traffic Estimation

Some interviewers I talked to didn't like upfront back-of-the-envelope estimation because the statistics are arbitrarily picked and the exact number doesn't mean much. To some extent, they are right. However, we do need some rough numbers to help us make critical decisions:

- We expect millions of daily active users(DAU) from all over the world.
- On average each message is around a few hundred characters. Each person sends out a few hundred messages per day.
- Some groups have a handful of people, some others have hundreds of members

Based on the numbers above, we can make the following conclusions:

- it is safe to say that we need a globally distributed system (to serve users in different regions). Users will be connected to different servers.
- The backend database should be horizontally scalable, as tons of messages (~100 GB) are saved each day.

- The system is write-heavy, as messages are written into DB but rarely read (most of them are delivered via the notification service and saved locally on the client. We shall see why later)

2. High-level Design

2.1. Database Design

When it comes to database design, it is always a good idea to consider the access pattern of your application.

Read Operations

- Given user A and user B, retrieve messages after a certain timestamp
- Given group G, retrieve all messages after a certain timestamp
- Given group G, find all member ID
- Given user A, find all groups he/she joined

Write Operations

- Save a message between user A and user B
- Save a new message by user A in group G
- Add/delete user A to/from group G

In our case, it is obvious that the database is used primarily as a key-value store. No complex relational ops such as join are needed. In addition to the access pattern, keep in mind that the database must be horizontally scalable and tuned for writes.

In our case, we could use NoSQL databases such as Cassandra or sharded SQL such as Postgres (SQL is also very scalable if you don't care about relations or foreign key constraints). In practice, many companies (e.g. Discord) use Cassandra because it scales up easily and is more suitable for write-heavy work (Cassandra uses LSTM rather than B+ Tree used by SQL). For a more detailed discussion, I refer you to this awesome [post](#).

Schema

In Cassandra, records are sharded by the partition keys. On each node, records with the same partition key are sorted by the sort key. The chat tables are designed to best fit our access pattern.

Private Chat Table

user ID 1	user ID 2	message ID	timestamp	message
John	Sally	8492	2021/12/21..	"Hello"

partition key sort key

Figure 1. Private Chat Table, Figure by Author

Group Chat Table

Group ID	bucket	message ID	timestamp	message	user ID
CSCI350	3	12345	2021/12/21..	"hw posted"	Jack

partition key sort key

Figure 2. Group Chat Table, Figure by Author

A key observation here is that the message ID is used to determine the ordering. The message ID is **not** globally unique, as its scope is determined by the partition key. The system will never retrieve an item by its message ID alone.

However, auto-increment key generation is not well supported by distributed databases. We could use a dedicated key generation service such as [Twitter's Snowflake ID](#) or simply use a precise timestamp as message ID (yes, [clock skew can happen](#), but the message volume in a group/between two users may be small enough to ignore this with careful time synchronization)

Group Membership Table

Group ID	user ID
CSCI109	Sally

User Membership Table

User ID	Group ID
John	CSCI109

partition key sort key partition key sort key

Figure 3. Membership Tables, Figure by Author

We need two tables to capture the relationship between users and groups. The Group Membership table is used for message broadcasting — we need to figure out who gets the message. The User Membership table is for listing all the groups a user has joined. We could use a single table with a secondary index, but the [cardinality of group ID/user ID is too large for a secondary index](#). The two-table approach isn't without problems either— if we mutate one, the other should be modified to maintain consistency, which requires distributed transactions.

User Profile Table

user ID	password hash	status	profile pic url	...
John	*****	ONLINE	"s3://photo.."	

partition key

Figure 4. User Profile Table, Figure by Author

Finally, the user profile table. It keeps track of user-specific data such as profile picture location and other stuff.

2.2. API Design

In a system design interview, it is always a good idea to lay down the API of the system upfront. It helps you formalize the features to implement and showcase your rigorous thinking.

The requirements of our system can be broken down into the following RPC calls:

```
send_message(user_id, receiver_id, channel_type, message)  
get_messages(user_id, user_id2, channel_type, earliest_message_id)  
join_group(user_id, group_id)  
leave_group(user_id, group_id)  
get_all_group(user_id)  
  
// ignore RPC for login/logout, ignore authentication token
```

- The *channel_type* field is used to distinguish private chats from group chats.
- The *receiver_id /user_id2* can be a user ID or a group ID.
- The *earliest_message_id* is the latest message locally available on the client. It is used as the sort key to range query the chat tables.

2.3. Architecture

Finally, time for architecture design! By now we've laid down a solid foundation for the application — the database schema, the RPC calls. With all these in mind, we can proceed to write down the list of components in the system.

- **Chat Service:** each online user maintains a WebSocket connection with a WebSocket server in the Chat Service. Outgoing and incoming chat messages are exchanged here.
- **Web Service:** It handles all RPC calls except *send_message()*. Users talk to this service for authentication, join/leave groups, etc. No WebSocket is needed here since all calls are client-initiated and HTTP-based.
- **Notification Service:** When the user is offline, messages are pushed to external phone manufacturers' notification servers (e.g. [Apple's](#))
- **Presence Service:** When a user is typing or changes status, the Presence Service is responsible for figuring out who gets the push update.
- **User Mapping Service:** Our chat service is globally distributed. We need to keep track of the server ID of the user's session host.

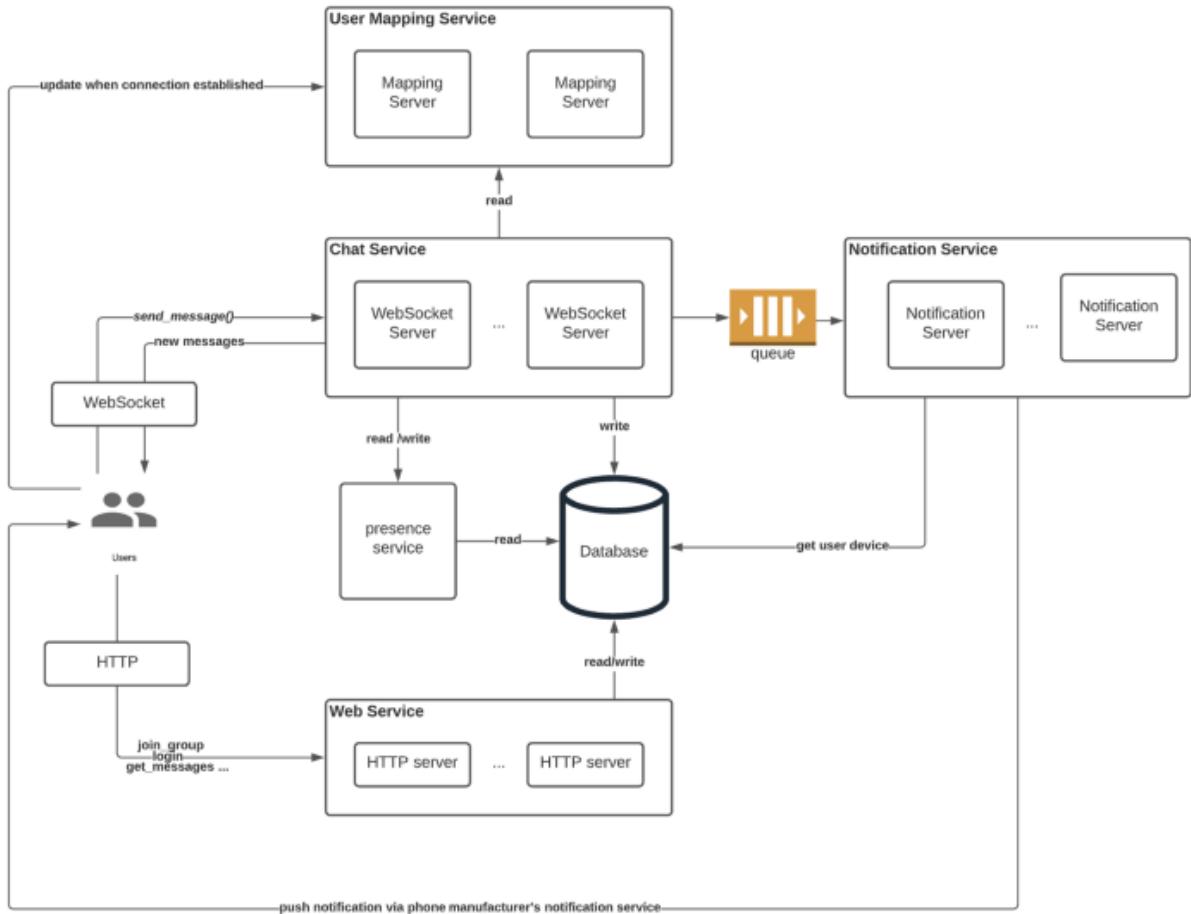


Figure 5. High-Level Architecture, Figure by Author

Note that the ID generation service is left out in figure 5.

2.4 Dataflow

Normal Message Delivery

When the user sends out a message, it is delivered to a WebSocket server in the same region. The WebSocket server will write the message to the database and acknowledge the client. If the recipient is another user, her WebSocket service ID is obtained by calling the User Mapping Service. Once the message is forwarded to the appropriate server, the recipient will get the push message via WebSocket.

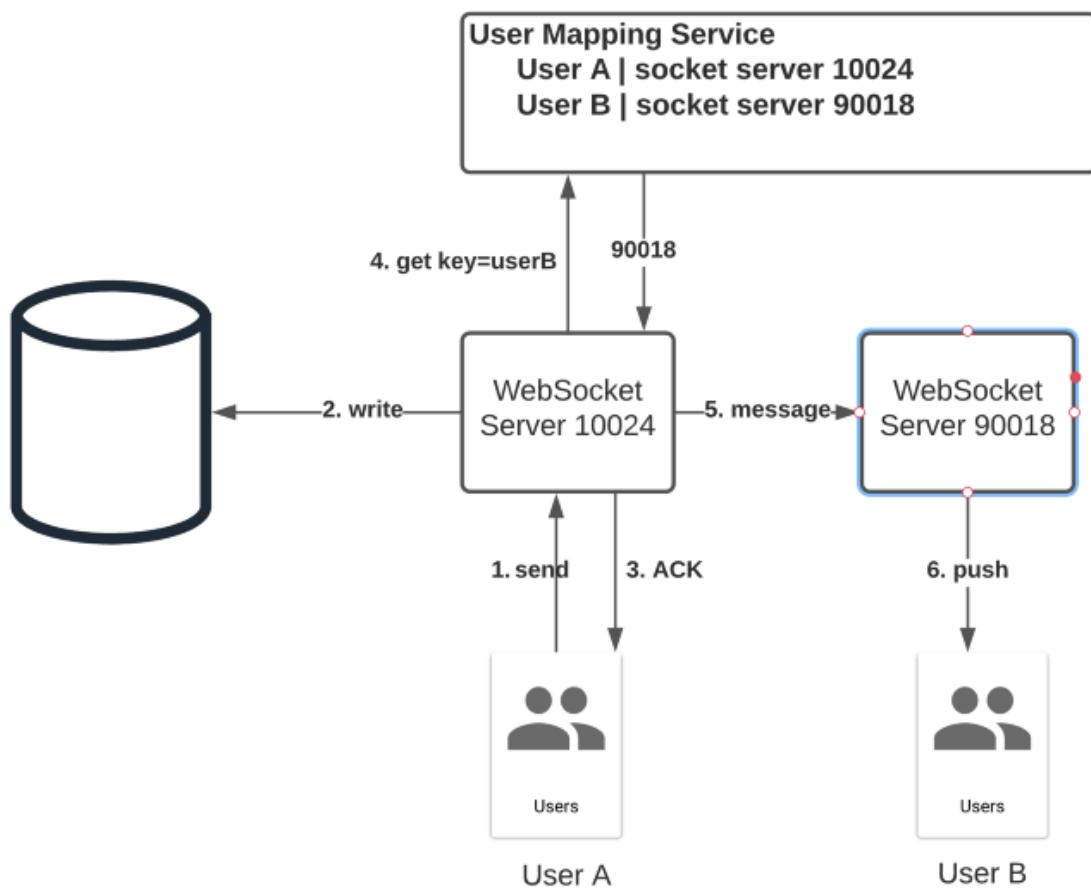


Figure 6. Normal Message Delivery, Figure by Author

Failed Message Delivery

If for some reason the WebSocket is cut off and the user is unreachable, all messages will be redirected to the notification service for a best-effort delivery (no guarantee of delivery, since the user might not have an internet connection).

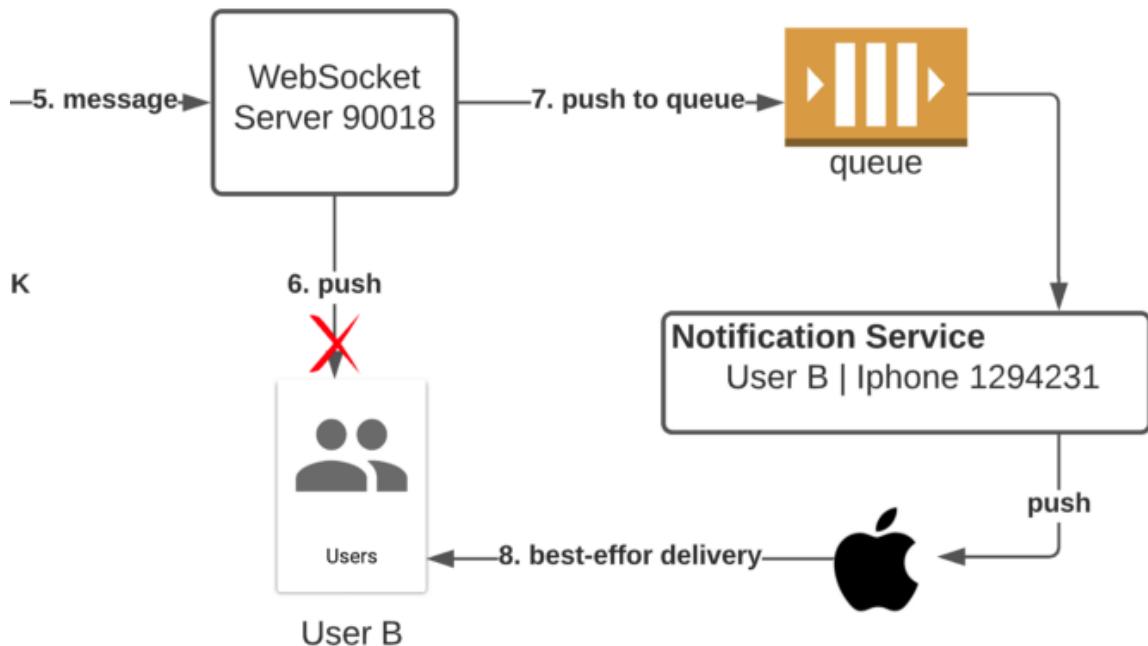


Figure 7. Failed Message Delivery, Figure by Author

History Catch-up

Even with the duo message delivery system, it is possible that a message is never received by the client. Therefore, it is critical that all clients request the Gateway Service for the authoritative chat history upon reconnection/with fixed intervals.

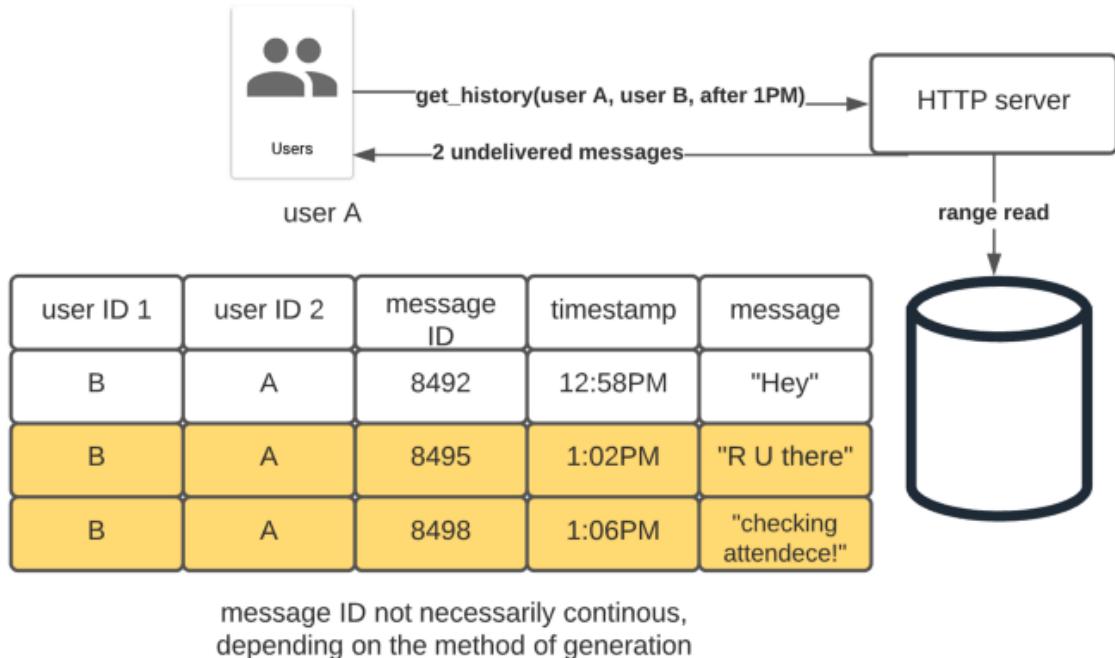


Figure 8. History Catch-up, Figure by Author

Group Chat

When a user participates in a group chat, his message is broadcast to all group members. Given a group ID, the WebSocket server queries the database forward the messages to other servers.

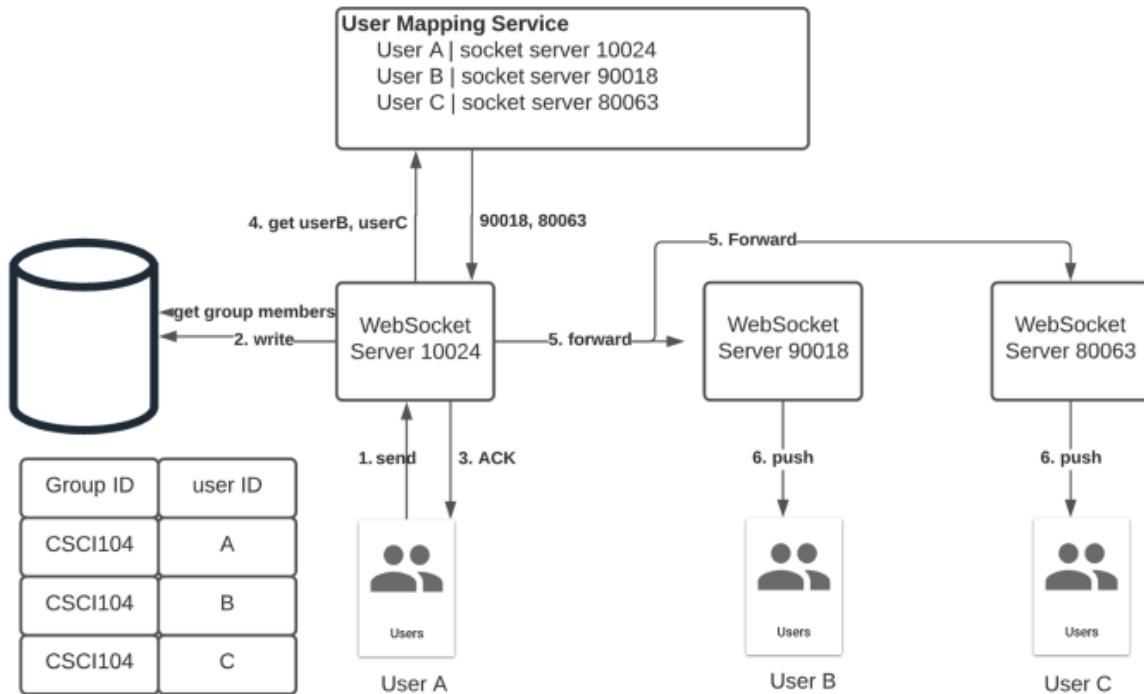


Figure 9. Group Chat, Figure by Author

Presence Detection

When a user is active in the app, applications like Slack will inform other users of your status. We can handle this signal as a special form of group chat. Instead of querying the database, the WebSocket server contacts the Presence Service who returns a list of contacts for broadcast. Since it is prohibitively expensive to inform everyone who has ever chatted with the user, the Presence Service runs some kind of algorithm (maybe based on chat frequency, latest exchange, ...) to keep the list short and precise.

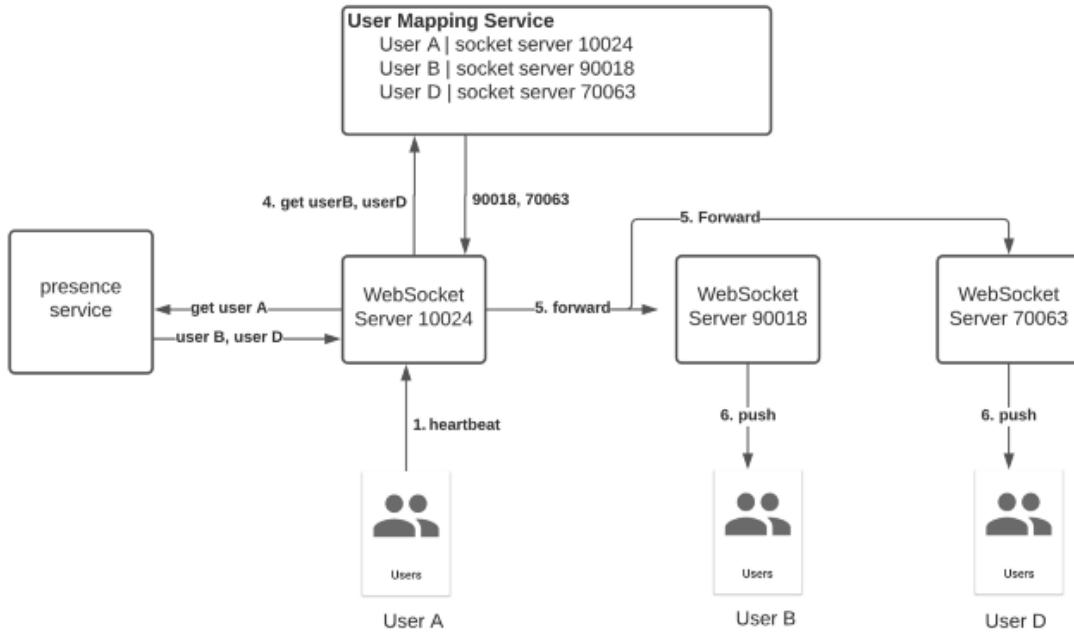


Figure 10. Presence Detection, Figure by Author

For less-frequent contacts, the status of a user can be obtained by polling the presence service instead of waiting for the push message.

Typing Indication

Logically there's no difference between a typing indication message and a normal message. We can propagate the typing indication messages in the same way as private chat messages (figure 6).

3. Details

By now, we have established a clear architecture and concrete data flow that covers all functional requirements. However, there are so many things over which an interviewer can grill you. In this section, I want to talk about some interesting tradeoffs that interviewers might ask you.

3.1. Communication Between Web Socket Servers

The first issue is how web socket servers communicate with each other (step 5 in figure 6). The easiest way is to use a separate, synchronous HTTP call for each message. However, two issues arise:

1. No message ordering: If two messages are sent in sequence, it is possible that the first one arrives super late. It will seem to the user that an unread message pops out above the latest message you just read, which is very confusing.
2. Large number of request per second: If each message is sent with a separate HTTP call, then the ingress traffic could overwhelm web socket Servers.

One possible solution is using queues as middleware. Each server has a dedicated incoming queue, which serves as a buffer and imposes ordering on messages.

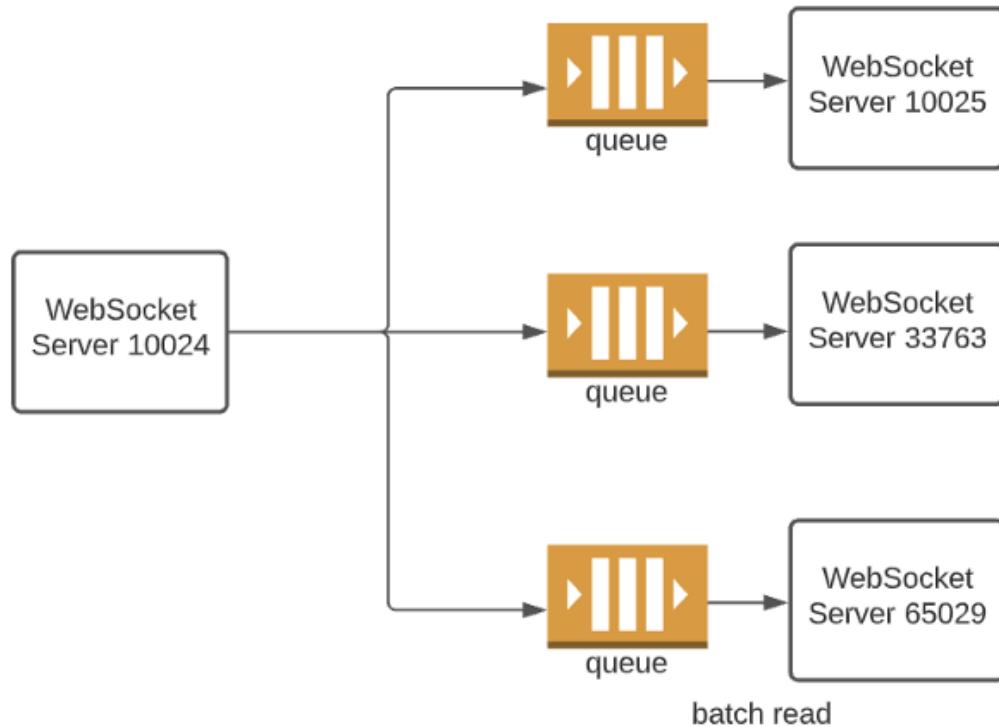


Figure 11. Queue as a Middleware. Figure by Author

Although it looks like that queue is a better solution, I would argue that it's a devil of its own:

- For large-scale applications such as Slack, there are tens of thousands of web socket servers. Maintains and scales the middleware is expensive and a challenge itself.
- If the consumer (web socket server) is down, we don't want the messages to accumulate in the queue since the users will reconnect to a different server and initiate history catch-up. Servers come and go all the time, it is laborious to create/purge queues with them.
- If the consumer rejoins, what should we do with the stale messages in the queue? What messages to discard and what to process?

Here I propose a third approach that is based on synchronous HTTP calls:

To solve the ordering issue, we can annotate every message with a *prevMsgID* field. The recipient checks his local log and initiates history catch-ups when an inconsistency is found.

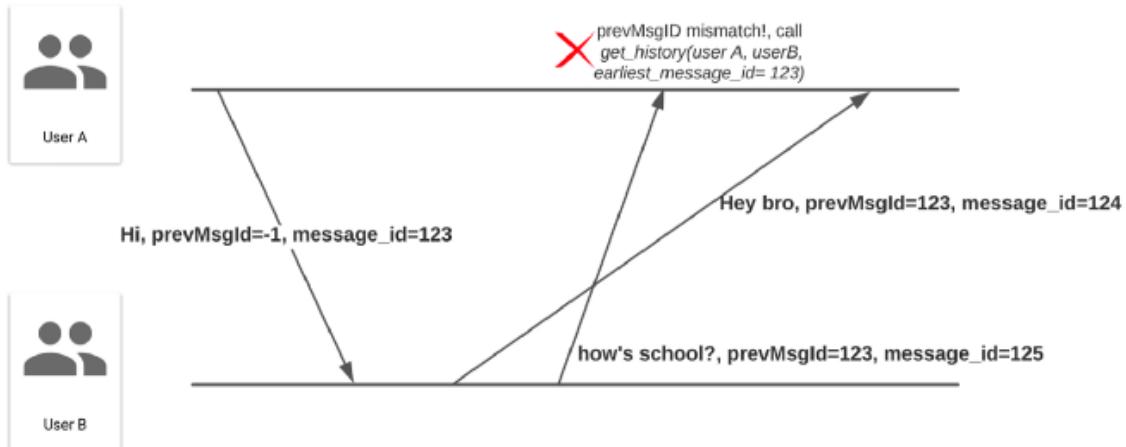


Figure 12. Out-of-order Detection, Figure by Author

To reduce the number of messages exchanged between servers, we can implement some buffering algorithm on WebSocket servers — send accumulated messages, say, ~50 MS with randomized offsets (preventing everyone from sending messages at the same time).

3.2. Handling Group Chat

Right now the current design broadcasts all group messages regardless of group size. This approach does not work well if the group is very large. Theoretically, there are two ways to handle group chats — pushing and pulling:

- pushing: The message is broadcast to other web socket servers who then push it to the client
- pulling: the client sends a request to HTTP servers for the latest messages.

The problem with pushing is that it converts one external request (one message) into many internal messages. This is called Write Amplification. If the group is large and active, pushing group messages will take up a tremendous amount of bandwidth.

The problem with pulling is that one message is read over and over again by different clients (Read Amplification). Going along with this approach will surely overwhelm the database.

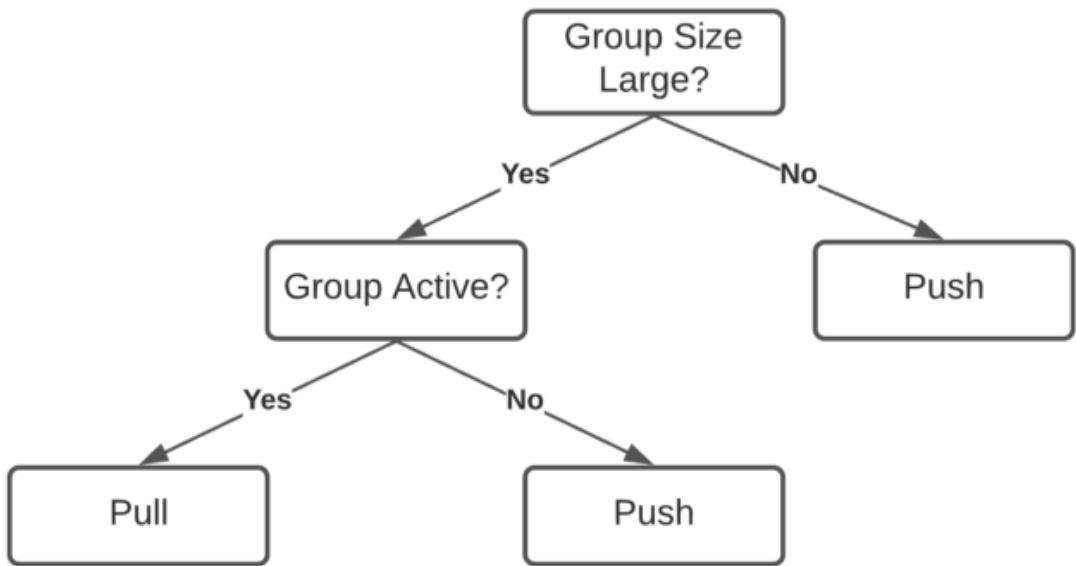


Figure 13. Hybrid Group Chat Handling, Figure by Author

My idea is that we can build a hybrid system with the above logic. For smaller groups or inactive groups, it is okay to do pushing since write amplification won't stress out the servers. For very active and large groups, clients must query the HTTP server regularly for messages.

3.3. Web Socket Server and Database

When a message is received by a web socket server, it is persisted into the database. The simplest approach would be calling the database directly. With this implementation, the database is bombarded with ~100K RPS traffic, which demands tons of infrastructure.

An alternative is using queues for batch insertion. When the queue service receives a write, the web socket server, in turn, acknowledges the client. At the other end of the queue, a dedicated batch writer can group hundreds of messages into a single request, thus reducing the RPS significantly.

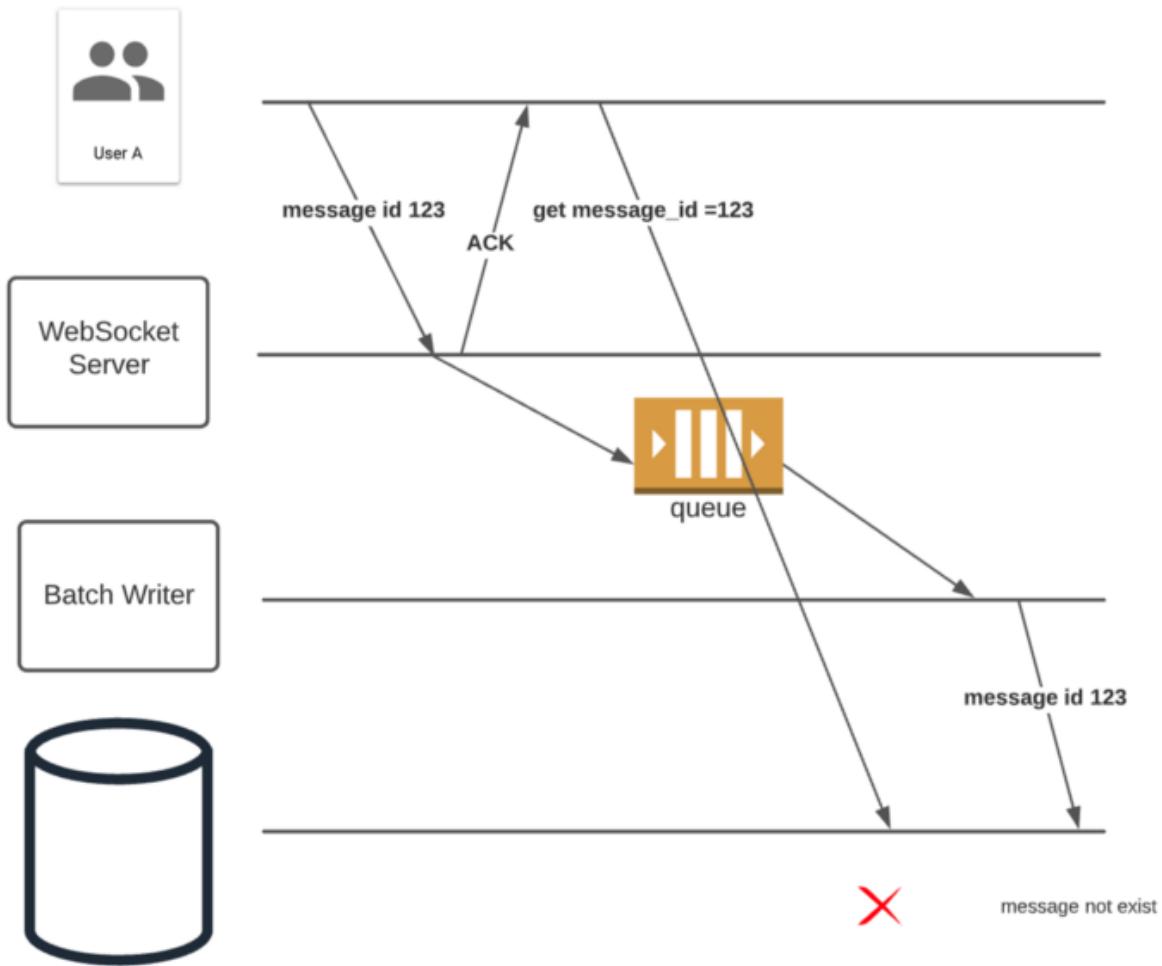


Figure 14. Issues with Middleware, Figure by Author

However, using queues such as Kafka does introduce delays between server ACK and the actual write. If a client queries the HTTP server immediately after receiving ACK from the web socket server, it is possible the message is not written yet. With middleware, it is hard to offer read-after-write consistency that might be important for certain applications.

4. Design Quora/Reddit/HackerNews (A Social Network + Message Board Service)

These services allow users to post questions, share links and answer the questions of other users. Users can also comment on questions or shared links.

Things to discuss and analyze:

- Approach to record stats of each answer such as the number of views, up-votes/down-votes, etc.
- Follow options should be there for users to follow other users or topics.

- News feed generation which means users can see the list of top questions from all the users and topics they follow on their timeline.

5. Designing Typeahead Suggestion

Typeahead service allows users to type some query and based on that it suggests top searched items starting with whatever the user has typed.

Things to discuss and analyze:

- Approach to store previous search queries
- Real-time requirement of the system
- Approach to keep the data fresh.
- Approach to find the best matches to the already typed string
- Queries per second to be handled by the system.
- Criteria for choosing the suggestions.
- A total number of data to be stored.
- Approach to find the best matches to the already typed string

Solution 1:

Typeahead suggestion or Autocomplete is an influential feature of any modern search text for searching. It recommends words to users as they enter text for searching. It suggests a list of different words or sentences based on the characters the user enters. It assists the user to frame their search queries nicely. It facilitates users to search for available and frequently searched terms.

Requirements

Requirements are an essential aspect of any system design strategy for the development of a product. Below are important topics that must be clarified and covered at the time of requirements:

- Functioning backwards
- Functional requirements
- Non-functional requirements

Functioning Backwards

Functioning backwards is mainly focused on the customer experiences to define requirements. Below are questions that should run in mind:

- Identification of customers and their type of character
- Use cases of customers or users
- Considering problems from a business standpoint and not only the virtuously technical side.

Below are functioning backwards points for the **typehead suggestions** for customers or users:

- Customers can expect the desired list of suggestions by entering only limited characters or words.
- There can be many customers who are not proficient in the correct spelling of search texts or words.
- Customers will not only expect the speeding up their search but also help them to articulate their search queries better.

Functional Requirements

A functional requirement characterises products features or functions. It is predominantly focused on the requirements of its system or components. This should be started from the customer experience.

Functional requirements for the typehead suggestion

As the user types in their textbox, it should suggest a list of terms starting and matching with whatever they have typed.

Questions/answers for the functional requirement gathering:

The number of suggestions.

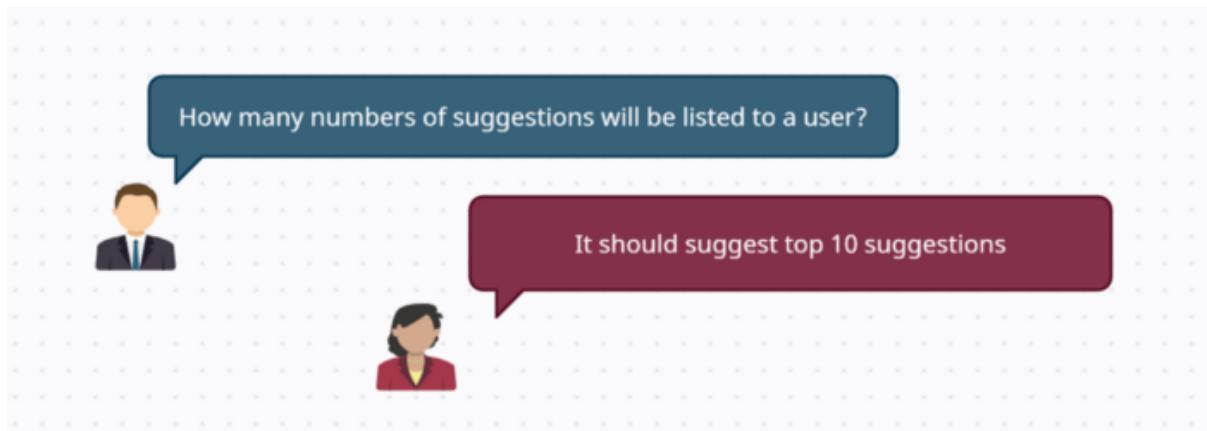


Figure 2: Question about the number of suggestions

Order of suggestions.

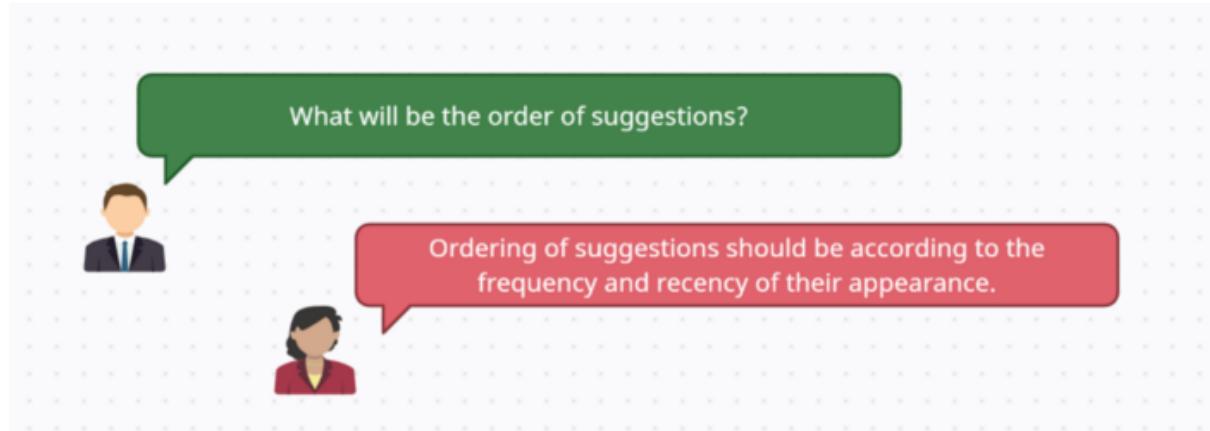


Figure 3: Question about the Order of suggestions

Local storage.

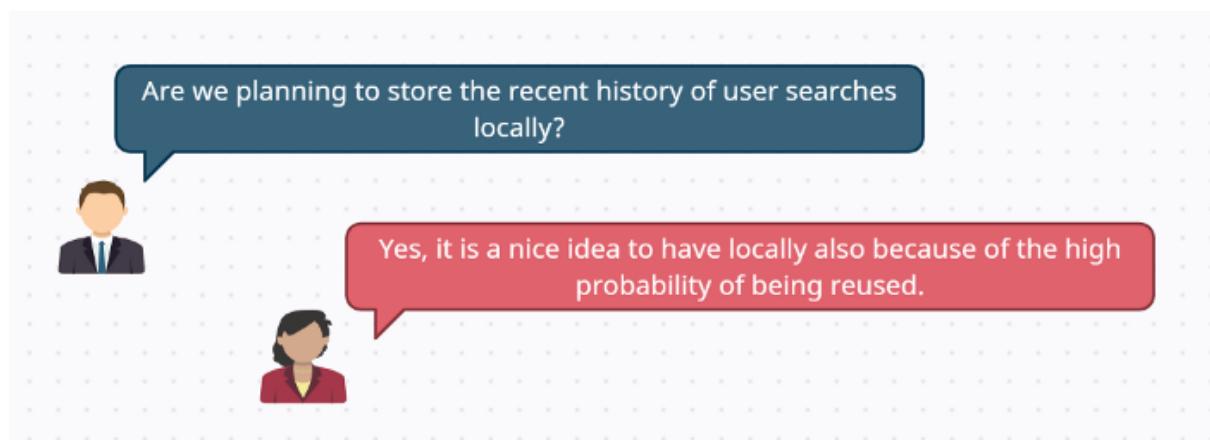


Figure 4: Question about local storage

Personalization.

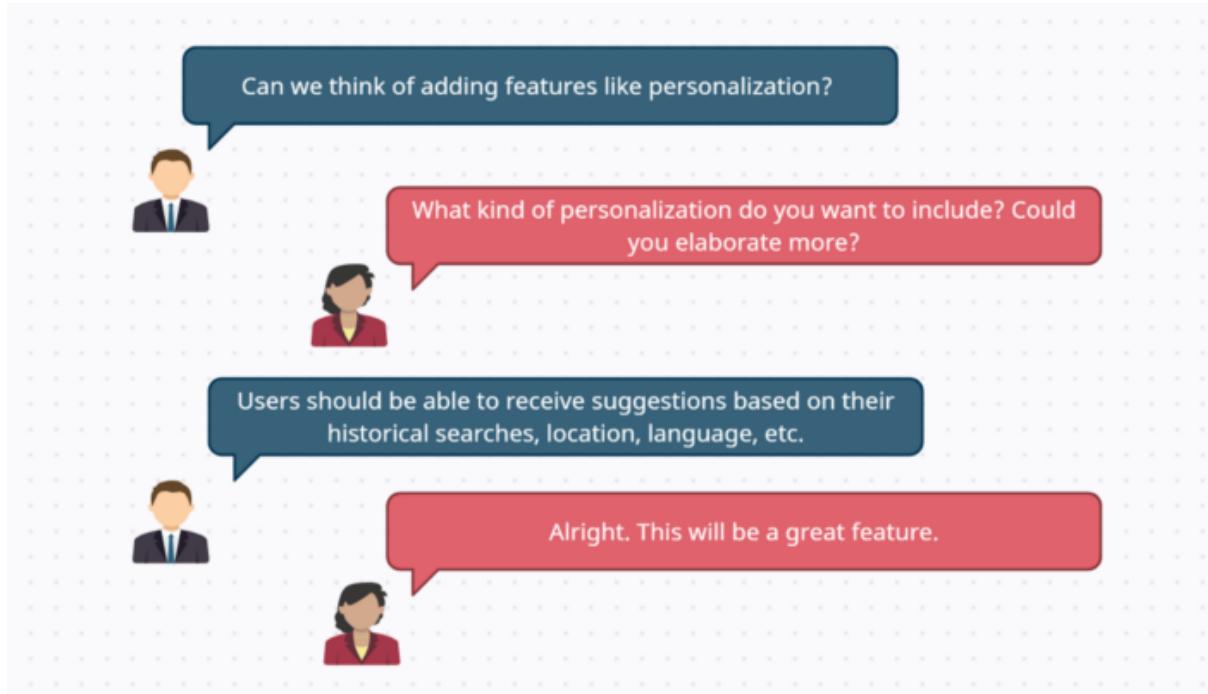


Figure 5: Question about personalization

Non-Functional Requirements

Non-functional requirements characterise the quality features of a software system.

Non-functional requirements should consider below topics:

- Specifying scaling requirements
- Specifying latency requirements
- Specifying availability requirements

Questions/answers related to non-functional requirements:

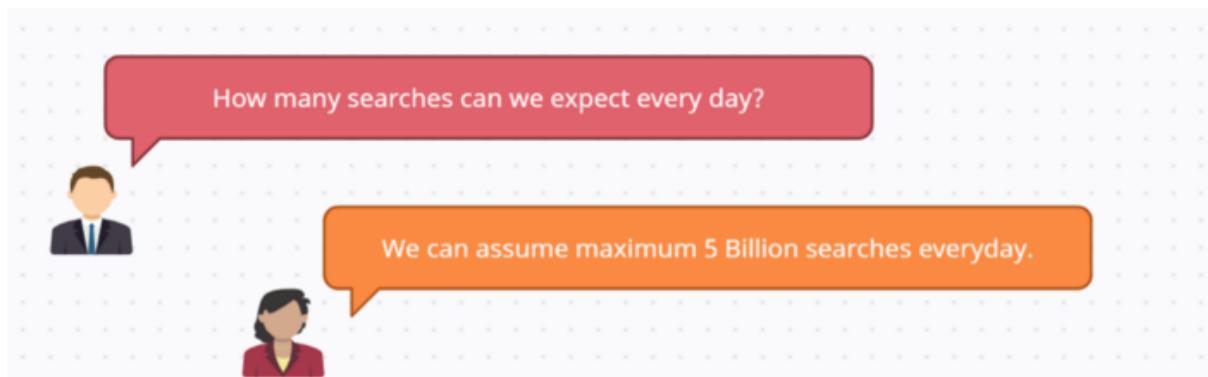
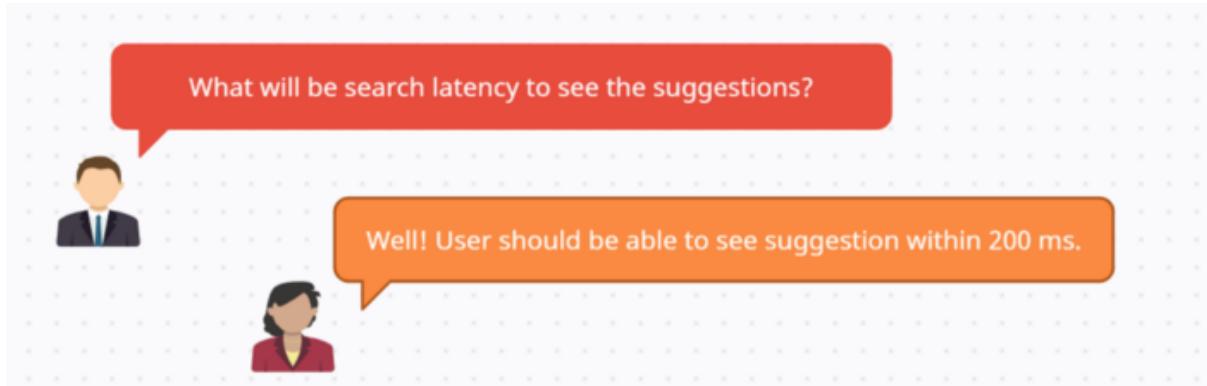
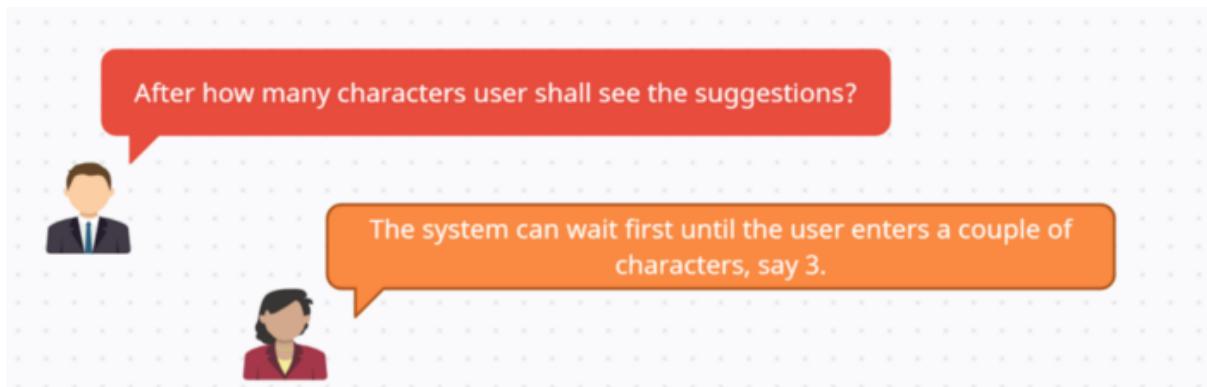


Figure 6: Question about the number of searches every day

Search latency for a user.



Question 7: Question-related to the search latency to see the suggestion



Question 8: Question about the minimum number of characters to be typed by users to get suggestions



Question 8: Availability and scalability

Scalability Estimation

This is an important consideration of any software before the start of development. This includes the performance measurements concerning the number of user requests. Based on this measurement system can be scaled up or scaled-down. It included hardware, software and a database.

Traffic Estimates

Let's say the system is expecting **5 Billion** searches every day. So, searches per second will be:

5B searches / 36400 sec a day \approx 60K searches/sec

Question 9: Traffic Estimation

There will be many duplicated queries in 5 Billion queries. So, we can assume 20% of 5 Billion is the unique queries.

Daily unique queries:

20% of 5B searches = 1000000 searches/day (Unique)

Question 10: Daily Unique Queries

Since there will be a bunch of duplicates in 5 billion queries, we can consider that only 20% of these will be unique. Let's assume we have 100 million unique queries on which we like to construct an index.

Storage Estimates

Can we consider that on an average each query consists of 3 words and each word is made up of on average 5 characters?



Ummm! Yeah, this will be great for the storage estimates.



Figure 11: Size estimates of each query

As per our conversation, let's take the values below:

Average number of words in each query = 3

Average number of characters in each word = 5

Average size of query = $3 * 5 = 15$ characters

Figure 12: Average size of the query

Let's calculate database storage size based on the above size of query:

Size of each character = 2 bytes

Size for each query = 15 characters * 2 bytes = 30 bytes

Size for the 100 million queries = 100 million * 30 bytes = ~ 3 GB

Figure 13: Database storage size for the 100 million queries

Every day new unique queries will keep coming and for these, we need to estimate the database size too.

Lets consider we have 2% unique new queries every day. So, how long are we going to store these queries in db?



You can consider for 1 year.



Figure 14: Storage estimates for the new unique queries

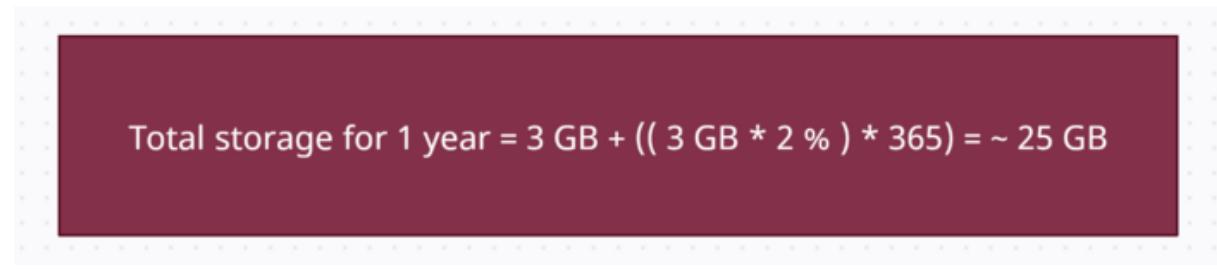

$$\text{Total storage for 1 year} = 3 \text{ GB} + ((3 \text{ GB} * 2\%) * 365) = \sim 25 \text{ GB}$$

Figure 15: Total storage for 1 year

Bandwidth Estimates

We are expecting around **60K** searches per second so, the total incoming data for the typeahead suggestion service will be:

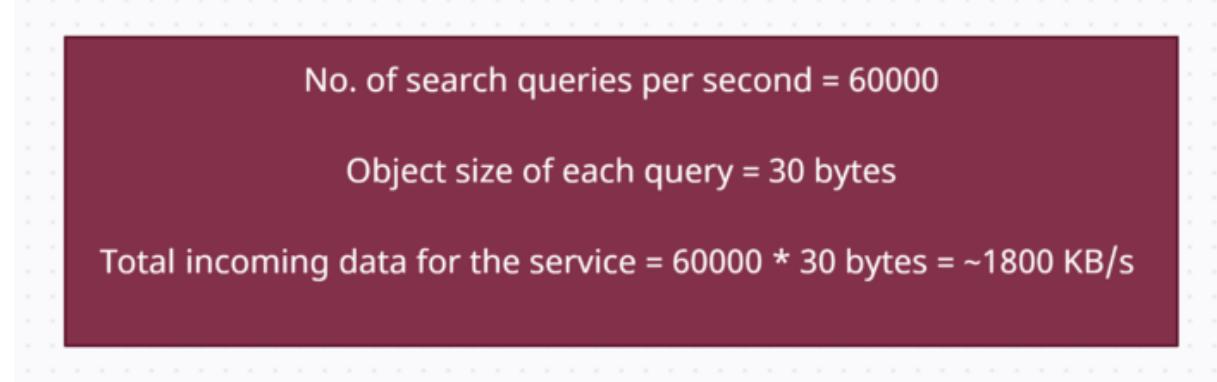

$$\text{No. of search queries per second} = 60000$$
$$\text{Object size of each query} = 30 \text{ bytes}$$
$$\text{Total incoming data for the service} = 60000 * 30 \text{ bytes} = \sim 1800 \text{ KB/s}$$

Figure 16: Bandwidth estimates per second

Algorithm Discussion

This problem is moreover related to lots of string and their searches. So, it is very important to think of a way to construct terms and think of a data structure for these in such a way that these can easily match with the user's prefixes.

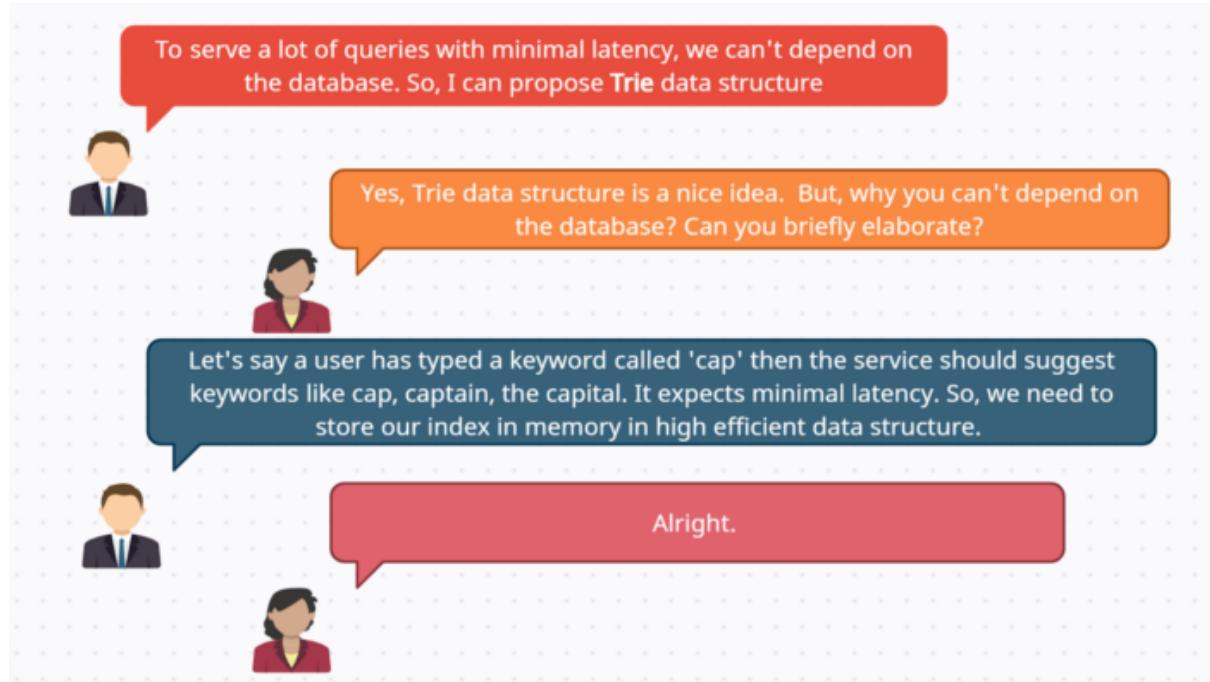


Figure 17: Why Trie Data Structure

Trie is an incredibly impressive and reasonable data structure that is based on the prefix of a string. It is used to characterise the retrieval of data. It stores strings that can be visualized like a graph. It consists of nodes and edges.

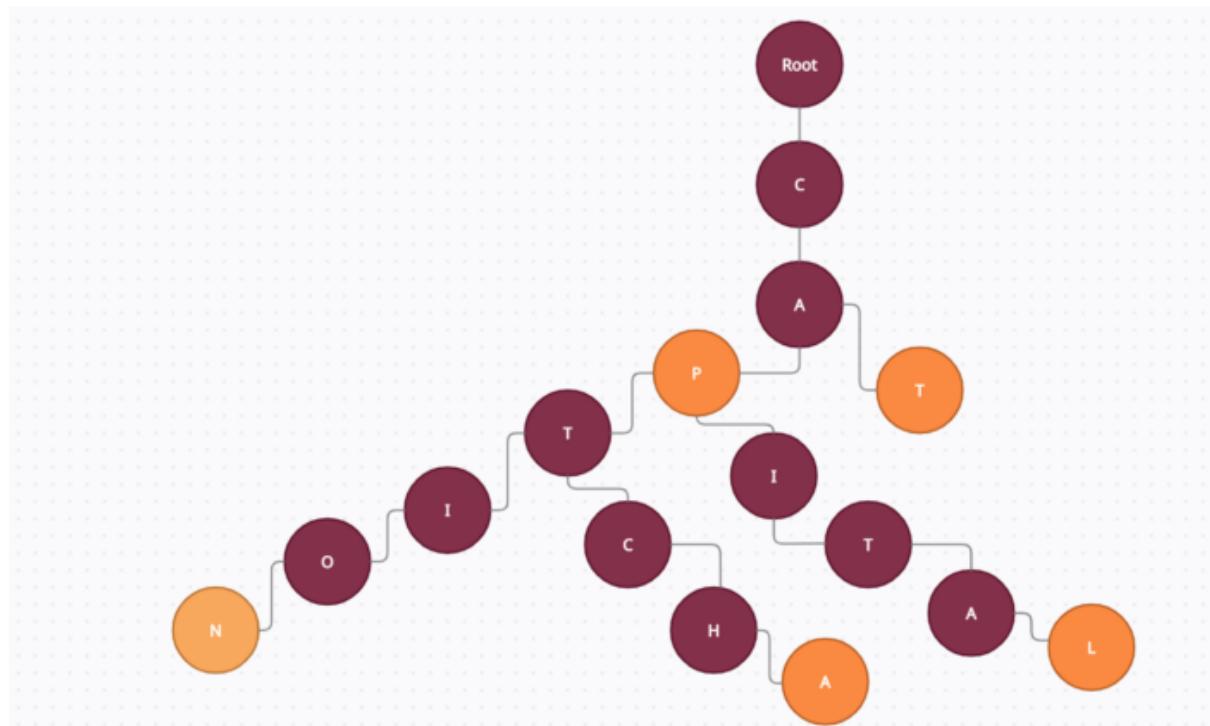


Figure 18: Storage of search query in Trie Data Structure

For example, user types the word '**CAP**' then it will suggest the below keywords:

- CAP
- CAPTION
- CAPTCHA
- CAPITAL

On typing of the word, CAP service takes this as prefix and traverses to trie to construct all top 10 keywords.

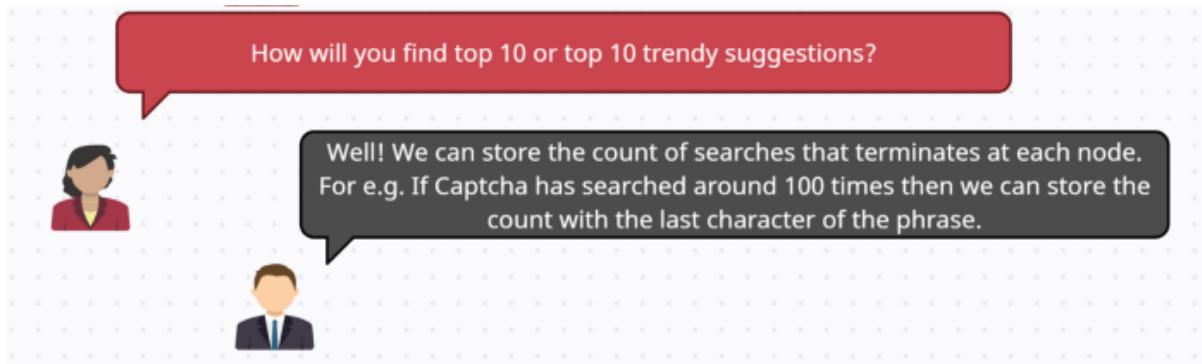


Figure 18: Top or trendy keywords or suggestions

Count each suggestions node.

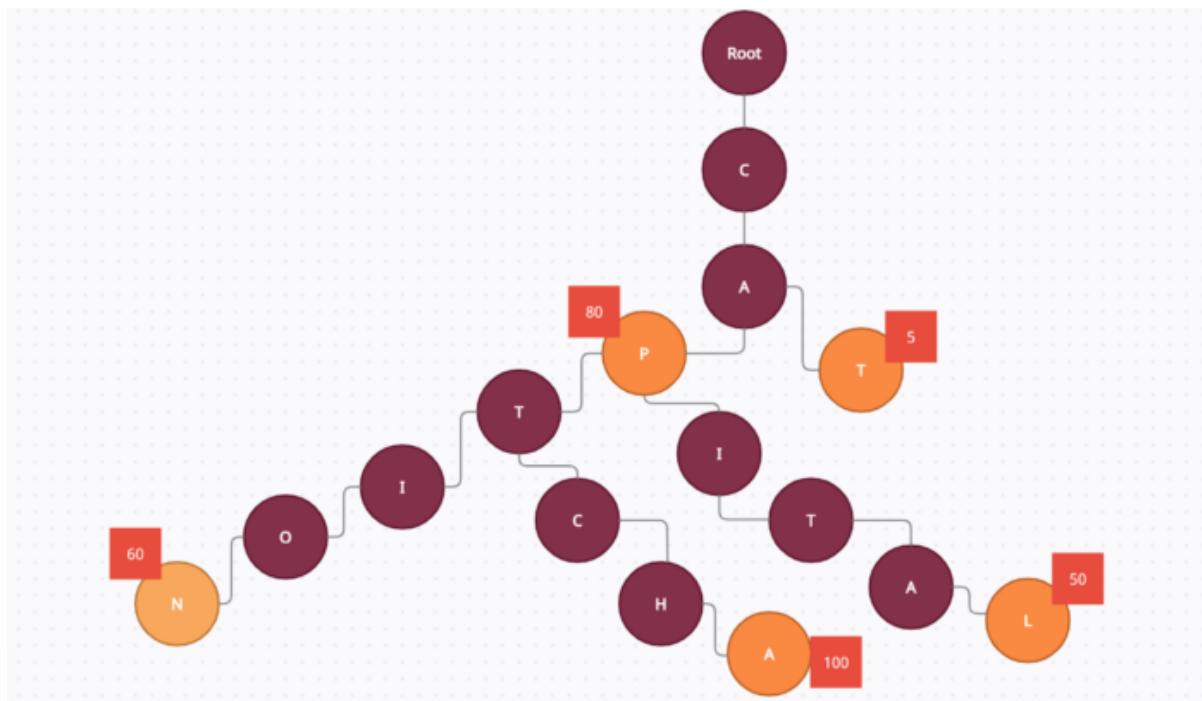


Figure 19: Count for each suggestion node

Here, in the above graph each searched suggestion node is maintaining its count so as per search frequency it will list suggestions like the below:

- Captcha
- Cap

- Caption
- Capital

To discover the top suggestions for an assigned prefix, we can traverse the sub-tree beneath it.

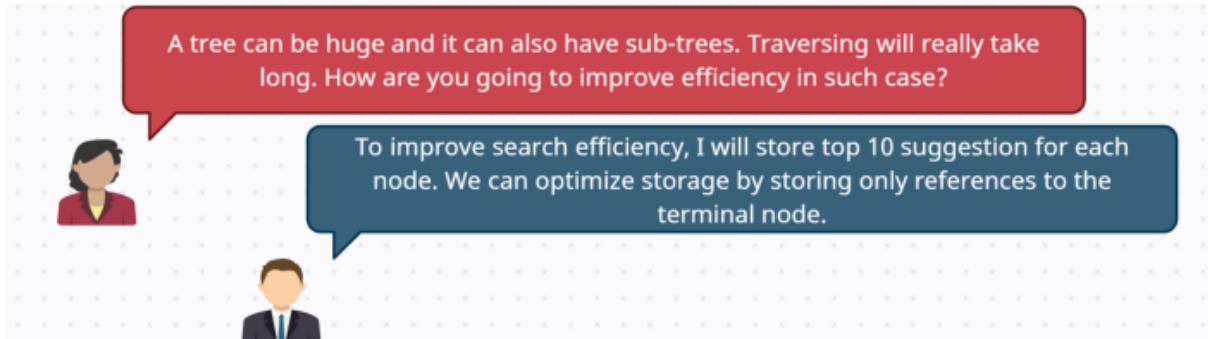


Figure 20: Improvement of efficiency

Question about the construction and traversal of tree nodes.

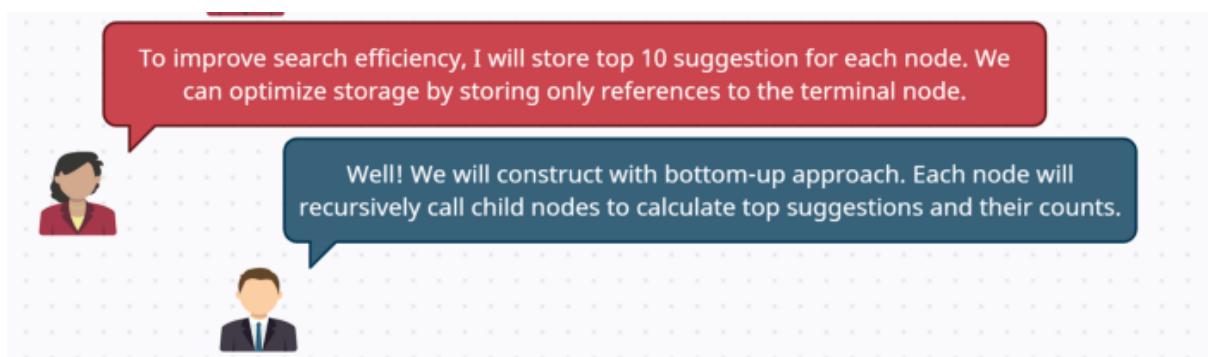


Figure 21: Question about the construction and traversal of tree nodes.

Question-related to user's location, language, etc.

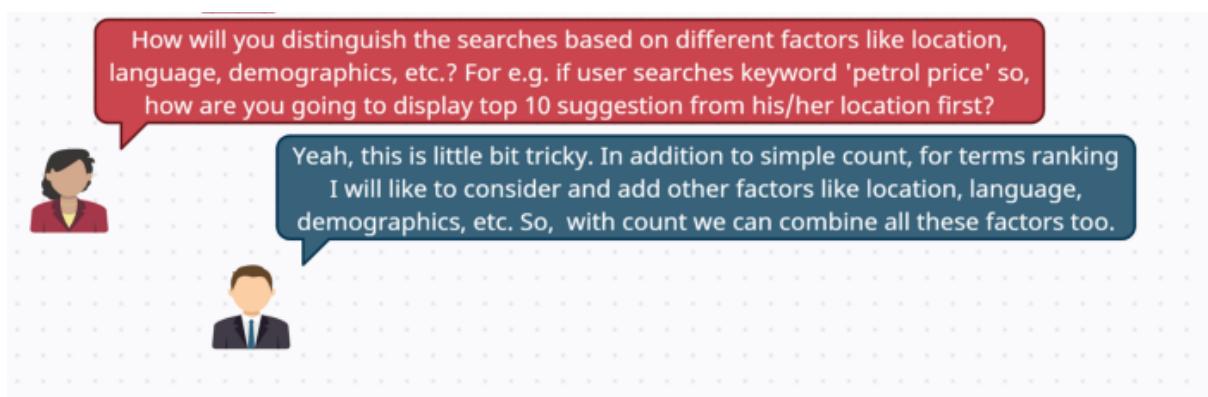


Figure 22: Question-related to user's location, language, etc.

For example, the behaviour of google is as below: I have searched the petrol price keyword and its top suggestion is from my location and country.

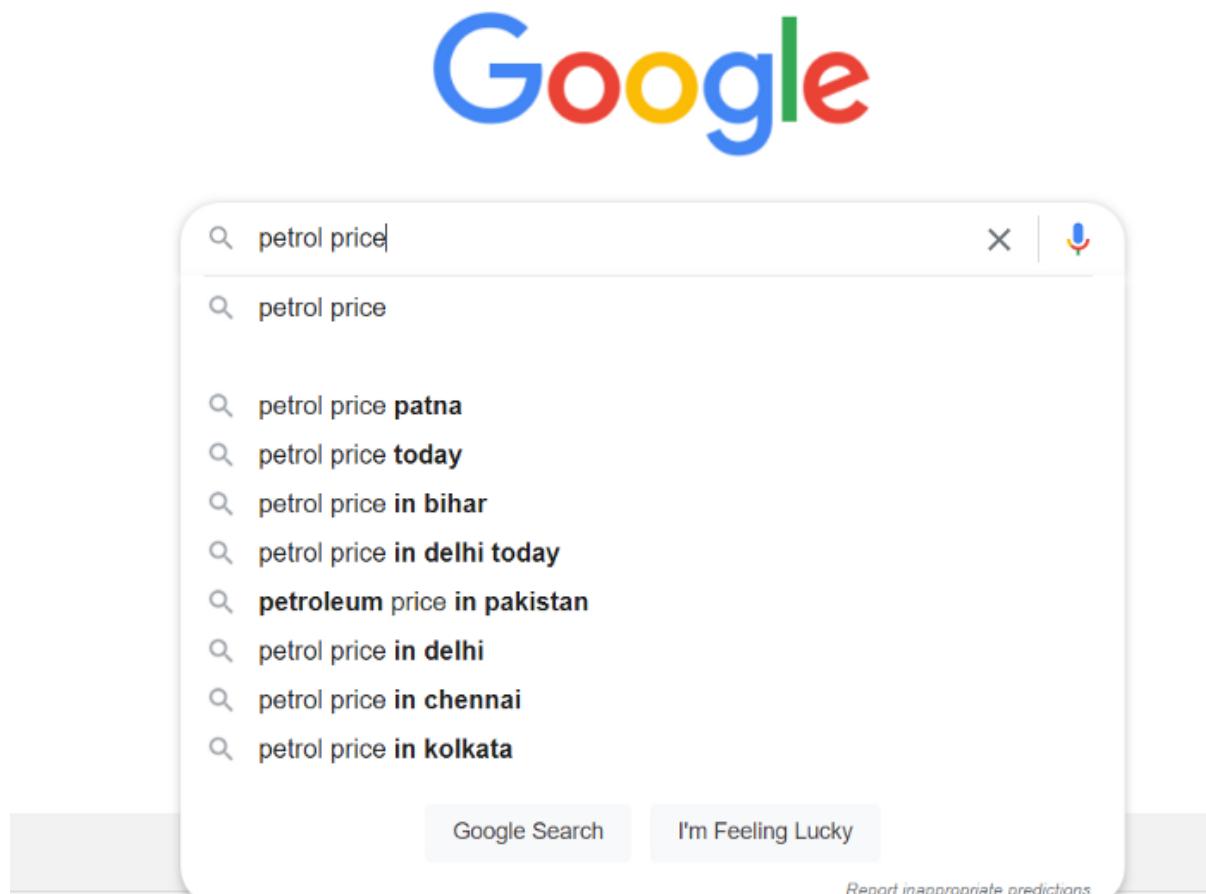


Figure 23: Google's suggestions based on the user's location

Update Strategy of Trie for Every Searches

Question related to the update of the trie.

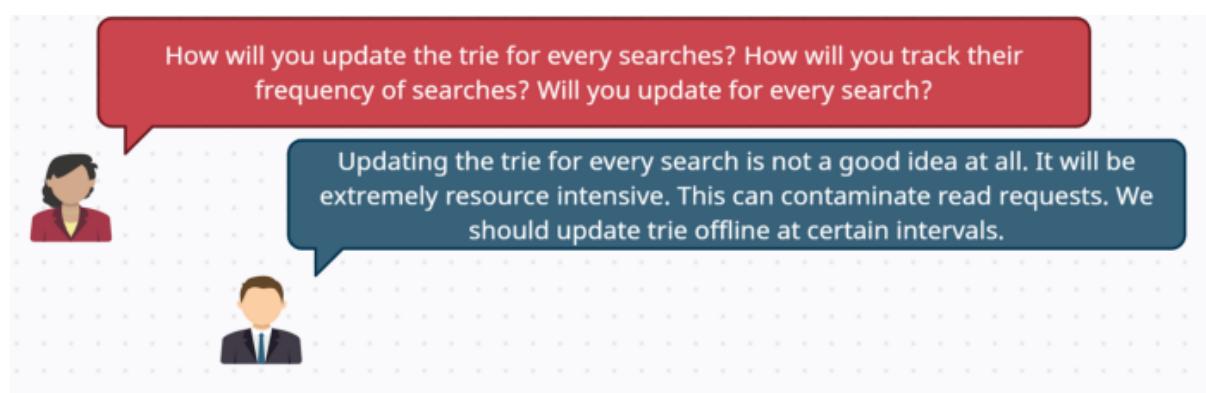


Figure 24: How will you update the trie

Calculate frequency.

We can use Map Reduce (MR) to process data periodically (may be every hour). Map Reduce will also calculate the frequencies of all search terms.



Figure 24: Calculation of frequency by Map Reduce

To update frequencies, there is a wonderful approach called “**Exponential moving average (EMA)**”.

Exponential moving average (EMA)

An exponential moving average (EMA), is also known as an exponentially weighted moving average (EWMA). It is a first-order infinite impulse response filter that applies weighting factors that decrease exponentially. It places a higher weight on recent data than on older data.

The EMA for a series Y may be calculated recursively:

$$S_t = \begin{cases} Y_0, & t = 0 \\ \alpha Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 0 \end{cases}$$

Where:

- The coefficient α represents the degree of weighting decrease, a constant smoothing factor between 0 and 1. A higher α discounts older observations faster.
- Y_t is the value at a time period t .
- S_t is the value of the EMA at any time period t .

Figure 25: EMA

Benefits from the EMA calculation:

- This will give more weight to the latest data.

Database Selection for the Permanent Storage

If the server goes down so we need to permanently store trie in some database so that trie can be rebuilt.

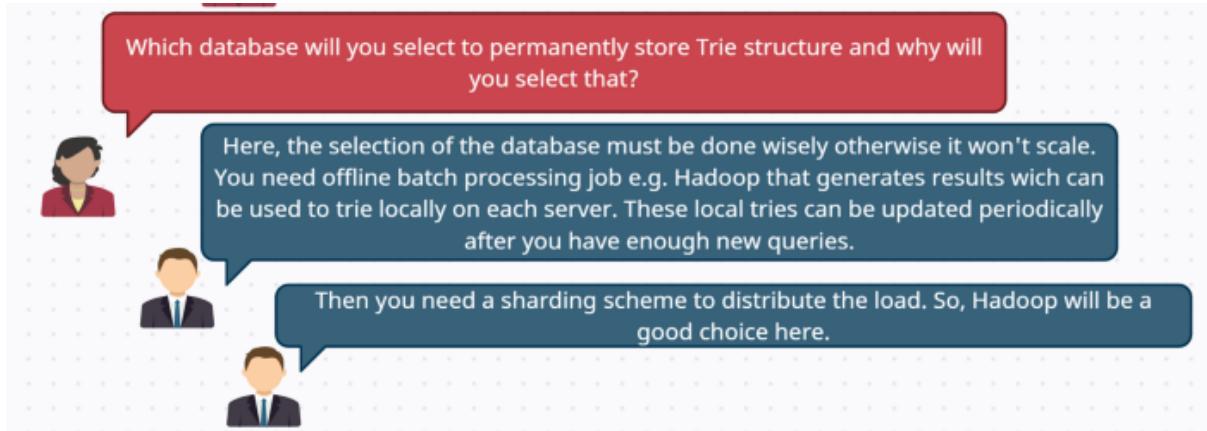


Figure 26: Storage of Trie structure

The storage scheme of Trie is as below:

Let's take below Trie.

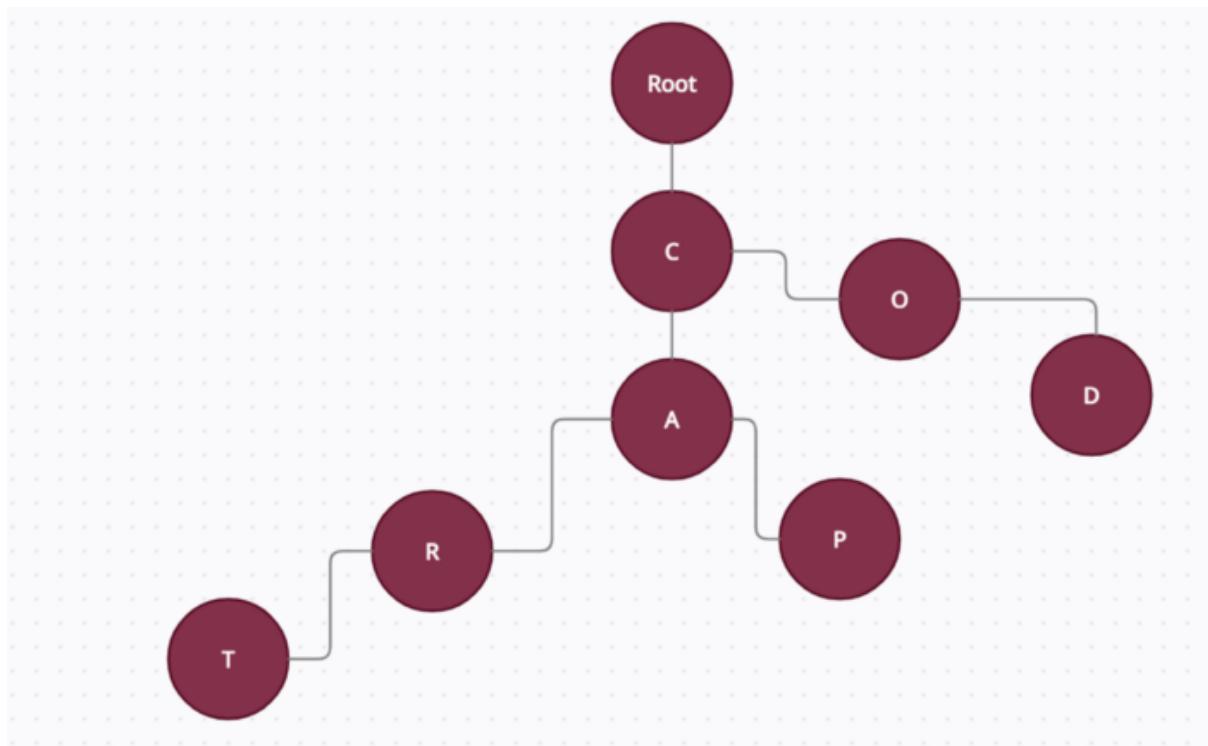


Figure 27: Trie structure

We can store the above Trie like below:

C2, A2, R1, T, P, O1, D

Figure 28: Storage structure of above trie

High-Level System Design

High-level design (HLD) defines the architecture that would be utilised to develop a system. The architecture diagram furnishes an outline of an entire system, determining the main components that would be developed for the product and their interfaces.

Below high-level system design.

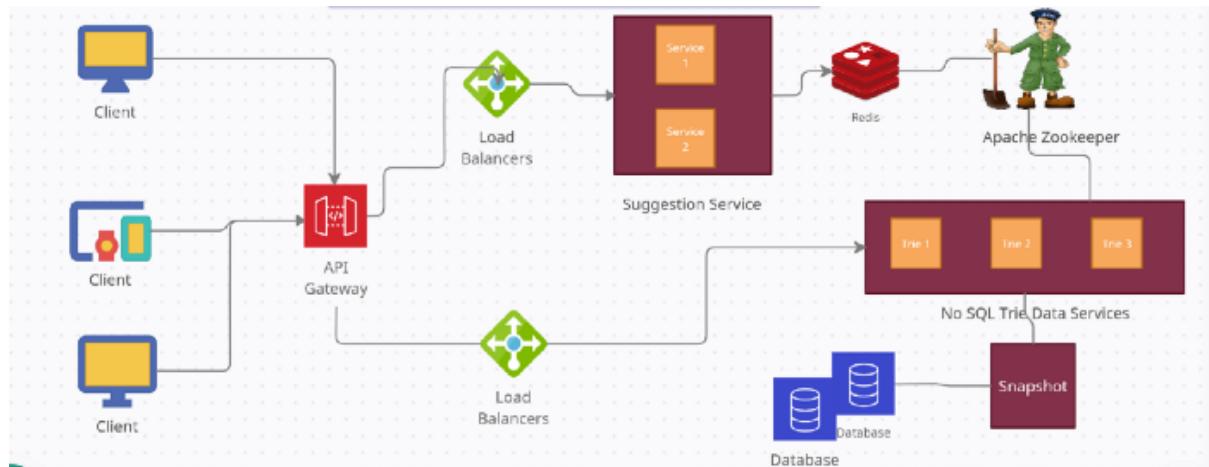


Figure 29: High-level design

Question-related to the Scaling of Trie Database.

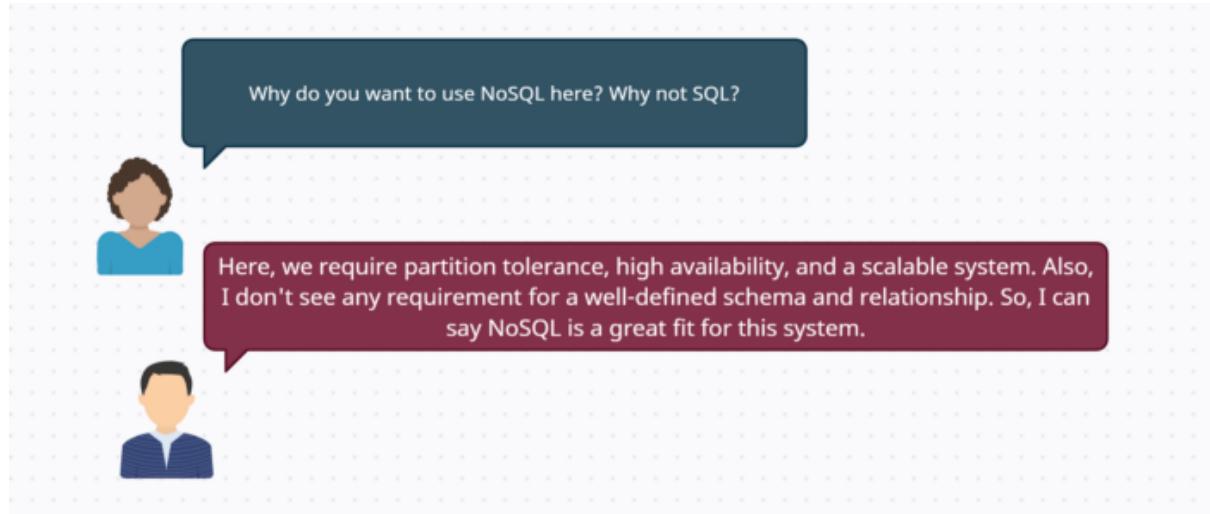


Figure 30: SQL Vs NoSQL

Question-related to Zookeeper.

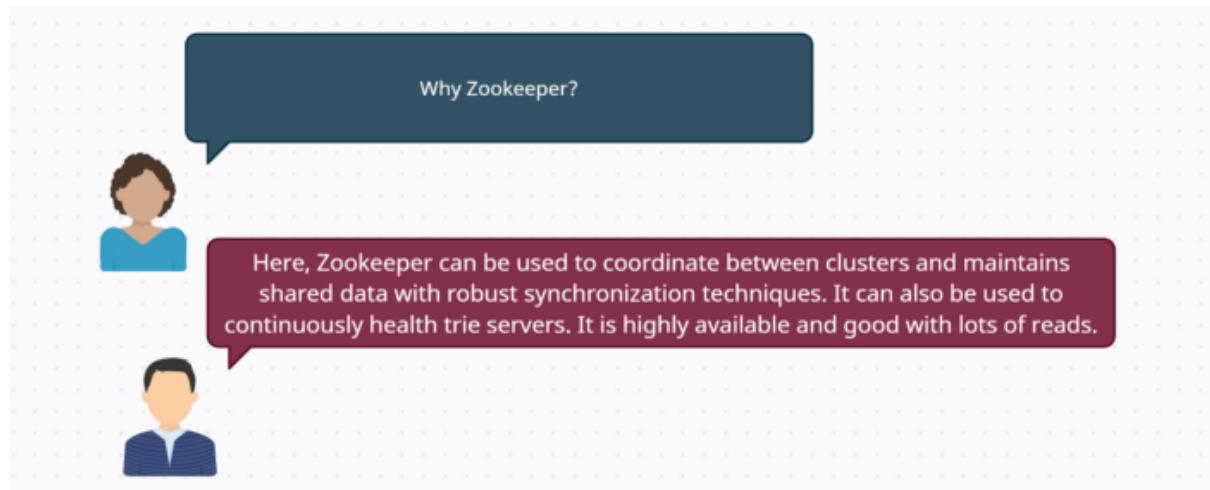


Figure 31: Why Zookeeper?

Question-related to Cache present in the design.

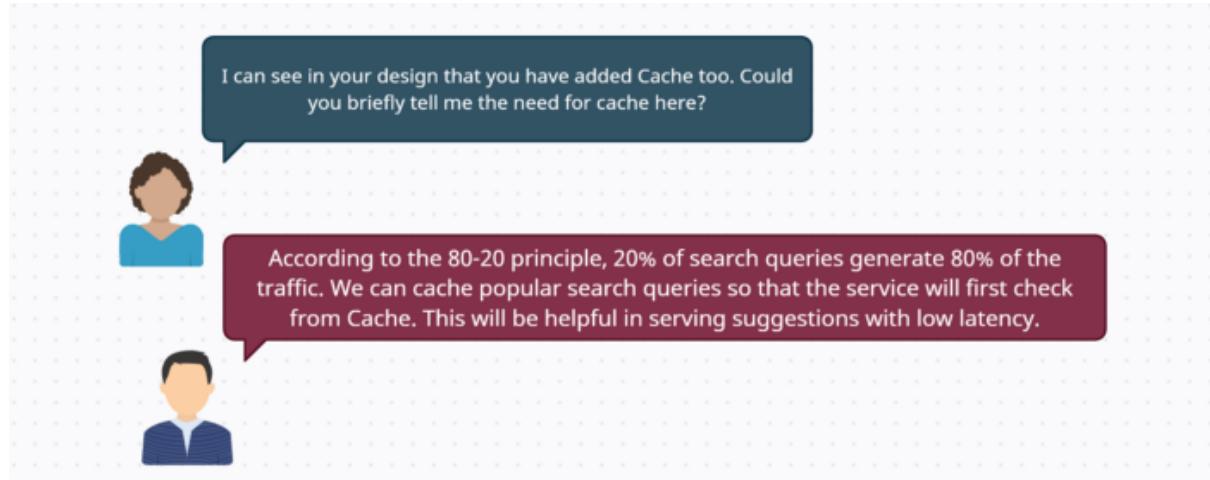


Figure 32: Why Cache?

Question-related to snapshot.

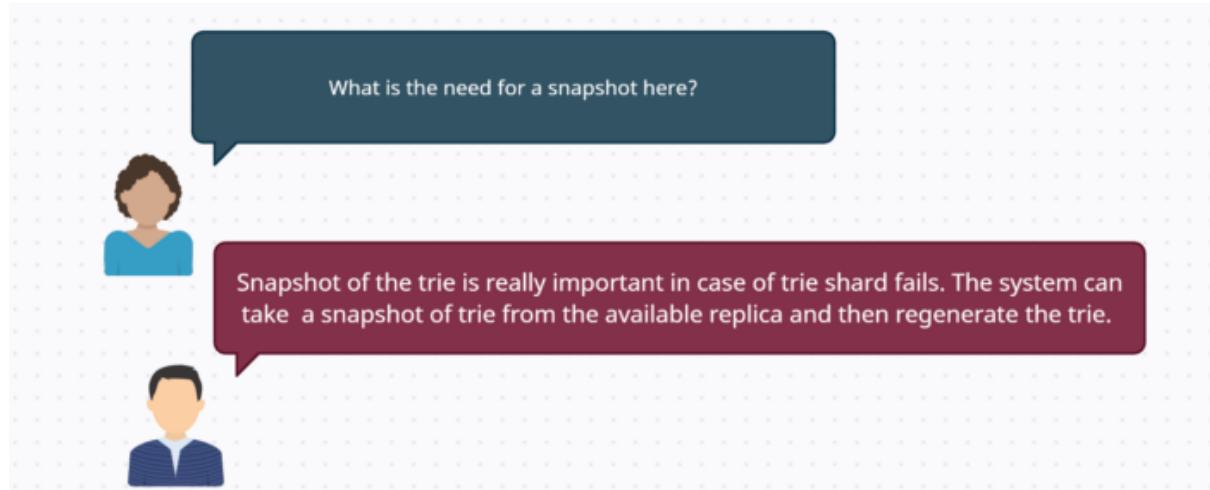


Figure 33: Why snapshot here?

Question-related to eventual consistency in DB.

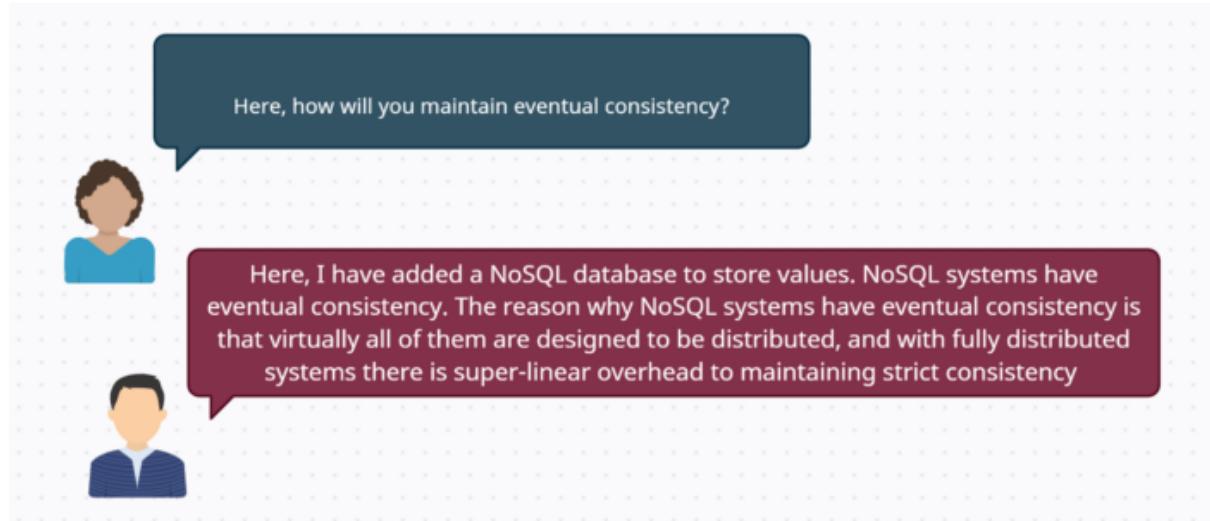


Figure 34: How will you maintain eventual consistency in Db.

Question-related to fault tolerance.

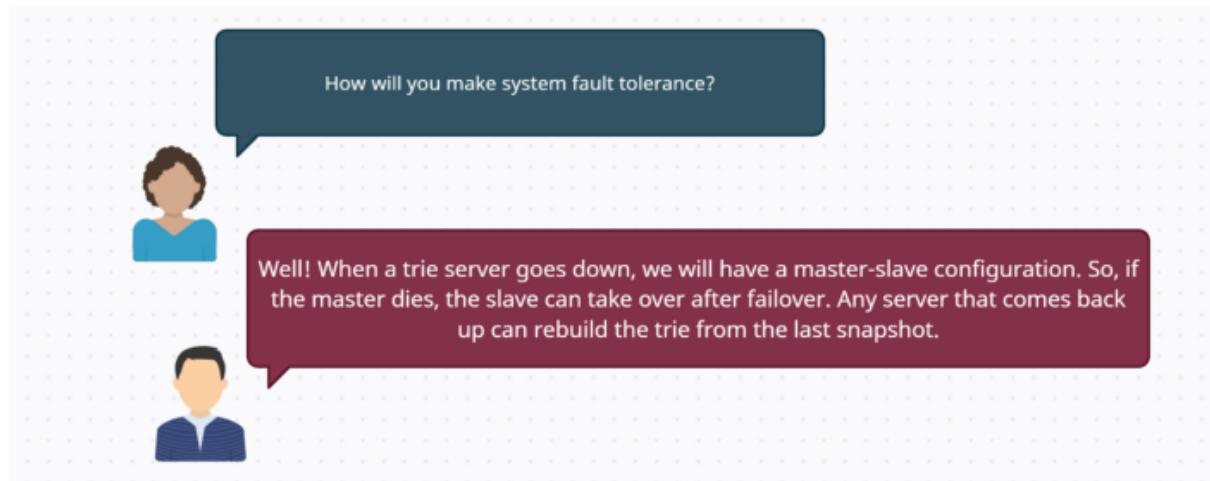


Figure 35: How will you make system fault tolerance?

Low-Level Design

Low-level design (LLD) is a component-level design approach that pursues a step-by-step advancement process. This approach can be used for designing data structures, required software architecture, source code and ultimately, performance algorithms.

Storage of precomputed data

The key-value structure in the cache will be as prefix(Key)- suggestions(Value).

Below key-value structure:

Prefix	Suggestion
C	{CAT, CAP, Capital, Cash}
CAT	{Category, Caterpillar, Cataract}
CAP	{Caption, Capital, Capsule}
CA	{Canada, California, Cash}

Figure 36: Key-Value store

This key-value store stores the top suggestion. Herewith every suggestion, we can add counts too.

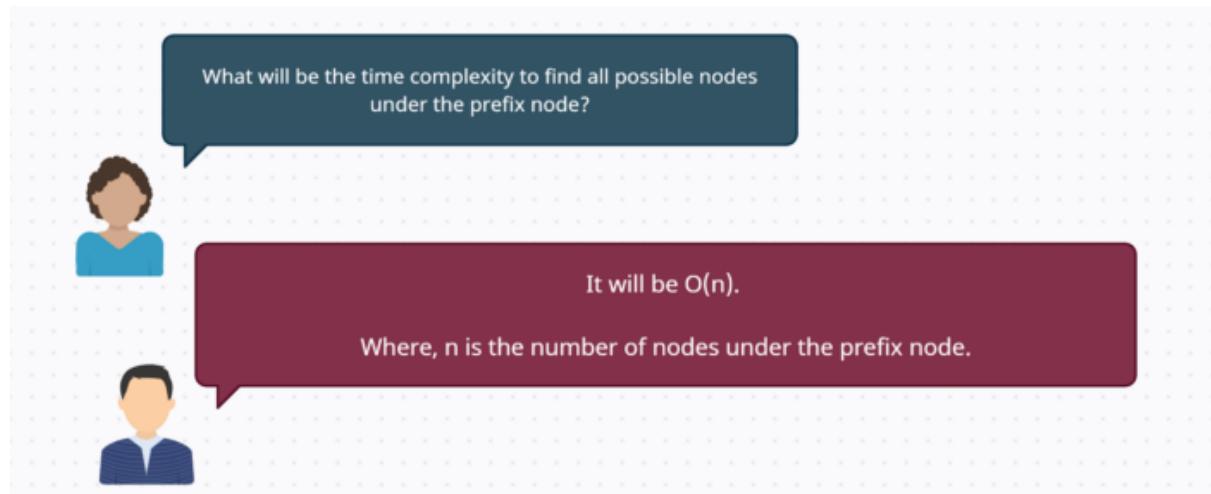


Figure 37: Time complexity

API Design

Below are basic APIs.

Get a List of top 10 suggestions.

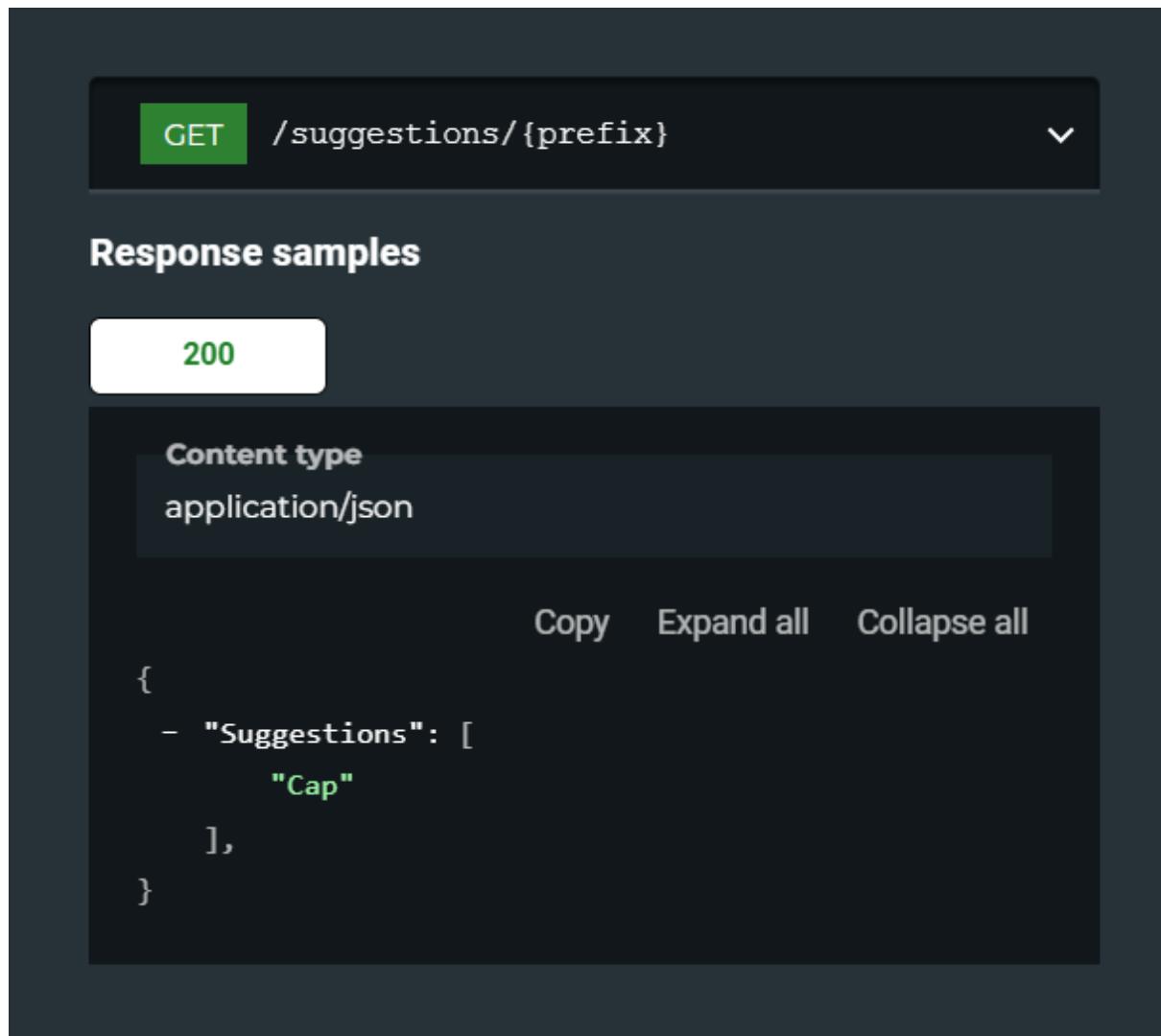


Figure 38: Get suggestion

A new unique trending query that has been searched above a particular threshold will be stored in the database.

POST /suggestions ▾

Request samples

Payload

Content type
application/json

Copy Expand all Collapse all

```
{  
  - "query": [  
    "string"  
  ],  
}
```

Response samples

201

Content type
application/json

Created

Figure 39: Store new unique query

A basic class diagram is as below:

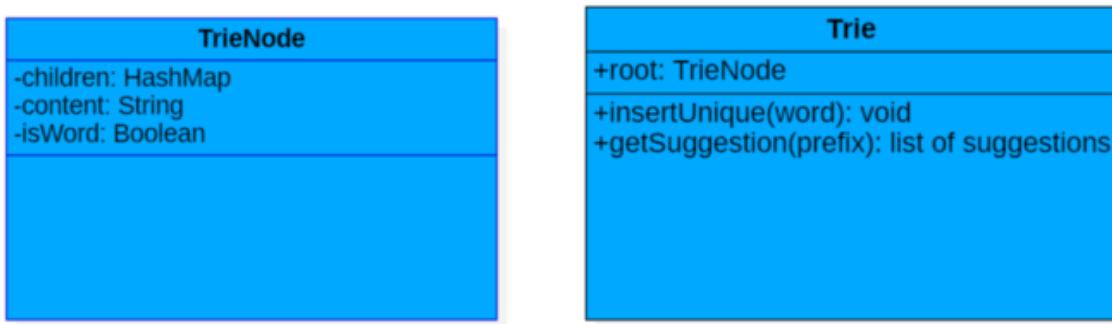


Figure 40: Class diagram

Consideration

Few considerations should be in mind while designing a system, service or database:

- How do you ensure that the data remain correct and complete, even when things go wrong?
- How do you provide consistently good performance to clients, even when parts of your system are degraded?
- How do you scale to handle an increase in load?
- What does a good API for the service look like?

How do you make your system reliable?

- Design a system in a way that minimizes the opportunities for error. For example, Well designed abstraction, APIs, etc.
- Decouple the places where people make the most mistakes from the place where they can cause failure.
- Test thoroughly at all levels.
- Allow quick and easy recovery from human errors.
- Setup detailed and clear monitoring such as performance metrics and error rates.

Summary to scale system to support millions of users

- Keep web tier stateless.
- Build redundancy at every tier.
- Cache data as much as you can.
- Support multiple data centres.
- Host static assets in CDN.
- Scale your data tier by sharding.
- Split tiers into individual services.
- Monitor your system and use automation tools.

Solution 2:

This design was implemented using Docker Compose¹, and you can find the source code here: <https://github.com/lopespm/autocomplete>

Requirements

The design has to accommodate a Google-like scale of about 5 billion daily searches, which translates to about 58 thousand queries per second. We can expect 20% of these searches to be unique, this is, 1 billion queries per day.

If we choose to index 1 billion queries, with 15 characters on average per query² and 2 bytes per character (we will only support the english locale), then we will need about 30GB of storage to host these queries.

Functional Requirements

- Get a list of top phrase suggestions based on the user input (a prefix)
- Suggestions ordered by weighting the frequency and recency of a given phrase/query³

The main two APIs will be:

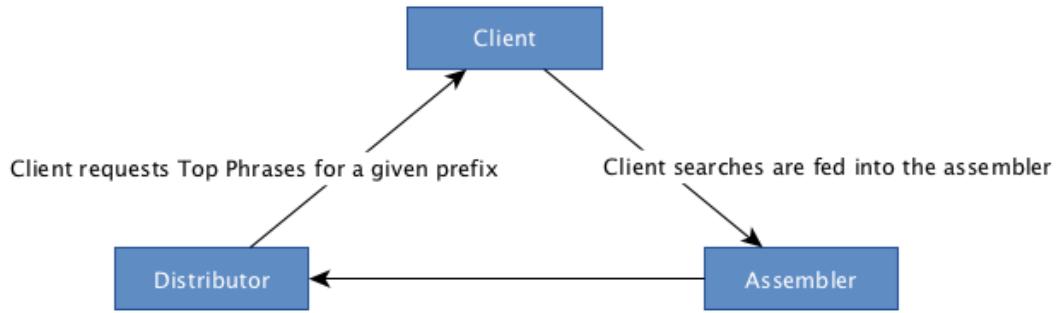
- *top-phrases(prefix)*: returns the list of top phrases for a given prefix
- *collect-phrase(phrase)*: submits the searched phrase to the system. This phrase will later be used by the assembler to build a data structure which maps a prefix to a list of top phrases

Non-Functional Requirements

- *Highly Available*
- *Performant* - response time for the top phrases should be quicker than the user's type speed (< 200ms)
- *Scalable* - the system should accommodate a large number of requests, while still maintaining its performance
- *Durable* - previously searched phrases (for a given timespan) should be available, even if there is a hardware fault or crash

Design & Implementation

High-level Design

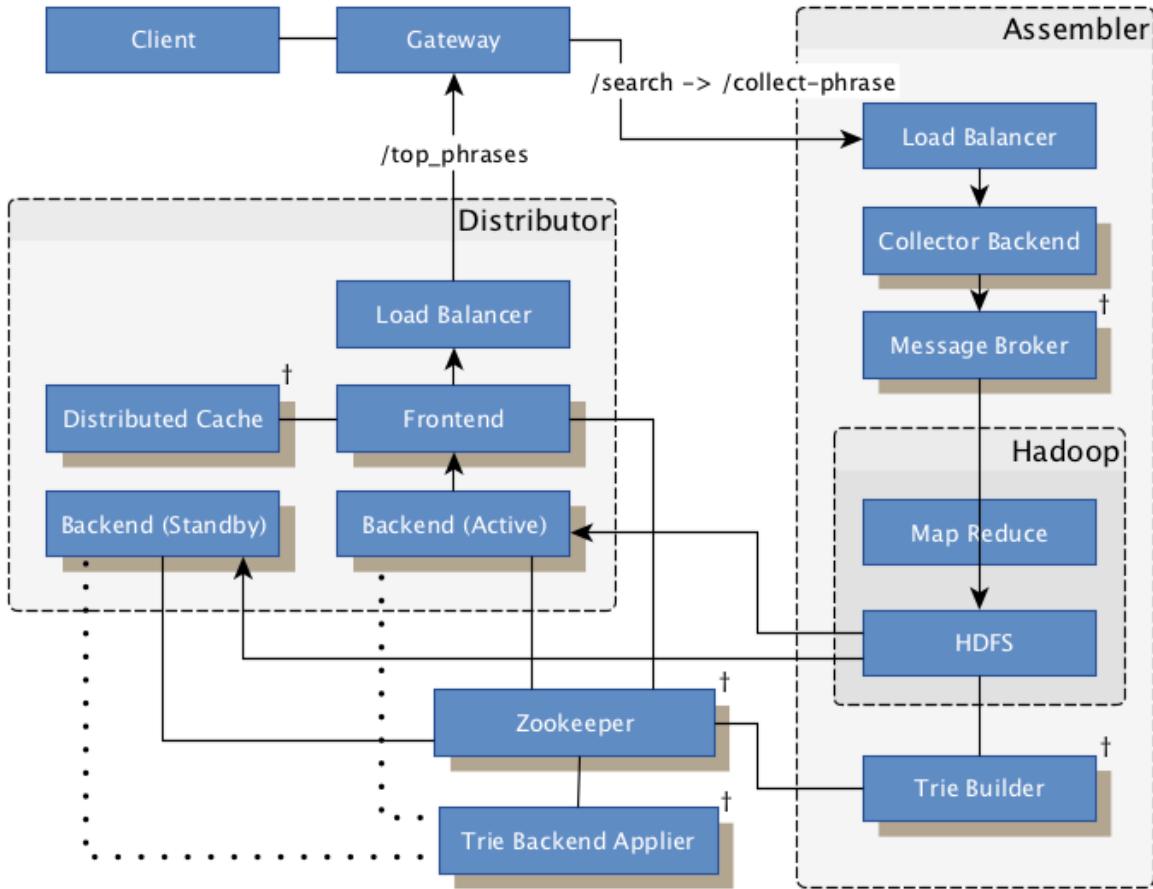


The Distributor uses the data structure built by the assembler, which maps a prefix to a list of Top Phrases

The two main sub-systems are:

- the distributor, which handles the user's requests for the top phrases of a given prefix
- the assembler, which collects user searches and assembles them into a data structure that will later be used by the distributor

Detailed Design



† – In this implementation, these services have a single instance

This implementation uses off-the-shelf components like kafka (message broker), hadoop (map reduce and distributed file system), redis (distributed cache) and nginx (load balancing, gateway, reverse proxy), but also has custom services built in python, namely the trie distribution and building services. The trie data structure is custom made as well.

The backend services in this implementation are built to be self sustainable and don't require much orchestration. For example, if an active backend host stops responding, its corresponding ephemeral znode registry eventually disappears, and another standby backend node takes its place by attempting to claim the position via an [ephemeral sequential](#) znode on zookeeper.

Trie: the bedrock data structure

The data structure used by, and provided to the distributor is a [trie](#), with each of its prefix nodes having a list of top phrases. The top phrases are referenced using the [flyweight pattern](#), meaning that the string literal of a phrase is stored only once. Each prefix node has a list of top phrases, which are a list of references to string literals.

As we've seen before, we will need about 30GB to index 1 billion queries, which is about the memory we would need for the above mentioned trie to store 1 billion queries. Since we

want to keep the trie in memory to enable fast lookup times for a given query, we are going to partition the trie into multiple tries, each one on a different machine. This relieves the memory load on any given machine.

For increased availability, the services hosting these tries will also have multiple replicas. For increased durability, the serialized version of the tries will be available in a distributed file system (HDFS), and these can be rebuilt via map reduce tasks in a predictable, deterministic way.

Information Flow

Assembler: collect data and assemble the tries

1. Client submits the searched phrase to the gateway via
`http://localhost:80/search?phrase="a user query"`
2. Since the search service is outside the scope of this implementation, the gateway directly sends the searched phrase to the collector's load balancer via
`http://assembler.collector-load-balancer:6000/collect-phrase?phrase="a user query"`
3. The collector's load balancer forwards the request to one of the collector backends via `http://assembler.collector:7000/collect-phrase?phrase="a user query"`
4. The collector backend sends a message to the *phrases* topic to the message broker (kafka). The key and value are the phrase itself ⁴
5. The Kafka Connect HDFS Connector assembler.kafka-connect dumps the messages from the *phrases* topic into the `/phrases/1_sink/phrases/{30 minute window timestamp}`⁵ folder ⁶
6. Map Reduce jobs are triggered ⁷: they will reduce the searched phrases into a single HDFS file, by weighting the recency and frequency of each phrase ⁸
 1. A TARGET_ID is generated, according to the current time, for example
`TARGET_ID=20200807_1517`
 2. The first map reduce job is executed for the K^9 most recent `/phrases/1_sink/phrases/{30 minute window timestamp}` folders, and attributes a base weight for each of these (the more recent, the higher the base weight). This job will also sum up the weights for the same phrase in a given folder. The resulting files will be stored in the `/phrases/2_with_weight/2_with_weight/{TARGET_ID}` HDFS folder
 3. The second map reduce job will sum up all the weights of a given phrase from `/phrases/2_with_weight/2_with_weight/{TARGET_ID}` into `/phrases/3_with_weight_merged/{TARGET_ID}`
 4. The third map reduce job will order the entries by descending weight, and pass them through a single reducer, in order to produce a single file. This file is placed in `/phrases/4_with_weight_ordered/{TARGET_ID}`
 5. The zookeeper znode `/phrases/assembler/last_built_target` is set to the `TARGET_ID`
7. The Trie Builder service, which was listening to changes in the `/phrases/assembler/last_built_target` znode, builds a trie for each partition¹⁰, based on the `/phrases/4_with_weight_ordered/{TARGET_ID}` file. For example, one trie may cover the prefixes until *mod*, another from *mod* to *racke*, and another from *racke* onwards.

1. Each trie is serialized and written into the `/phrases/5_tries/{TARGET_ID}/{PARTITION}` HDFS file (e.g. `/phrases/5_tries/20200807_1517/mod\racke`), and the zookeeper znode `/phrases/distributor/{TARGET_ID}/partitions/{PARTITION}/trie_data_hdfs_path` is set to the previously mentioned HDFS file path.
2. The service sets the zookeper znode `/phrases/distributor/next_target` to the `TARGET_ID`

Transferring the tries to the Distributor sub-system

1. The distributor backend nodes can either be in active mode (serving requests) or standby mode. The nodes in standby mode will fetch the most recent tries, load them into memory, and mark themselves as ready to take over the active position. In detail:
 1. The standby nodes, while listening to changes to the znode `/phrases/distributor/next_target`, detect its modification and create an [ephemeral sequential](#) znode, for each partition, one at a time, inside the `/phrases/distributor/{TARGET_ID}/partitions/{PARTITION}/nodes/` znode. If the created znode is one of the first R znodes (R being the number of replica nodes per partition ¹¹), proceed to the next step. Otherwise, remove the znode from this partition and try to join the next partition.
 2. The standby backend node fetches the serialized trie file from `/phrases/5_tries/{TARGET_ID}/{PARTITION}`, and starts loading the trie into memory.
 3. When the trie is loaded into memory, the standby backend node marks itself as ready by setting the `/phrases/distributor/{TARGET_ID}/partitions/{PARTITION}/nodes/{CREATED_ZNODE}` znode to the backend's hostname.
2. The Trie Backend Applier service polls the `/phrases/distributor/{TARGET_ID}/` sub-znodes (the `TARGET_ID` is the one defined in `/phrases/distributor/next_target`), and checks if all the nodes in all partitions are marked as ready
 1. If all of them are ready for this next `TARGET_ID`, the service, in a single transaction, changes the value of the `/phrases/distributor/next_target` znode to empty, and sets the `/phrases/distributor/current_target` znode to the new `TARGET_ID`. With this single step, all of the standby backend nodes which were marked as ready will now be active, and will be used for the following Distributor requests.

Distributor: handling the requests for Top Phrases

With the distributor's backend nodes active and loaded with their respective tries, we can start serving top phrases requests for a given prefix:

1. Client requests the gateway for the top phrases for a given prefix via `http://localhost:80/top-phrases?prefix="some prefix"`
2. The gateway sends this request to the distributor's load balancer via `http://distributor.load-balancer:5000/top-phrases?prefix="some prefix"`
3. The load balancer forwards the request to one of the frontends via `http://distributor.frontend:8000/top-phrases?prefix="some prefix"`

4. The frontend service handles the request:
 1. The frontend service checks if the distributed cache (redis) has an entry for this prefix ¹². If it does, return these cached top phrases. Otherwise, continue to next step
 2. The frontend service gets the partitions for the current TARGET_ID from zookeeper (/phrases/distributor/{TARGET_ID}/partitions/ znode), and picks the one that matches the provided prefix
 3. The frontend service chooses a random znode from the /phrases/distributor/{TARGET_ID}/partitions/{PARTITION}/nodes/ znode, and gets its hostname
 4. The frontend service requests the top phrases, for the given prefix, from the selected backend via
`http://{BACKEND_HOSTNAME}:8001/top-phrases="some prefix"`
 1. The backend returns the list of top phrases for the given prefix, using its corresponding loaded trie
 5. The frontend service inserts the list of top phrases into the distributed cache (cache aside pattern), and returns the top phrases
5. The top phrases response bubbles up to the client

Zookeeper Znodes structure

Note: Execute the shell command `docker exec -it zookeeper ./bin/zkCli.sh` while the system is running to explore the current zookeeper's znodes.

- phrases
 - distributor
 - last_built_target - set to a TARGET_ID
 - assembler
 - current_target - set to a TARGET_ID
 - next_target - set to a TARGET_ID
 - {TARGET_ID} - e.g. 20200728_2241
 - partitions
 - |{partition 1 end}
 - trie_data_hdfs_path - HDFS path where the serialized trie is saved
 - nodes
 - 0000000000
 - 0000000001
 - 0000000002
 - ...
 - {partition 2 start}|{partition 2 end}
 - ...
 - {partition 3 start}
 - ...

HDFS folder structure

Note: Access <http://localhost:9870/explorer.html> in your browser while the system is running to browse the current HDFS files and folders.

- phrases
 - 1_sink *- the searched phrases are dumped here, partitioned into 30min time blocks, *
 - {e.g 20200728_2230}
 - {e.g 20200728_2300}
 - 2_with_weight - *phrases with their initial weight applied, divided by time block*
 - {TARGET_ID}
 - 3_with_weight_merged - *consolidation of all the time blocks: phrases with their final weight*
 - {TARGET_ID}
 - 4_with_weight_ordered - *single file of phrases ordered by descending weight*
 - {TARGET_ID}
 - 5_tries - *storage of serialized tries*
 - {TARGET_ID}
 - |{partition 1 end}|
 - {partition 2 start}|{partition 2 end}|
 - {partition 3 start}|

Client interaction

You can interact with the system by accessing <http://localhost> in your browser. The search suggestions will be provided by the system as you write a query, and you can feed more queries/phrases into the system by submitting more searches.

Type a search term:

A screenshot of a web-based search interface. On the left, there is a search input field containing the text "hel". To the right of the input field is a "Submit Search" button. Below the input field, a dropdown menu is open, displaying two suggestions: "hello world" and "hello", both highlighted in green. The rest of the menu area is white.

[15:0:50:806] Successfully fetched top phrases: hello world,hello

Source Code

You can get the full source code at <https://github.com/lopespm/autocomplete>. I would be happy to know your thoughts about this implementation and design.

1. Docker compose was used instead of a container orchestrator tool like Kubernetes or Docker Swarm, since the main objective of this implementation was to build and share a system in simple manner.[←](#)
2. The average length of a search query was [2.4 terms](#), and the average word length in English language is [4.7 characters](#)[←](#)
3. *Phrase* and *Query* are used interchangeably in this article. Inside the system though, only the term *Phrase* is used.[←](#)
4. In this implementation, only one instance of the broker is used, for clarity. However, for a large number of incoming requests it would be best to partition this topic along multiple instances (the messages would be partitioned according to the *phrase* key), in order to distribute the load.[←](#)
5. */phrases/1_sink/phrases/{30 minute window timestamp} folder*: For example, provided the messages A[time: 19h02m], B[time: 19h25m], C[time: 19h40m], the messages A and B would be placed into folder */phrases/1_sink/phrases/20200807_1900*, and message C into folder */phrases/1_sink/phrases/20200807_1930*[←](#)
6. We could additionally pre-aggregate these messages into another topic (using Kafka Streams), before handing them to Hadoop[←](#)
7. For clarity, the map reduce tasks are triggered manually in this implementation via *make do_mapreduce_tasks*, but in a production setting they could be triggered via cron job every 30 minutes for example.[←](#)
8. An additional map reduce could be added to aggregate the */phrases/1_sink/phrases/* folders into larger timespan aggregations (e.g. 1-day, 5-week, 10-day, etc)[←](#)
9. Configurable in *assembler/hadoop/mapreduce-tasks/do_tasks.sh*, by the variable MAX_NUMBER_OF_INPUT_FOLDERS[←](#)
10. Partitions are defined in *assembler/trie-builder/triebuilder.py*[←](#)
11. The number of replica nodes per partition is configured via the environment variable NUMBER_NODES_PER_PARTITION in docker-compose.yml[←](#)
12. The distributed cache is disabled by default in this implementation, so that it is clearer for someone using this codebase for the first time to understand what is happening on each update/step. The distributed cache can be enabled via the environment variable DISTRIBUTED_CACHE_ENABLED in docker-compose.yml[←](#)

Solution 3:

Typeahead is a real-time suggestion service which recommends terms to users as they enter text for searching. As the user types into the search box, it tries to predict the query based on the characters the user has entered, and gives a list of suggestions to complete the query. It's not about speeding up the users' search but to help the user articulate their search queries better.

1. Requirements and Goals of the System

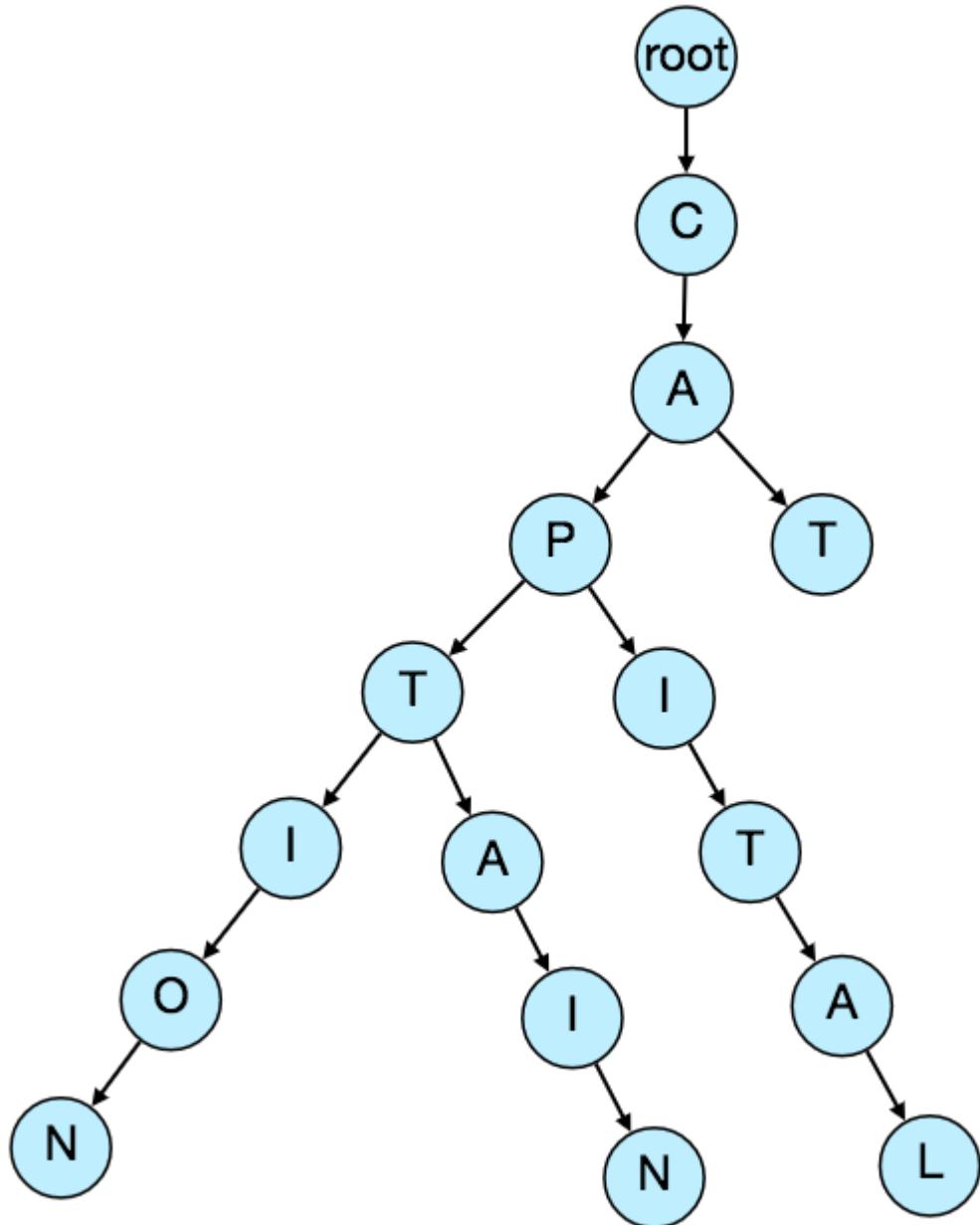
Functional requirements: As the user types in their search query, our service should suggest top 10 terms starting with whatever the user typed.

Non-functional requirements: The suggestions should appear in real-time, allowing the user to see it in about 200ms.

2. Basic System Design and Algorithm

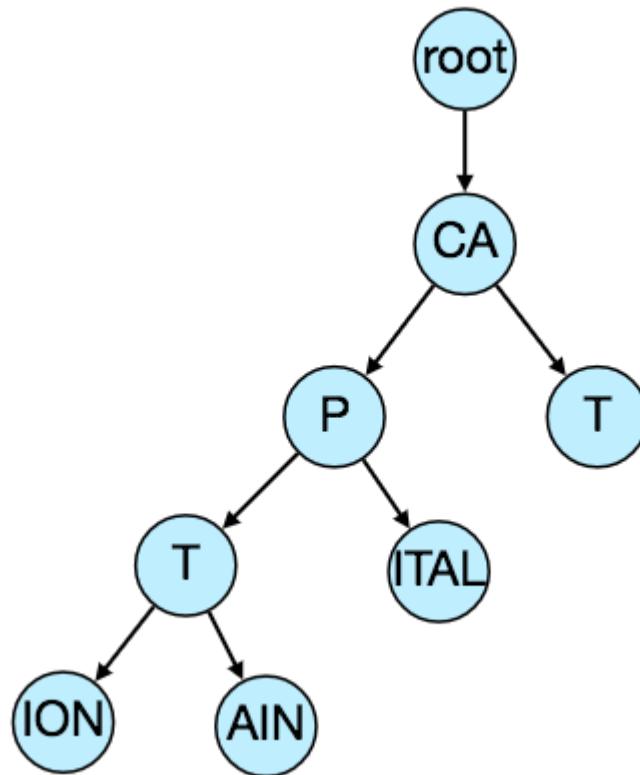
The problem to solve is that we have a lot of strings we need to store in such a way that the user can search with any prefix. The service will suggest the terms that match with the prefix. For example, if our DB contains the terms (cat, cap, captain, capital), and the user has typed in cap, then the system should suggest cap, captain and capital.

To serve a lot of queries with minimal latency, we can't depend on the DB for this; we need to store our index in memory in a highly efficient data structure – a Trie(pronounced "try").



If the user types cap, then the service can traverse the trie and go to the node **P**, to find all the terms that start with this prefix. (i.e cap-ital, cap-tain, cap-tion).

We can also merge nodes that have only one branch to save memory.



Should we have a case insensitive trie? For simplicity, let's assume our data is case insensitive.

How do we find top 10 suggestions?

We can store the count of searches that terminated at each node. For example, if the user searched about 20 times, and caption 50 times, we can store the count with the last character of the phrase. Now if the user types cap we know that the top most searched word under 'cap' is caption.

So to find the top suggestions for a given prefix, we can traverse the sub-tree under it.

Given a prefix, how long will it take to traverse its sub-tree?

Given the amount of text we need to index, we should expect a huge tree. Traversing a tree will take really long. Since we have strict latency requirements, we need to improve the efficiency of our solution.

- Store top 10 suggestions for each node. This will require extra storage.
- We can optimize our storage by storing only references to the terminal node rather than storing the entire phrase.
- Store frequency with each reference to keep track of top suggestions.

How do we build this trie?

We can efficiently build it bottom-up.

- Each parent node will recursively call child nodes to calculate top suggestions and their counts.
- The parent nodes will combine their suggestions from all their children nodes to determine their top suggestions.

how do we update the trie?

Assume 5 billion daily searches

5B searches / 36400 sec a day \approx 60K searches/sec

Updating the trie for every search will be extremely resource intensive and this can hamper our read requests too. Solution: Update the trie offline at certain intervals.

As new queries come in, log them and track their frequency of occurrences. We can log every 1000th query. For example, if we don't want to show a term that hasn't been searched for < 1000 times, it's safe to only log queries that occur the 1000th time.

We can use [Map Reduce \(MR\)](#) to process all the logging data periodically, say every hour.

- The MR jobs will calculate the frequencies of all search terms in the past hour.
- Then update the trie with the new data. We take the current snapshot of the trie and update it offline with the new terms and their frequencies.

We have two options to updating offline:

1. Make a copy of the trie from each server and update the copy offline. Once done, switch to the copy and discard the old one.
2. Master-slave configure each trie server. Update slave while master is serving incoming traffic. Once update is complete, make our slave the new master. Then later, update our old master, which can then start serving traffic too.

How do we update the frequencies of suggestions?

We are storing frequencies of suggestions for each node, so we need to update them too.

We can update only differences in frequencies, instead of recounting all search terms from scratch.

Approach: Use [Exponential Moving Average\(EMA\)](#)

- This will allow us to give more weight to the latest data.
- This means that: if we keep count for searches done the last 10 days, we need to subtract the count from the time period *no longer included* and add counts for the new time period included.

Inserting a new term and updating corresponding frequencies

After inserting a new term in the trie, we will go to the term's terminal node and increase its frequency. Since we are storing top 10 suggestions for a node, it's possible that the new term is also in the top 10 suggestions of a few other nodes. We'll therefore update the top 10 queries of those nodes too.

Traverse back from the node all the way to the root. For each parent, we check if the new query is part of the top 10. If so, we update the corresponding frequency. If not, we check if the current query's frequency is high enough to be top 10. If so, we insert the new term and remove the term with the lowest frequency.

How do we remove a term from the trie?

Let's say we have to remove some term because it's highly offensive or for some legal issue. We can do that when the periodic updates happen. Meanwhile, we can also add a filtering layer on each server which will remove any such term before sending them to users.

What could be different ranking criteria for suggestions?

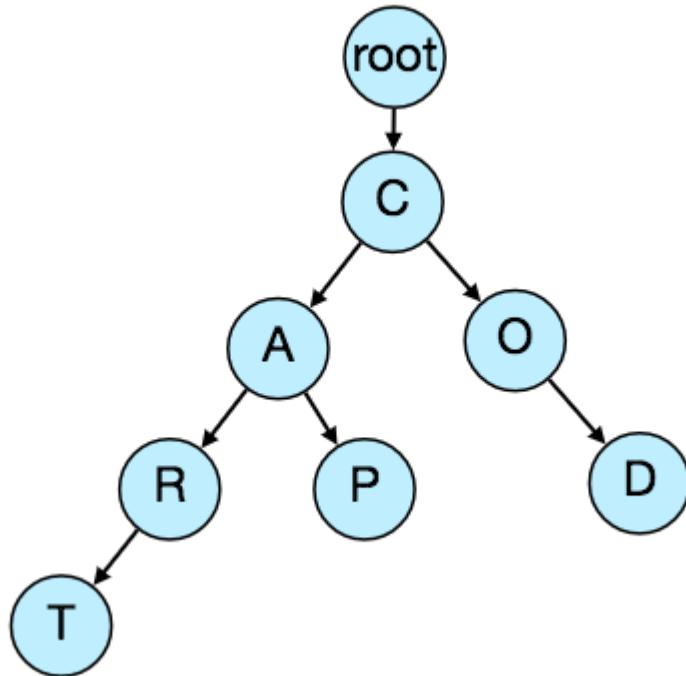
In addition to a simple count for ranking terms, we can use factors such as user's location, freshness, language, demographics, personal history, etc.

3. Permanent Storage of the Trie

How to store a trie in a file to rebuild it easily.

We enables us to rebuild a trie if the server goes down. To store, start with the root node, and save the trie level-by-level. With each node, store the character it contains and how many children it has. Right after each node, we should put all its children.

Let's assume we have the following trie:



With the mentioned storage scheme, we can store the above trie as:

C2,A2,R1,T,P,O1,D

From this, we can easily rebuild the trie.

4. Capacity Estimation

Since there will be a lot of duplicates in 5 billion queries, we can assume that only 20% of these will be unique. Let's assume we have 100 million unique queries on which we want to build an index.

Storage Estimates

Assume: average query = 3 words, each averaging 5 characters, then average query size = 15 characters.

Each character = 2 bytes

We need 30 bytes to store each query.

100 million * 30 bytes => 3 GB

We can expect this data to grow everyday, but we are also removing terms that are not being searched anymore. If we assume we have 2% unique new queries every day, total storage for a year is:

$$3\text{GB} + (3\text{GB} * 0.02\% * 365 \text{ days}) ==> 25\text{GB}$$

5. Data Partition

Although the index can fit in a single server, we still need to partition to meet our requirement of low latency and higher efficiency.

Partition based on maximum memory capacity of the server

We can store data on a server as long as it has memory available. Whenever a sub-tree cannot fit in a server, we break our partition there and assign that range to this server. We then move on to the next server and repeat the process. If Server 1 stored A to AABC, then Server 2 will store AABD onwards. If our second server could store up to BXA, the next server will start from BXB, and so on.

Server 1: A-AABC

Server 2, AABD-BXA

Server 3, BXB-CDA

...

Serve N, ...

For queries, if a user types A, we query Server 1 and 2 to find top suggestions. If they type AAA we only query Server 1.

Load Balancing

We can have a LB to store the above mappings and redirect traffic accordingly.

6. Cache

Caching the top searched terms will be extremely helpful in our service.

There will be a small percentage of queries (20/80 rule) that will be responsible for most of the traffic.

We can have separate cache servers in front of the trie servers holding most frequently searched terms and their typeahead suggestions. Application servers should check these

cache servers before hitting the trie servers to see if they have the desired searched terms. This will save us time to traverse the trie.

We can also build a Machine Learning (ML) model that can try to predict the engagement on each suggestion based on simple counting, personalization, or trending data, and cache these terms beforehand.

7. The Client

1. The client should only try hitting the server if the user has not pressed any key for 50ms.
2. If the user is constantly typing, the client can cancel the in-progress request.
3. The client can wait first until the user enters a couple of characters, say 3.
4. Client can pre-fetch data from server to save future requests.
5. Client can store recent history locally, because of the high probability of being reused.
6. The server can push some part of their cache to a CDN for efficiency.
7. The client should establish the connection to the server fast, as soon as the user opens up the search bar. So that when the user types the first letter, the client doesn't waste time trying to connect at this point.

8. Replication and Fault Tolerance.

We should have replicas for our trie servers both for load balancing and also for fault tolerance. When a trie server goes down, we already have a Master-Slave configuration. So if the master dies, the slave can take over after failover. Any server that comes back up can rebuild the trie from the last snapshot.

9. Personalization

Users will receive some typeahead suggestions based on their historical searches, location, language, etc. We can store the personal history of each user separately on the server and also cache them on the client. The server can add these personalized terms in the final set before sending it to the user. Personalized searches should always come before others.

Solution 4:

Autocomplete is a popular feature of search engines. As the user types something in the search box, this feature creates suggestions to complete the sentence for what the user has already typed. These suggestions come from the queries that users have already searched in that search engine and the popularity of these searches. A query that has been searched several times will appear among the top suggestions. A typeahead benefits the user by helping them form sentences faster. It's an important part of all search engines and many search boxes as it enhances the user experience.

Let's learn how to design an autocomplete system, also called a search typeahead, for a search box.

What Are The Requirements Of A Search Autocomplete System

Functional Requirements

1. The service should return a list of, say, 3 top suggestions, based on what the user types in the search box.
2. The suggestions are ordered according to the frequency and recency of their appearance in user searches.

Non-Functional Requirements

1. The service should be able to deliver the suggestions in real-time, as the user types in the query. This means that the latency should be very low.
2. The system should be able to scale to a large number of requests without any degradation in performance.
3. The system should be highly available.

Capacity Constraints Of The System

We need to design a search autocomplete system that works on a scale similar to Google and returns the top suggestions for a phrase that the user types. Google receives around 5 billion queries per day. Assuming about 30% of these queries are unique, this amounts to about 1.5 billion unique queries per day. If each query comprises 15 characters on average each character takes 2 bytes of storage, you'll need 45GB to store the queries made in a day.

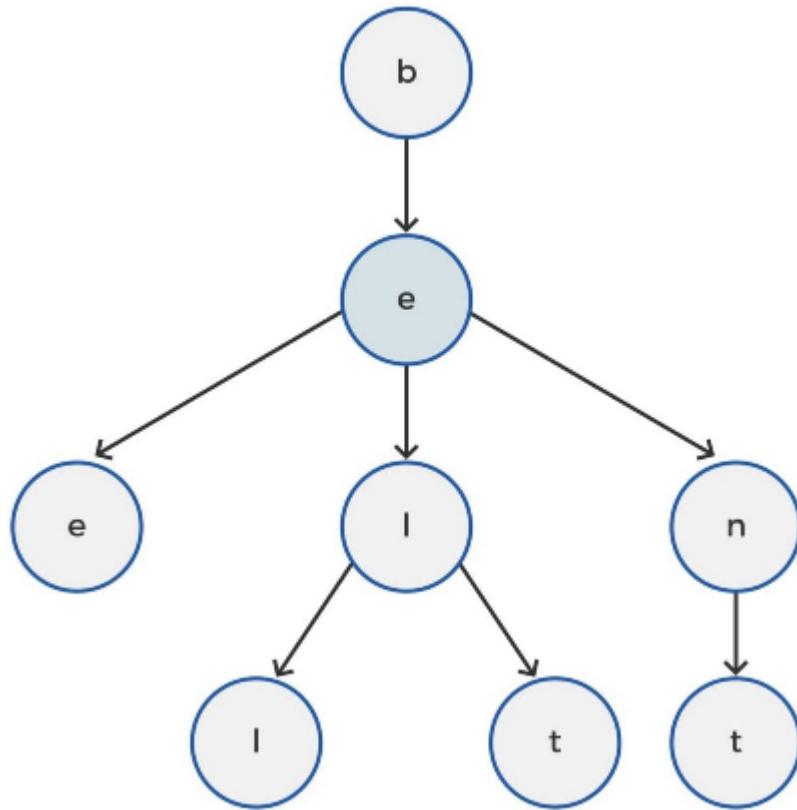
Choice Of Data Structure

The system involves the storage of a large number of *strings*. What the user enters into the search box is taken by the autocomplete system as an input prefix. This prefix is matched against the *strings* stored in the database and the most closely related, recent and popular matches are returned to the user.

The database will be queried several times each second and the system must return the response with minimum latency. The indexes need to be stored in a database that can be scanned quickly to retrieve *strings* to return to the querying user.

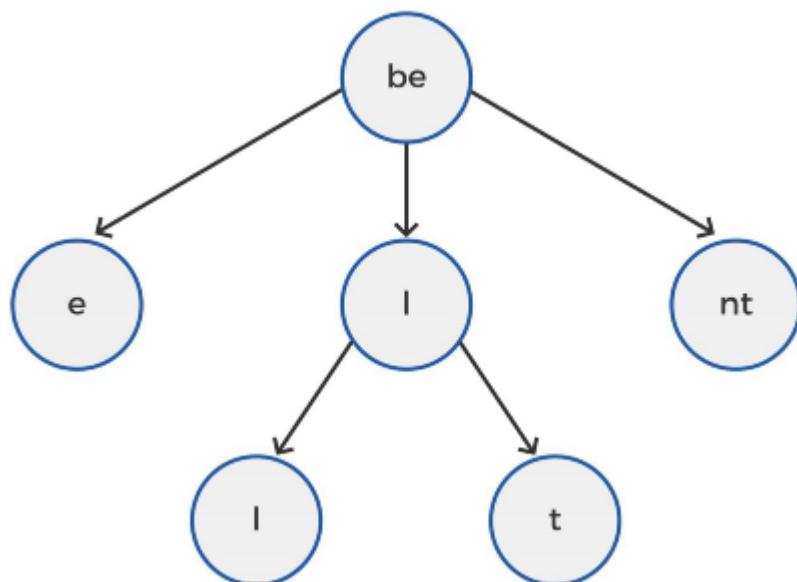
Taking into account the basic requirements of the system, the data structure ‘Trie’ is the most appropriate choice. A trie is a tree-like data structure that stores phrases in the form of a tree. Each node carries a character from the phrase in the order that it appears in the word.

Take for example the set of phrases, “be”, “bee”, “bell”, “bent”, “belt”. If this set is to be stored, this is how it will appear in a trie data structure:



If the user types “be” in the search box, our search typeahead system will scan the trie and reach the node shaded darker in the image above. All the nodes that fall below this shaded node start with the prefix “be” and are returned to the user. The user will see suggestions including terms like be-e, be-l, be-nt, be-lt and all other terms stored in the trie that start with the prefix “be”.

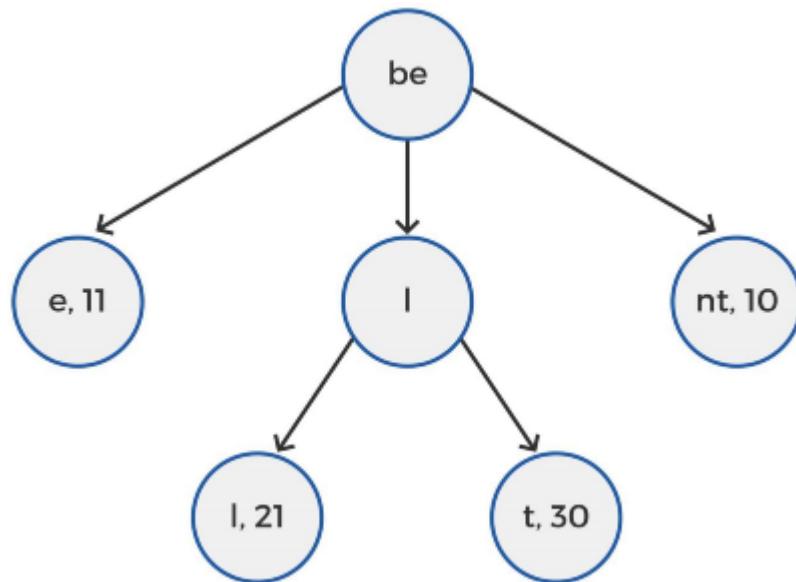
To save space and the time that our system takes to traverse the database, the trie can also combine nodes as one where only a single branch exists. For example, a space and time-efficient model of the above trie can be:



Tracking Top Searches

The typeahead system keeps track of the top searches and returns the top suggestions to the user based on the prefix that the user enters. Suppose our system returns the top 3 terms for every prefix that the user enters. For this, we will also store the number of times that each term was searched. For example, let's assume that the users searched for "bell" 21 times, "bee" 11 times, "bent" 10 times and "belt" 30 times, this information will also need to be stored by the system in order to return the top 3 searches to the user.

So the number of times a term is searched can be stored in the node where that term terminates. The terminating node for each term will also store the number of searches along with the characters. This is how our trie will appear with terms and their ranks stored in it:



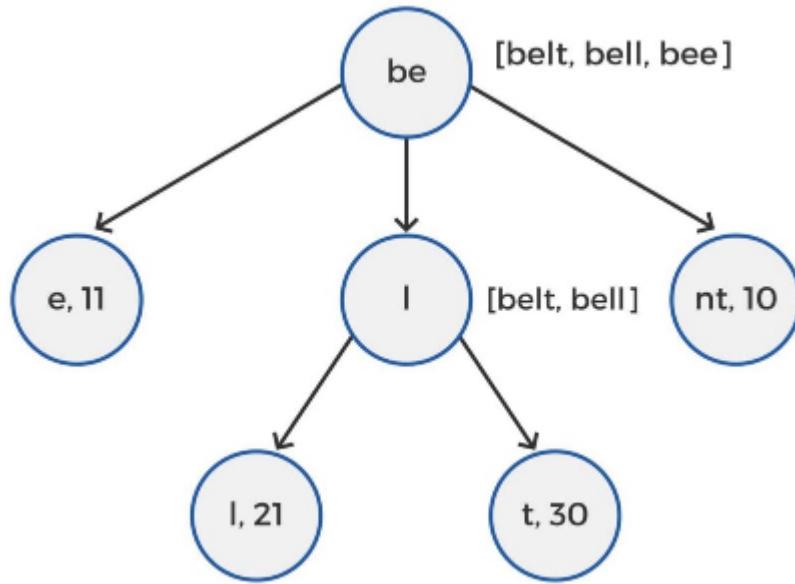
Now if the user types "be", the system will locate the root node for "be" and traverse the tree for the child nodes under it. Comparing all the terms originating from the root node, the top 3 results, "belt", "bell" and "bee" (with the highest number of searches) are returned to the user. Since the term "belt" was searched the most times (30 times), it appears on top, "bell", with 21 searches appears next and "bee" with 11 searches appears last. If the user picks "bell" from the suggestions and searches for it, the number of searches against "bell" will be incremented and updated to 22 in the terminating node for "bell".

Reducing Latency In Traversing The Trie

Our search autocomplete system has strict latency requirements since we need to create a real-time experience in returning suggestions to the users as they type the prefix. Though we reduced the time to traverse the trie by combining nodes with single branches and reducing the number of levels, it is possible to make the system even more efficient.

One way to increase efficiency is to precompute and save the top 3 suggestions for every prefix in the node for that prefix. This means that instead of traversing the trie each time a

user types in “be” into the search box, the system will have pre-computed, sorted and stored the solution to the prefix “be”, i.e. “belt”, “bell” and “bee” inside the node that carries “be”.



The image above shows how the suggestions for prefix “be” are stored in the node for “be” so they can be instantly returned to the user just after they type “be” in the search. Since the trie in the image is just an example, it is much smaller than the ones developed for actual implementations. Assuming that there are several branches and several levels in the actual trie, we will have to precompute and store suggestions for every node. If the user types “bel”, our system will move to the second level node that carries “l” in the image, and return the suggestions stored in that node for the prefix “bel”.

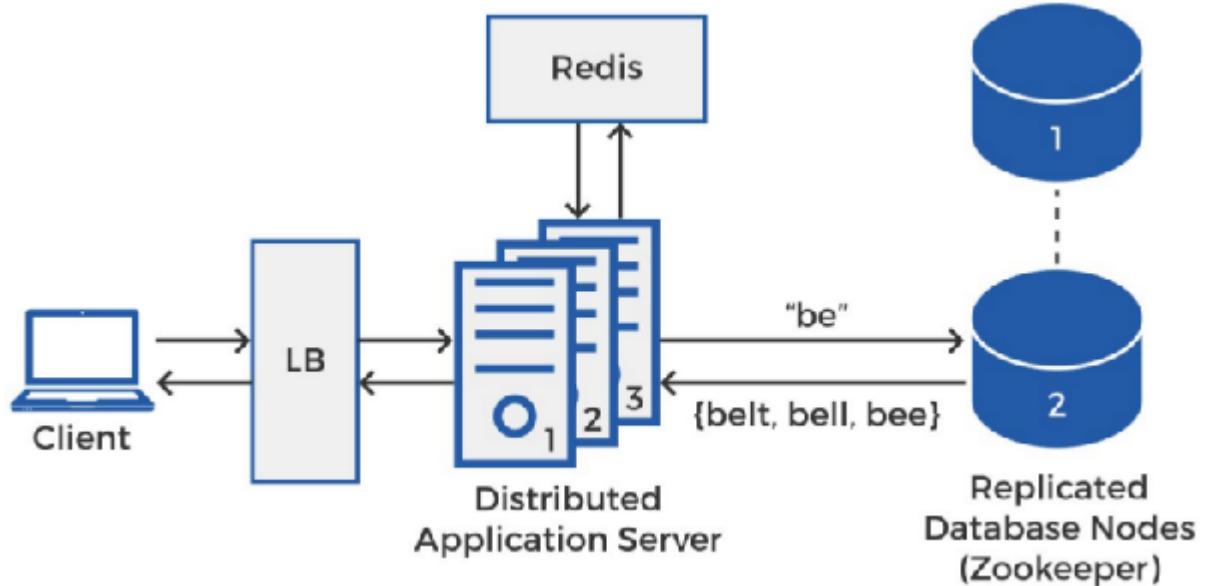
Although the system returns top 3 suggestions, it’s possible that a node towards the bottom of the trie has only 2 suggestions. A terminating node will have only 1 suggestion, i.e., the prefix of the node itself. For example, if no other term originates from the prefix “belted”, the node for the prefix “belted” will carry only one suggestion which will be returned to the user as the only suggestion.

So pre-computing the top suggestions for every prefix allows us to return them to the querying user without any delay.

Database Schema For Trie

So we know that trie is the best data structure to store suggestions for prefixes entered by the user. If the trie data structure is stored in a single server and the server goes down, the system will be unavailable. To increase availability and reliability of the system, the trie will be replicated across multiple servers. Now, if one server goes down, or is overloaded with requests, the other nodes that carry the same information can continue serving the requests. Replicating the information across multiple nodes also makes the system reliable since even if a node is lost, the autocomplete information is not lost and can be accessed from the other storage servers. The replicated instances of trie can be stored in Apache Zookeeper. Zookeeper is a good choice for storage since it is highly available and can effectively serve

high volumes of reads and some writes. You can also use a different distributed database, such as Cassandra for the purpose.



All the database nodes are stored in Zookeeper. Each database node carries all the phrases from 'a' to 'z'. The user's request to autocomplete a prefix goes through the load balancer to any of the available application servers based on round-robin, or any other mechanism, as shown in the diagram above.

Suppose the user enters "be". Once the request reaches an available application server, the application will check the Redis, or cached database to see if it has suggestions against the prefix "be". If the suggestions for "be" are not present in the Redis, the request from the application server will scan the Zookeeper instances to see where the suggestions for "be" are stored. Since all the node replicas carry the same trie for phrases from "a" to "z", any of the nodes can be randomly picked to return the suggestions for "be". So the request from the application server 1 goes to database node 2 which stores the complete trie. The trie is traversed to find the node for "be". The same trie node also stores the top 3 suggestions for the prefix "be".

The top 3 suggestions for "be" are returned by the database instance in Zookeeper to the application server. The application server first populates the Redis with this information so the next time the same request is made, it can be served faster, directly from the Redis. Next, the application server returns the response (top 3 suggestions for "be") back to the load balancer, that will in turn return it to the user. The user sees the response in the drop-down menu of the search box. He can either pick one of the responses and search for it or continue to type, in which case our system will receive further requests and serve them as they are typed by the user.

Splitting Trie Into Multiple Nodes

With a popular search engine such as Google, there can be billions of phrases to store. Storing all these phrases in a single trie is not ideal for availability, scalability or durability of

the system. A better solution is to split trie across multiple servers for a durable system and better user experience.

Suppose we split the trie into two parts and each part also has a replica for durability. Phrases starting from characters “a” to “k” are stored in Trie 1 and its replica is stored in Trie 2. Phrases starting from “l” to “z” are stored in Trie 3 and its replica is stored in Trie 4.

```
graph TD; subgraph Part1 [ ]; direction LR; a --- k; end; subgraph Part2 [ ]; direction LR; l --- z; end; Part1 --> T1[Trie 1, Trie 2]; Part2 --> T3[Trie 3, Trie 4]
```

a — k → Trie 1, Trie 2

l — z → Trie 3, Trie 4

Database (Zookeeper)

Now, when the user types something into the search box, the request goes to the load balancer, which forwards it to any of the available application servers. The application server that gets the request, searches the appropriate trie depending upon the prefix typed by the user. If the user types something that starts with “b”, the application server will access either Trie 1 or Trie 2 because both of them hold the phrases starting with “b”.

Let’s suppose our search autocomplete system needs to store even more phrases to return to the users. In this case, instead of splitting the trie into 2 parts, we can split it even further. Depending on the size of the trie, you can hold all phrases starting with “a” in Trie 1, with its replica in Trie 2, all phrases starting from “b” in Trie 3, with its replica in Trie 4, phrases starting from “c” and “d” in Trie 5, with replica in Trie 6, and so on. Theoretically, we can continue expanding the information as long as Zookeeper database can hold it.

How To Update The Trie

So the trie holds searched phrases along with the number of times they were searched, i.e., their popularity. This information will change with every new query. If “bell” was searched 30 times and a user types in “bell” in the search box, the popularity needs to change from 30 to 31.

Taking into account a popular search engine, such as Google, that serves billions of searches each day, it is resource and time-intensive to update the trie with every new query. Instead we can log the queries and their frequency in a hash table and aggregate the data,

let's say, every hour. After a day passes, we will have the aggregate frequency of each search phrase for a period of 24 hours.

Phrase	Day	Frequency
bell	1	1311
bell	2	1422
bell	3	1011
bell	4	1156
bell	5	986

For this purpose, let's assume that we have a microservice called aggregator. The aggregator collects the information from searches and dumps them in a hash table in a database such as Cassandra. We'll use this data for updating our trie.

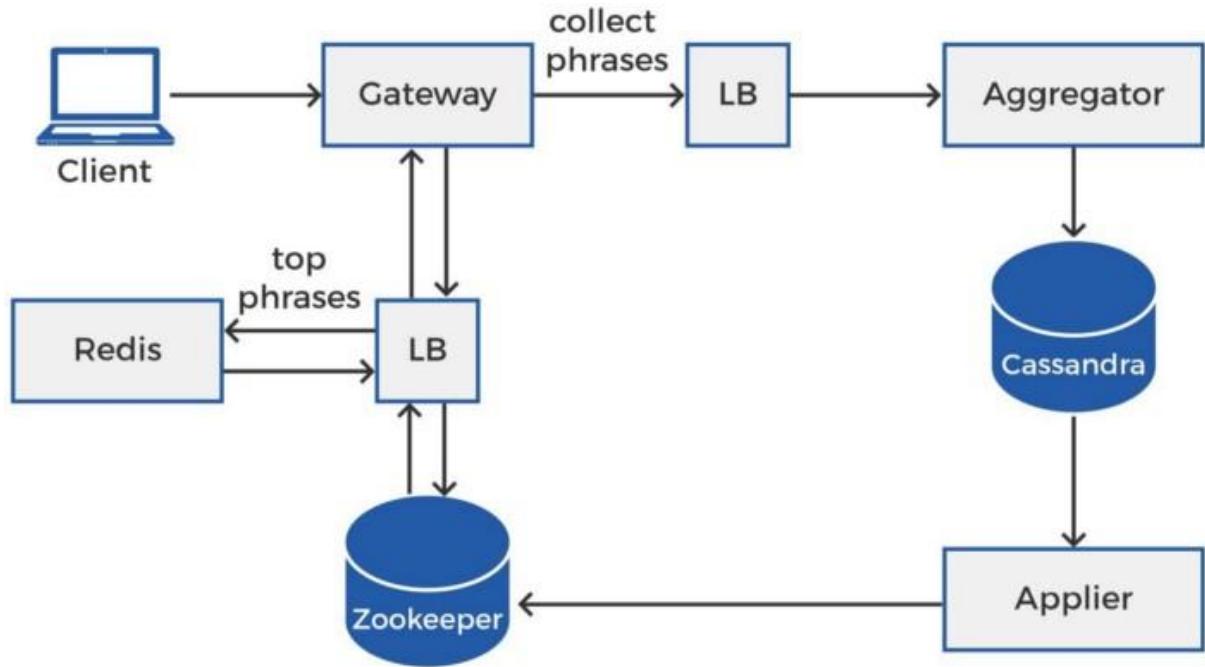
Two types of optimization are possible on this logged data:

1. We can discard the data older than a certain number of days if it is not relevant to our search autocomplete system. For example, the application might not need to incorporate search queries older than a month.
2. Our search autocomplete system might not have use for phrases with low search frequencies. For example if the aggregate daily or hourly frequency of a phrase is less than a certain number, let's say 100, we might want to lose that phrase from the hash table because we know that the phrase does not fulfill the criteria to show up in the autocomplete search results.

Once we have the data log in the form of a table for thousands or millions of phrases based on the day the queries were made and their frequency as shown in the diagram above, we can incorporate the entire data into our trie. The microservice called 'applier' incorporates the temporary data logs into the trie in the database. It can run every hour, half an hour, or at any predetermined interval.

The applier will fetch the phrases from the hash table together with their frequency and build the trie internally based on some mathematical formula. Once the new trie is ready, it is dumped in place of the old trie in Zookeeper. Once the new trie is available at the Zookeeper, any new request made by a user will receive a response of top phrases based on the newly created trie.

Keeping in view the above technique, the final system design diagram will look like the one in the diagram below:



6. Designing Dropbox

Design a file or image hosting service that allows users to upload, store, share, delete and download files or images on their servers and provides synchronization across various devices.

Things to discuss and analyze:

- Approach to upload/view/search/share/download files or photos from any device.
- Service should support automatic synchronization between devices, i.e., after updating a file on one device, it should get synchronized on all devices.
- ACID (Atomicity, Consistency, Isolation, and Durability) property should be present in the system.
- Approach to track permission for file sharing.
- Allowing multiple users to edit the same document.
- The system should support storing large files up to a GB.

Solution 1:

Don't jump into the technical details immediately when you are asked this question in your interviews. Do not run in one direction, it will just create confusion between you and the interviewer. Most of the candidates make mistakes here and immediately they start listing out some bunch of tools, frameworks, or databases. Remember that your interviewer wants high-level ideas about how you will solve the problem. It doesn't matter what tools you will use, but how you define the problem, how you design the solution, and how you analyze the issue step by step. In this blog, instead of talking a lot about the web kind of services, we will assume there is a sync client installed in your computer or system and that client is always

looking into particular sync folders in which it is always monitoring the changes to the files and uploads it. Also, we won't be talking about how we build cloud storage. We will use some cloud storage services like Amazon S3 or any other storage services that keep the file in the cloud.

1. Discuss The Core Features

Before you jump into the solution always clarify all the assumptions you're making at the beginning of the interview. Ask questions to identify the scope of the system. This will clear the initial doubt and you will get to know what is specific detail the interviewer wants to consider in this service. So start with the core features of dropbox to discuss with the interviewers. If the interviewers want to add some more features (such as API integration) they will let you know.

- Users should be able to upload/download, update and delete the files
- File versioning (History of updates)
- File and folder sync

2. Traffic

- 12+ million unique users
- 100 million requests per day with lots of reads and write.

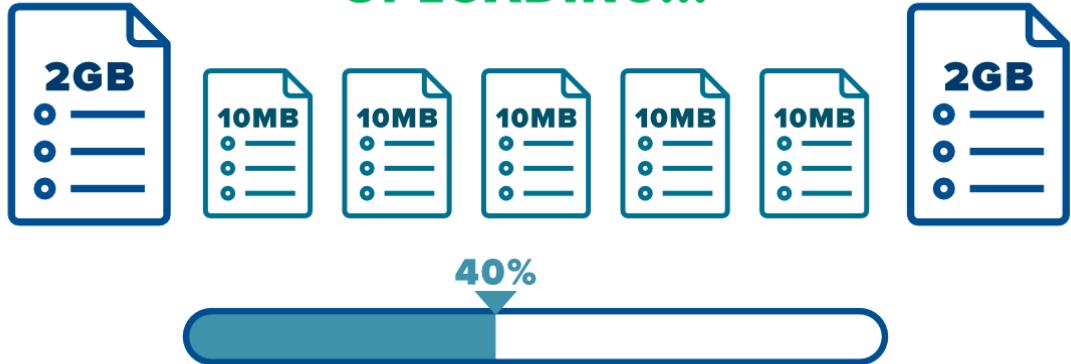
3. Discuss The Problem Statement

A lot of people assume designing a dropbox is that all they just need to do is to use some cloud services, upload the file, and download the file whenever they want but that's not how it works. The core problem is "*Where and how to save the files?*". Suppose you want to share a file that can be of any size (small or big) and you upload it into the cloud. Everything is fine till here but later if you have to make an update in your file then it's not a good idea to edit the file and upload the whole file again and again into the cloud. The reason is...

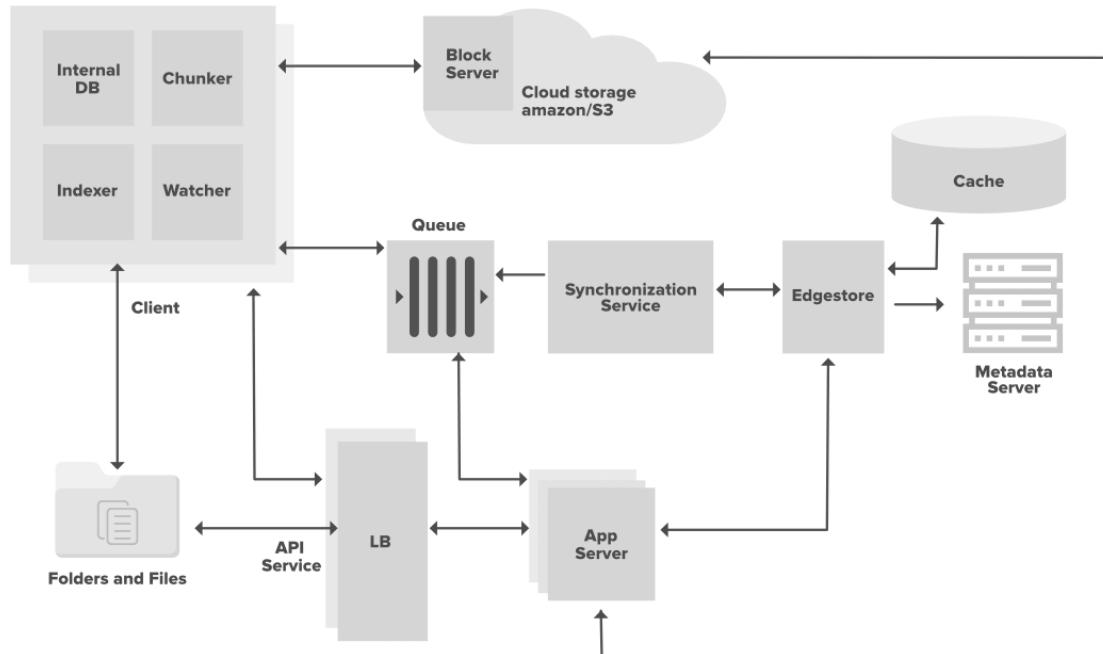
- **More bandwidth and cloud space utilization:** To provide a history of the files you need to keep multiple versions of the files. This requires more bandwidth and more space in the cloud. Even for the small changes in your file, you will have to back up and transfer the whole file into the cloud again and again which is not a good idea.
- **Latency or Concurrency Utilization:** You can't do time optimization as well. It will consume more time to upload a single file as a whole even if you make small changes in your file. It's also not possible to make use of concurrency to upload/download the files using multi threads or multi processes.

Discuss The Solution (High-Level Solution)

UPLOADING...



We can break the files into multiple chunks to overcome the problem we discussed above. There is no need to upload/download the whole single file after making any changes in the file. You just need to save the chunk which is updated (this will take less memory and time). It will be easier to keep the different versions of the files in various chunks. We have considered one file which is divided into various chunks. If there are multiple files then we need to know which chunks belong to which file. To keep this information we will create one more file named a **metadata file**. This file contains the indexes of the chunks (chunk names and order information). You need to mention the hash of the chunks (or some reference) in this metadata file and you need to sync this file into the cloud. We can download the metadata file from the cloud whenever we want and we can recreate the file using various chunks. Now let's talk about the various components for the complete system design solution of the dropbox service.



Let's assume we have a client installed on our computer (an app installed on your computer) and this client has 4 basic components. These basic components are Watcher, Chunker,

Indexer, and Internal DB. We have considered only one client but there can be multiple clients belonging to the same user with the same basic components.

- The client is responsible for uploading/downloading the files, identifying the file changes in the sync folder, and handling conflicts due to offline or concurrent updates.
- The client is actively monitoring the folders for all the updates or changes happening in the files.
- To handle file metadata updates (e.g. file name, size, modification date, etc.) this client interacts with the Messaging services and Synchronization Service.
- It also interacts with remote cloud storage (Amazon S3 or any other cloud services) to store the actual files and to provide folder synchronization.

Discuss The Client Components

- **Watcher** is responsible for monitoring the sync folder for all the activities performed by the user such as creating, updating, or deleting files/folders. It gives notification to the indexer and chunker if any action is performed in the files or folders.
- **Chunker** breaks the files into multiple small pieces called chunks and uploads them to the cloud storage with a unique id or hash of these chunks. To recreate the files these chunks can be joined together. For any changes in the files, the chunking algorithm detects the specific chunk which is modified and only saves that specific part/chunk to the cloud storage. It reduces the bandwidth usage, synchronization time, and storage space in the cloud.
- **Indexer** is responsible for updating the internal database when it receives the notification from the watcher (for any action performed in folders/files). It receives the URL of the chunks from the chunker along with the hash and updates the file with modified chunks. Indexer communicates with the Synchronization Service using the Message Queuing Service, once the chunks are successfully submitted to the cloud Storage.
- **Internal databases** stores all the files and chunks of information, their versions, and their location in the file system.

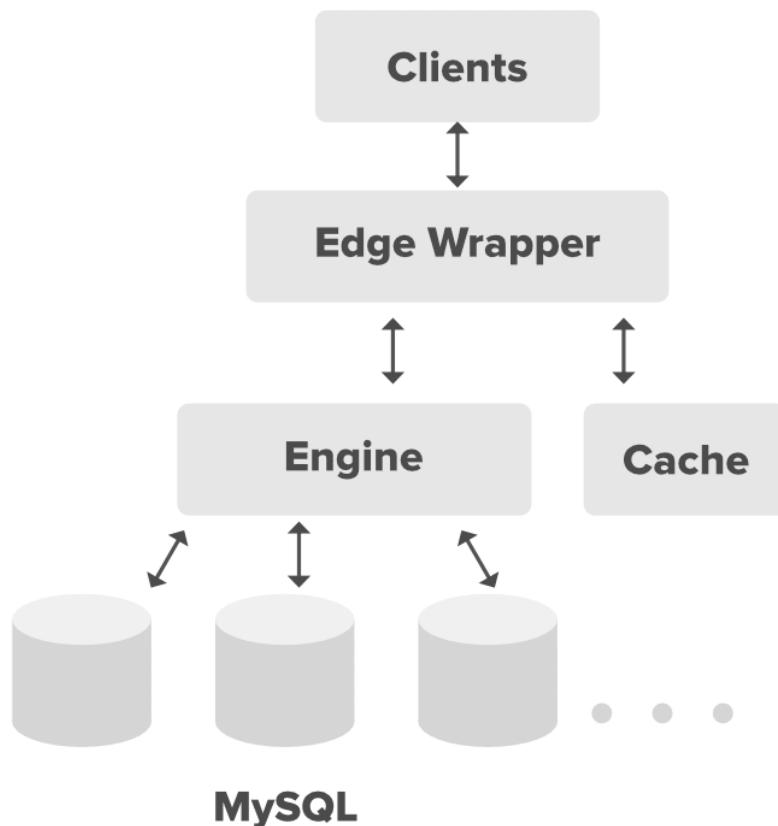
Discuss The Other Components

1. Metadata Database

The metadata database maintains the indexes of the various chunks. The information contains files/chunks names, and their different versions along with the information of users and workspace. You can use RDBMS or NoSQL but make sure that you meet the data consistency property because multiple clients will be working on the same file. With **RDBMS** there is **no problem with the consistency** but with **NoSQL**, you will get **eventual consistency**. If you decide to use NoSQL then you need to do different configurations for different databases (For example, the Cassandra replication factor gives the consistency level). **Relational databases** are **difficult to scale** so if you're using the MySQL database then you need to use a **database sharding technique** (or master-slave technique) to scale the application. In database sharding, you **need to add multiple MySQL databases** but it will be difficult to manage these databases for any update or for any new information that will be added to the databases. To overcome this problem we need to build an edge wrapper

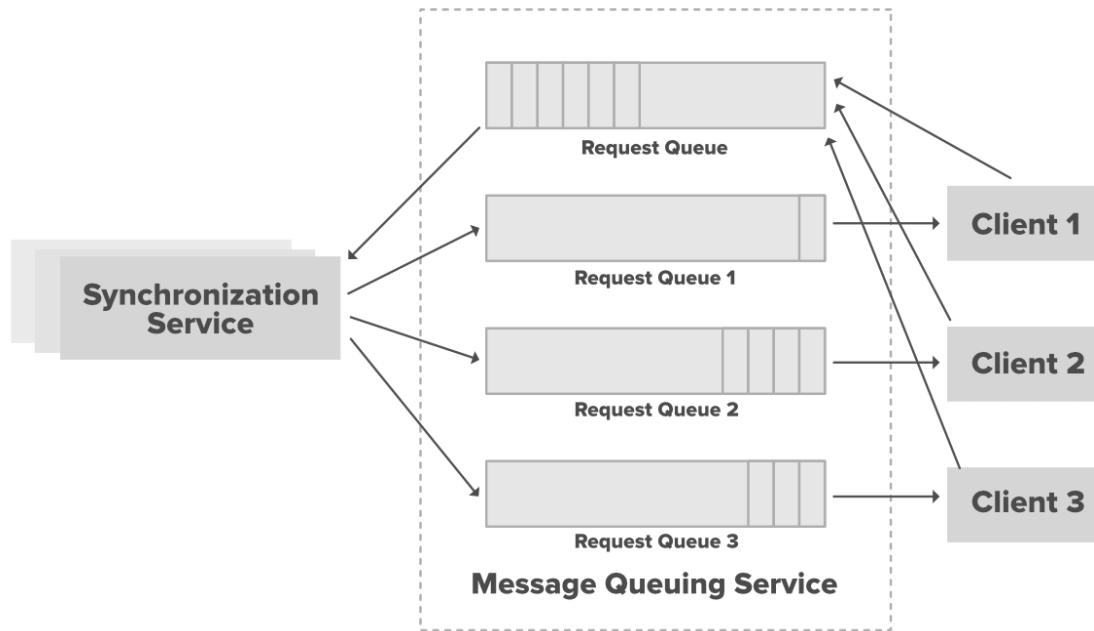
around the sharded databases. This edge wrapper provides the ORM and the client can easily use this edge wrapper's ORM to interact with the database (instead of interacting with the databases directly).

Metadata



2. Message Queuing Service

The messaging service queue will be responsible for the asynchronous communication between the clients and the synchronization service.



Below are the main requirements of the Message Queuing Service.

- Ability to handle lots of reading and writing requests.
- Store lots of messages in a highly available and reliable queue.
- High performance and high scalability.
- Provides load balancing and elasticity for multiple instances of the Synchronization Service.

There will be two types of messaging queues in the service.

- **Request Queue:** This will be a global request queue shared among all the clients. Whenever a client receives any update or changes in the files/folder it sends the request through the request queue. This request is received by the synchronization service to update the metadata database.
- **Response Queue:** There will be an individual response queue corresponding to the individual clients. The synchronization service broadcast the update through this response queue and this response queue will deliver the updated messages to each client and then these clients will update their respective files accordingly. The message will never be lost even if the client will be disconnected from the internet (the benefit of using the messaging queue service). We are creating n number of response queues for n number of clients because the message will be deleted from the queue once it will be received by the client and we need to share the updated message with the various subscribed clients.

3. Synchronization Service

The client communicates with the synchronization services either to receive the latest update from the cloud storage or to send the latest request/updates to the Cloud Storage. The synchronization service receives the request from the request queue of the messaging

services and updates the metadata database with the latest changes. Also, the synchronization service broadcast the latest update to the other clients (if there are multiple clients) through the response queue so that the other client's indexer can fetch back the chunks from the cloud storage and recreate the files with the latest update. It also updates the local database with the information stored in the Metadata Database. If a client is not connected to the internet or offline for some time, it polls the system for new updates as soon as it goes online.

4. Cloud Storage

You can use any cloud storage service like **Amazon S3** to store the chunks of the files uploaded by the user. The client communicates with the cloud storage for any action performed in the files/folders using the API provided by the cloud provider.

Solution 2:

Google Drive is a file hosting system powered by Google. It offers cloud file storage and synchronization service, allowing users to store their data on remote servers. Besides storing the file on these servers, Google Drive will also synchronize their files across multiple devices that they use and share it with other users as requested. Dropbox, OneDrive and Google Photos are similar applications that handle file storage and sharing for massive amounts of files for millions of users.

Based on the design of Google Drive, let's create a basic file storage and sharing service that can scale to millions of users and handle petabytes of data.

Requirements Of The System

It's important to clarify the requirements of the design. The actual design of such applications can cover several features and involve complexities that are beyond the scope of a system design interview. Narrow down the requirements to a few core components before building the system.

Functional Requirements

1. Users can upload and download files from any device that they are logged in.
2. Users can share files with other users.
3. The service should automatically synchronize files across all devices.

Non-Functional Requirements

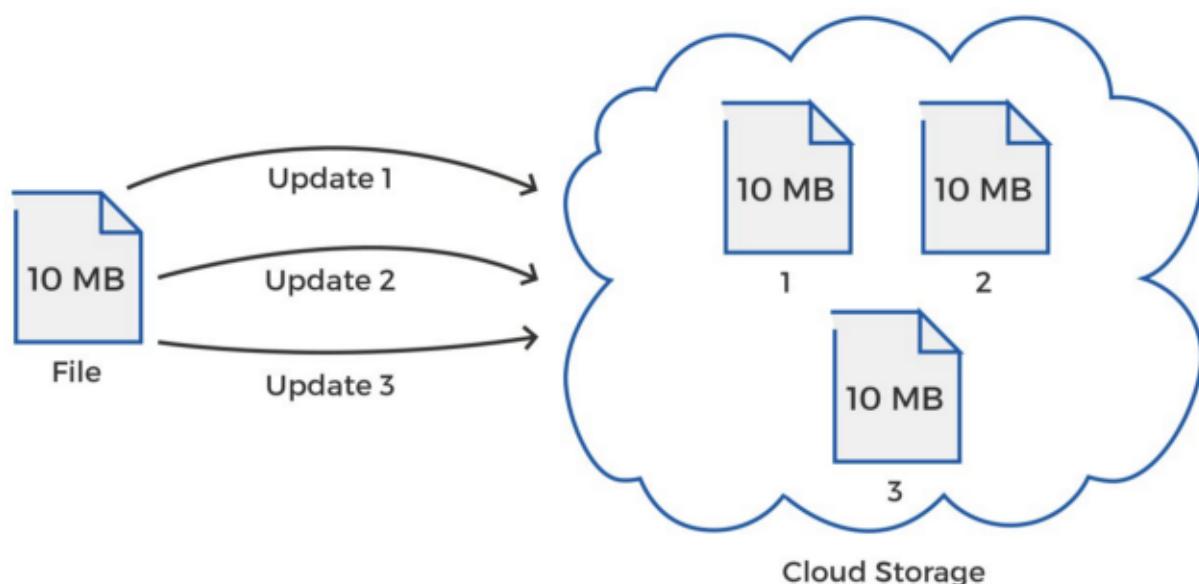
1. The system should support storage of large files of up to 1GB each.
2. The system should be able to scale to an enormous number of users (Google Drive has over 1 billion active users as of July 2018).
3. The system should be able to handle a high number of reads and writes (around 100 million requests per day). Read to write ratio is comparable in this case.
4. Minimum possible network bandwidth should be utilized for file synchronization.
5. There should be minimum latency in file transfer.

Uploading Files In Chunks

The last two non-functional requirements, including minimum bandwidth and minimum latency are both very important and is exactly why Google Drive and similar services choose to upload files in chunks rather than uploading a single large file.

Why Is Uploading A Complete File Not Practical?

If Google Drive were to upload a large file of, say, 10 MB, to the cloud storage, its upload as a single file would involve high latencies and bandwidth utilization. In case the upload fails, the entire file will need to be uploaded fresh, using further time, bandwidth and money. Also, if you were to update the file, the service will upload and store the entire 10 MB file again. With this approach, each update involves uploading and storing a fresh 10 MB file on the cloud storage, as shown in the diagram below.



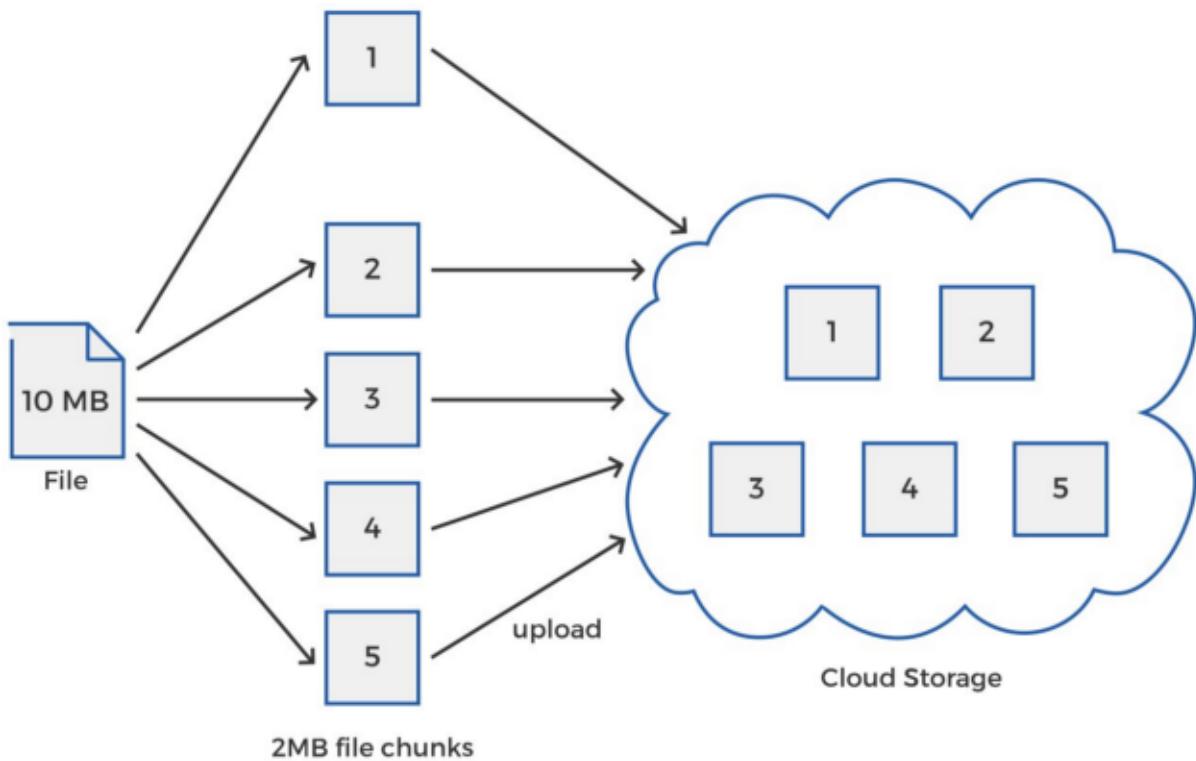
There are two drawbacks to this technique:

1. Each upload utilizes a 10MB bandwidth on the network.
2. You will use up 10 MB space on the cloud each time the file is updated.

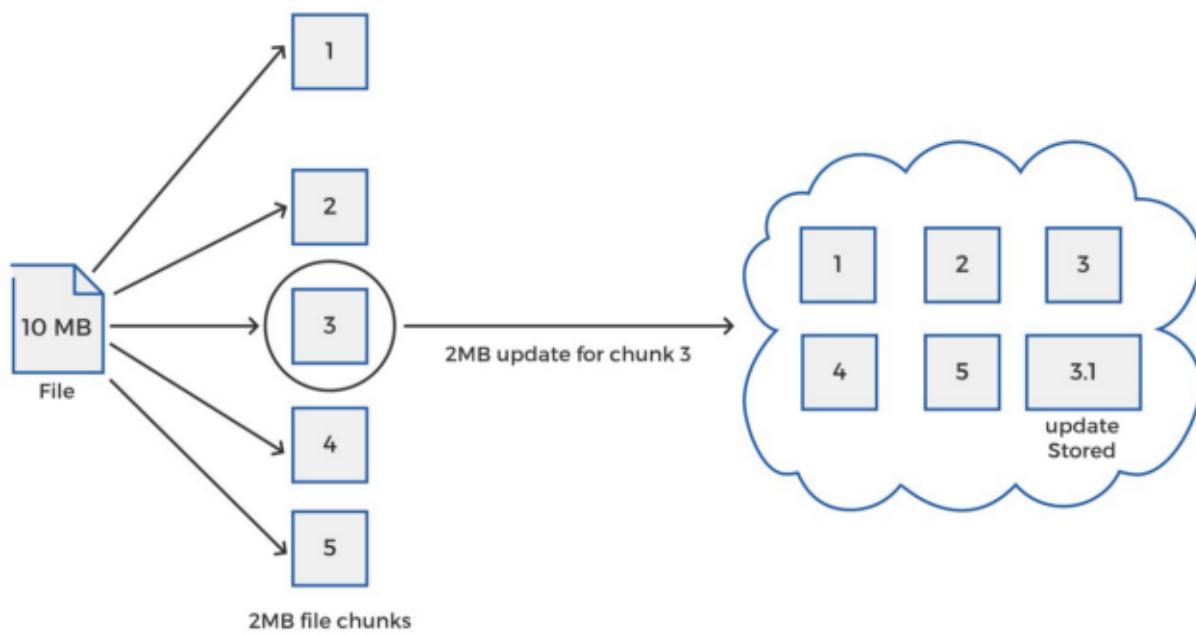
Now, we may overwrite the original file on the cloud storage and save all this space utilization. However, a file storage service typically keeps track of the update history, so we need a model that can store all the modifications while using minimum space on the cloud.

How Does Chunking Save Bandwidth And Space?

To improve efficiency, Google Drive came up with the solution to break files into chunks and handle these pieces independently instead of uploading complete files. Each file is divided into small pieces of equal, predetermined size, say 2 MB. Metadata database will hold information, including name and order, for each of these file chunks so the original file can be recreated using the chunks when downloading or syncing.



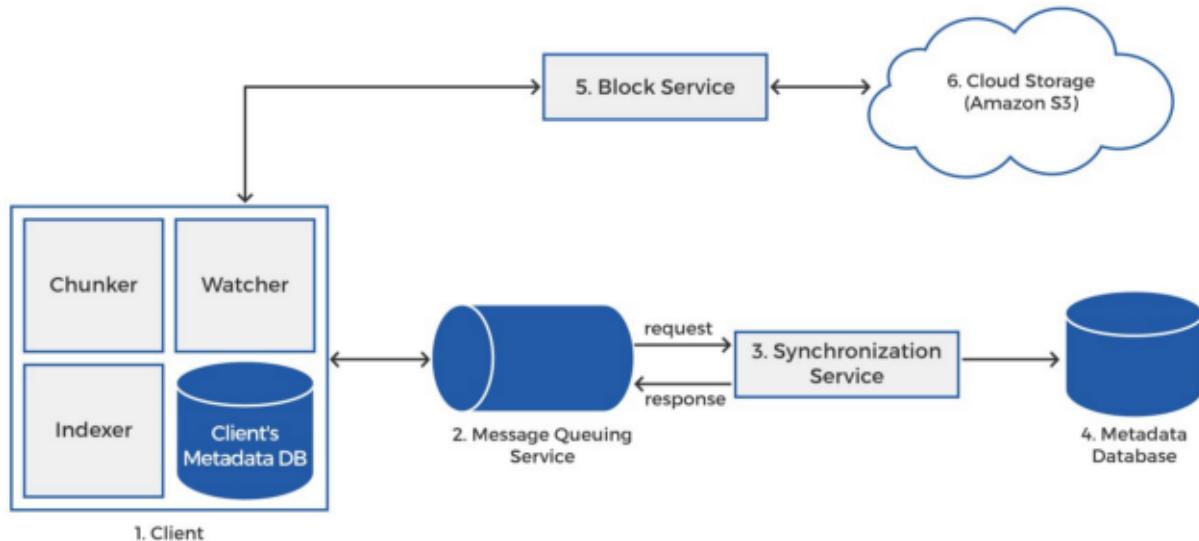
If the client were to upload a 10MB file on cloud storage, our file storage system will first break it into 5 parts, each of 2 MB and then upload it, as shown in the diagram above. If the client wishes to update any portion of the file, only the 2MB chunk that contains the updated portion will be uploaded again and not the entire file.



Consider the diagram above to see how an update happens with this improved approach. If you make a small update to a 10MB file, instead of transferring the entire 10 MB file again, you might only be uploading a single chunk (the modified chunk of the file), i.e., 2 MB, on the cloud storage. Through this approach, the system saves network bandwidth, uploading time and cloud storage.

Google Drive System Architecture

There are various components involved in the complete design of Google Drive system. Let's look at the high level system diagram and discuss the major components that make it work.



1. Client

The client is the Google Drive app installed on your device (computer or phone). The client actively monitors the workspace folders. These workspace folders, also called sync folders, are present on the client's machines. The client is also responsible for synchronizing these local folders with the cloud storage. The upload, download and any modification to the files on the cloud storage will occur from the client application on your desktop. The client will request the block server, or storage server, for accessing or uploading files on cloud storage.

There are multiple responsibilities that the client handles. To make it easier to understand and streamline performance, let's divide the desktop client into 4 subcomponents: Chunker, Watcher, Indexer and Client's Metadata DB.

1.1. Chunker

We have already covered the need for chunking files. The chunker, responsible for splitting the file into smaller pieces, is present within the client. The chunker is also responsible for detecting the chunks that are updated by the client so that only these specific chunks are uploaded to the cloud storage, thus saving bandwidth. Chunker will also reassemble the file by putting together the chunks in their correct order.

1.2. Watcher

The watcher will watch the workspace folders for any changes, and will relay changes made to the files by the client (such as creation, modification or deletion) to the indexer. In case multiple clients can make changes to the workspace folder, the watcher also watches for any changes made to the files by other clients that it receives through the synchronization service.

1.3. Indexer

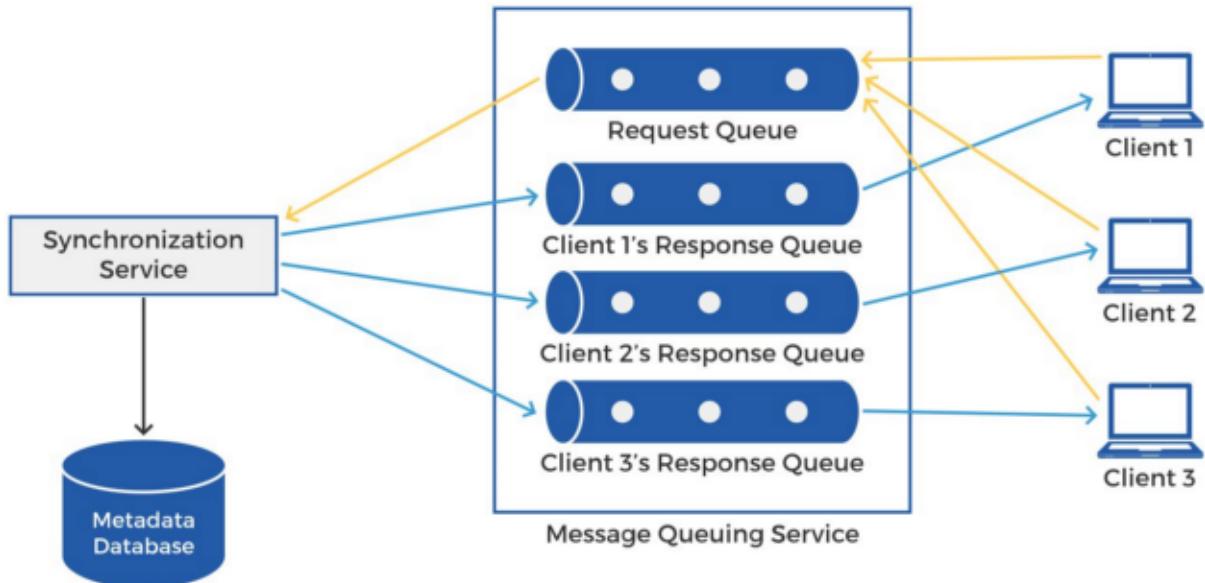
The indexer receives changes made to the workspace folder via the watcher and updates the client's local metadata database with the information of modified chunks for different files. After the modified chunks are uploaded to the cloud storage, the indexer will send messages to the synchronization service via a message queuing service to update the metadata database and notify all the clients of the recent updates.

1.4. Client's Metadata DB

Other than the main remote metadata database, each client maintains its local database. This database carries all the information about the files, chunks, and their versions for the workspace folders for that client. Client's Metadata Database allows the clients to make offline changes to files which can be updated in the remote metadata database once the client is online.

2. Message Queuing Service

We use a scalable message queuing service for asynchronous communication between the indexer (which is a part of the client) and the synchronization service. Since all the file updates are pushed via the message queuing service, it should be able to handle a large number of read and write requests. Other than high scalability, the message queuing service should allow reliable storage for a large number of concurrent messages in a highly available queue.



Our message queuing service will handle two types of queues:

2.1. Request Queue

The message queuing service maintains a single global request queue that receives requests from all the clients. When the indexer inside the client receives any updates to the files or folders, the indexer pushes the request through the request queue. The request queue will relay the request to the synchronization service to update the metadata database.

2.2. Response Queue

Unlike request queue, each client will have its individual response queue maintained by the message queuing service. The updates received by the synchronization service are broadcasted over the response queues of the subscribed clients. The subscribed clients will receive messages from the response queue to update their workspace folders accordingly. The reason for maintaining individual response queues for clients is that messages are deleted from the queue once they are delivered. We need to have a separate response queues for each subscribed client to ensure that every update is sent to all the clients before being removed from the queue.

3. Synchronization Service

The client communicates with the synchronization service over the message queue for two purposes:

1. To send the latest update made in the local workspace to the cloud storage. The synchronization service receives the client's request from the request queue over the message queuing service and updates the remote metadata database with the latest update sent by the client.
2. To receive the latest updates that occur at the cloud storage. It synchronizes the client's metadata database present with each of the subscribed clients with the updates made at the remote metadata database. These updates are broadcast to the subscribed clients by the synchronization service over the response queues at the message queuing service. The indexer picks up the latest chunks from the cloud storage and syncs its local files with these updates. If a client is offline for an interval, it will poll Google Drive for the latest updates when it goes back online.

In short, the synchronization service manages metadata information and synchronizes users' files.

4. Metadata Storage

For file storage, we can use a cloud storage service, such as Amazon S3. The cloud storage service will simply store the files for us. From this cloud storage, our system can download or upload files as needed. In simple words, we are designing a service that uploads and downloads files from the cloud and can share it with other users. The design of the cloud storage is beyond our scope.

Although our Google Drive -like service does not cater file storage, it will need to keep track of all the file chunks (including which file they belong to and their order), file versions, users and workspaces. All this metadata information can be stored in a metadata database. You can use a relational database such as MySQL, or a NoSQL database like MongoDB to store metadata information.

Since multiple clients can make changes to the file at the same time, we need to ensure data consistency. MySQL might be a better choice in this case since it already supports ACID properties. If you choose to go with NoSQL, you will need to incorporate ACID properties in the design for file synchronization.

5. Block Service

Block service is the microservice responsible for communicating with the cloud storage for uploading and downloading files. The client will request the block service whenever it needs to upload or download a file chunk from the cloud storage.

In the case where multiple clients can edit a file, as soon as the indexer receives a broadcast message from the messaging queue that an update has been made, the client will request the block service to fetch the updated file chunk from the cloud storage.

6. Cloud Storage

We can use Amazon S3 as our cloud storage service. All the file chunks will be uploaded and stored at the Amazon S3 data centers. The clients communicate directly with the block service to send or receive file chunks from the cloud storage.

Solution 3:

Requirements

Let's look in to some of the functional and non-functional requirements before we start to design the system.

Functional Requirements

1. Users should be able to sign up using their email address and subscribe to a plan. If they don't subscribe, they will get 1 GB of free storage.
2. Users should be able to upload and download their files from any device.
3. Users should be able to share files and folders with other users.
4. Users should be able to upload files up to 1 GB.
5. System should support automatic synchronization across the devices.
6. System should support offline editing. Users should be able to add/delete/modify files offline and once they come online, changes should be synchronized to remote server and other online devices.

Non-functional requirements

1. System should be highly reliable. Any file uploaded should not be lost.
2. System should be highly available.

Out of scope

1. Real-time collaboration on a file.
2. Versioning of the file.

Capacity Estimation

Let's do some back-of-the-envelope calculations to estimate the bandwidth and storage required.

Assumptions

1. Total number of users = 500 million
2. Total number of daily active users = 100 million
3. Average number of files stored by each user = 200
4. Average size of each file = 100 KB
5. Total number of active connections per minute = 1 million

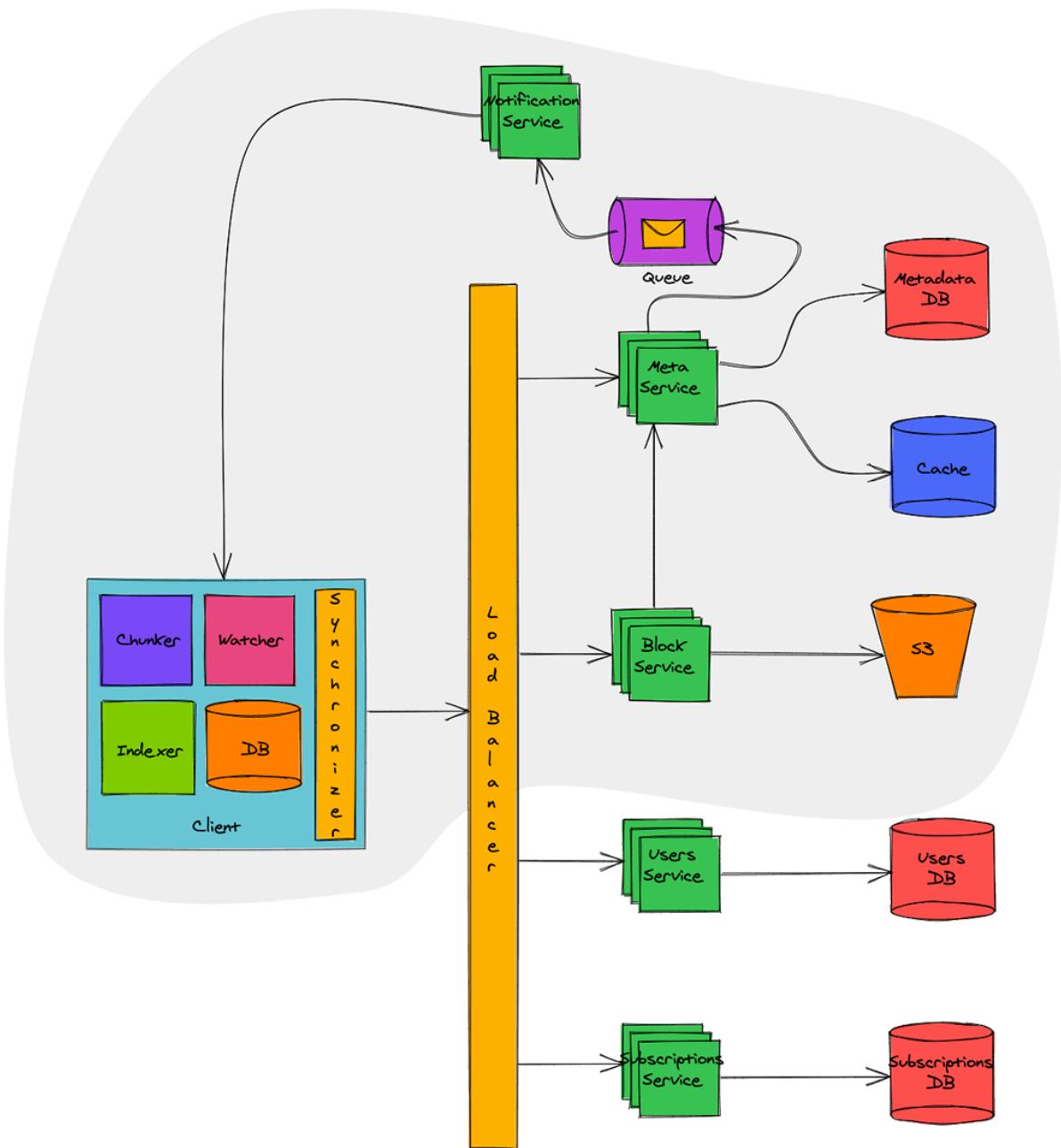
Storage Estimations

Total number of files = 500 million * 200 = 100 billion

Total storage required = 100 billion * 100 KB = 10 PB

Detailed Component Design

The system needs to deal with huge volume of read and write data and their ratio will remain almost same. Hence while designing the system, we should focus on optimizing the data exchange between client and server.



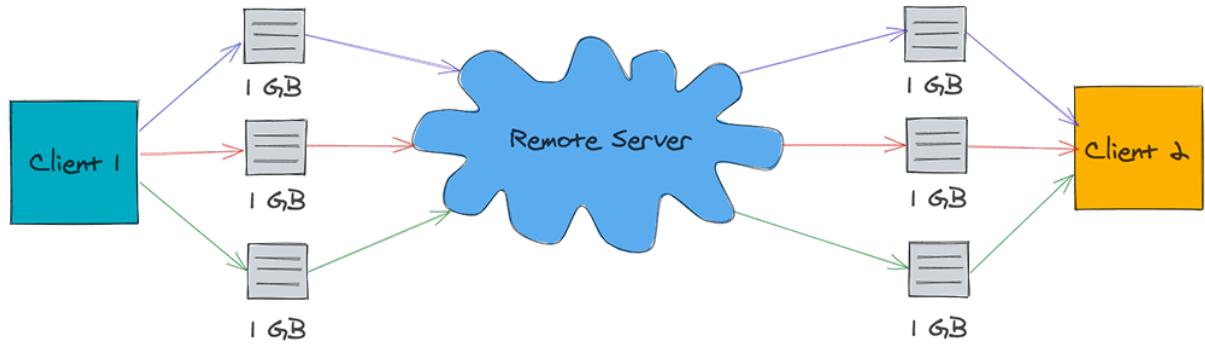
Our focus is on building the components which are in grey area in diagram above. Other components, which are outside, like [Users Service](#) and [Subscriptions Service](#) have been discussed in detail previously. Hence let's look in to remaining components in detail.

Client

Client here means desktop or mobile application which keeps a watch on user's workspace and synchronizes the files with remote server. Below are some main responsibilities of client:

- Watch workspace for changes
- Upload or download changes to file from remote server
- Handle the conflicts due to offline or concurrent updates
- Update file metadata on remote server if they change

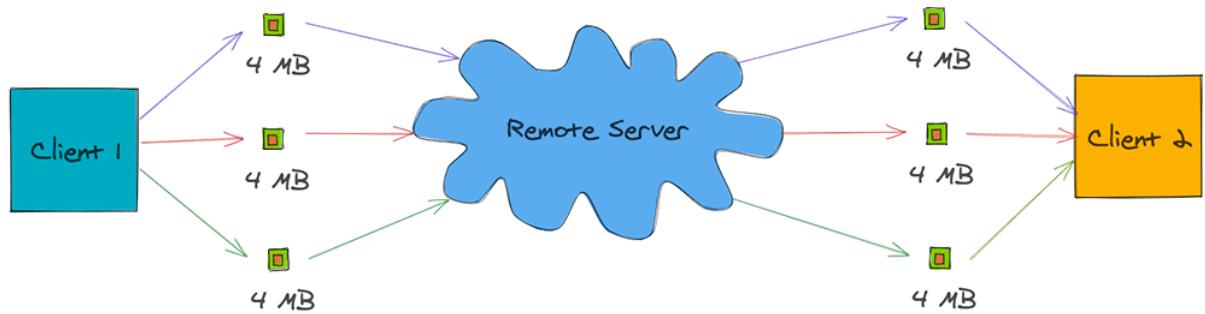
Let's naively assume that we build a client which synchronizes the file on each modification to remote server.



As shown in image above, let's assume that there is a file of 1 GB and three successive small changes were made to this file. Due to this, the file is sent three time to remote server and same is downloaded three times on another client. In this whole process, 3 GB of bandwidth was used for upload and 3 GB of bandwidth was used for download. Further, the download bandwidth increases in proportion of clients watching the file.

Did you observe that a total of 6 GB of bandwidth was used for just a small change? Also if there is a loss in connectivity, client has to upload/download entire file again. This is a huge waste of bandwidth and hence let's try to optimize it.

Now let's assume we build a client which breaks the file in to smaller chunks of say 4 MB and uploads them to remote server as shown below:



If there is any change in file, client determines which chunk has changed and just uploads that chunk to remote server. Similarly on other side, other client gets notified about the chunk that has changed and downloads just that chunk. This way just 24 MB of bandwidth was used, in contrast to 6 GB earlier, for synchronizing three small changes to the file.

Keeping this in mind, let's look in to the different components of this optimized client:

Client Metadata Database: Client Metadata Database stores the information about different files in workspace, file chunks, chunk version and file location in the file system. This can be implemented using a lightweight database like [SQLite](#).

Chunker: Chunker splits the big files in to chunks of 4 MB. This also reconstructs the original file from chunks.

Watcher: Watcher monitors for file changes in workspace like update, create, delete of files and folders. Watcher notifies Indexer about the changes.

Indexer: Indexer listens for the events from watcher and updates the Client Metadata Database with information about the chunks of modified file. It also notifies Synchronizer after committing changes to Client Metadata Database.

Synchronizer: Synchronizer listens for events from Indexer and communicates with Meta Service and Block service for updating meta data and modified chunk of file on remote server respectively. It also listens for changes broadcasted by Notification Service and downloads the modified chunks from remote server.

Meta Service

Meta Service is responsible for synchronizing the file metadata from client to server. It's also responsible to figure out the change set for different clients and broadcast it to them using Notification Service.

When a client comes online, it pings Meta Service for an update. Meta Service determines the change set for that client by querying the Metadata DB and returns the change set.

If a client updates a file, Meta Service again determines the change set for other clients watching that file and broadcasts the change set via Notification Service.

Meta Service is backed by Metadata DB. This database contains the metadata of file like name, type (file or folder), sharing permissions, chunks information etc. This database should have strong [ACID](#) (atomicity, consistency, isolation, durability) properties. Hence a relational database, like MySQL or PostgreSQL, would be a good choice.

Since querying the database for every synchronization request is a costly operation, a in-memory cache is put in front of Metadata DB. Frequently queries data is cached in this cache thereby eliminating the need of database query. This cache can be implemented using Redis or Memcached and [write-around cache](#) strategy can be applied for optimal performance.

Dropbox uses a clever algorithm for efficiently synchronizing files across multiple clients. You can read more details about the same [here](#).

Block Service

Block Service interacts with block storage for uploading and downloading of files. Clients connect with Block Service to upload or download file chunks.

When a client finishes downloading file, Block Service notifies Meta Service to update the metadata. When a client uploads a file, Block Service on finishing the upload to block storage, notifies the Meta Service to update the metadata corresponding to this client and broadcast messages for other clients.

Block Storage can be implemented using a distributed file system like [Glusterfs](#) or Amazon S3. Distributed file system provides high reliability and durability making sure the files uploaded are never lost.

When Dropbox started, they used S3 as block storage. However as they grew, they developed an in-house multi-exabyte storage system known as [Magic Pocket](#). In magic Pocket, files are split up into blocks, replicated for durability, and distributed across data centers in multiple geographic regions.

Notification Service

Notification Service broadcasts the file changes to connected clients making sure any change to file is reflected all watching clients instantly.

Notification Service can be implemented using HTTP Long Polling, Websockets or Server Sent Events. Websockets establish a persistent duplex connection between client and server. It is not a good choice in this scenario as there is no need of two way communication. We only need to broadcast the message from service to client and hence it's an overkill.

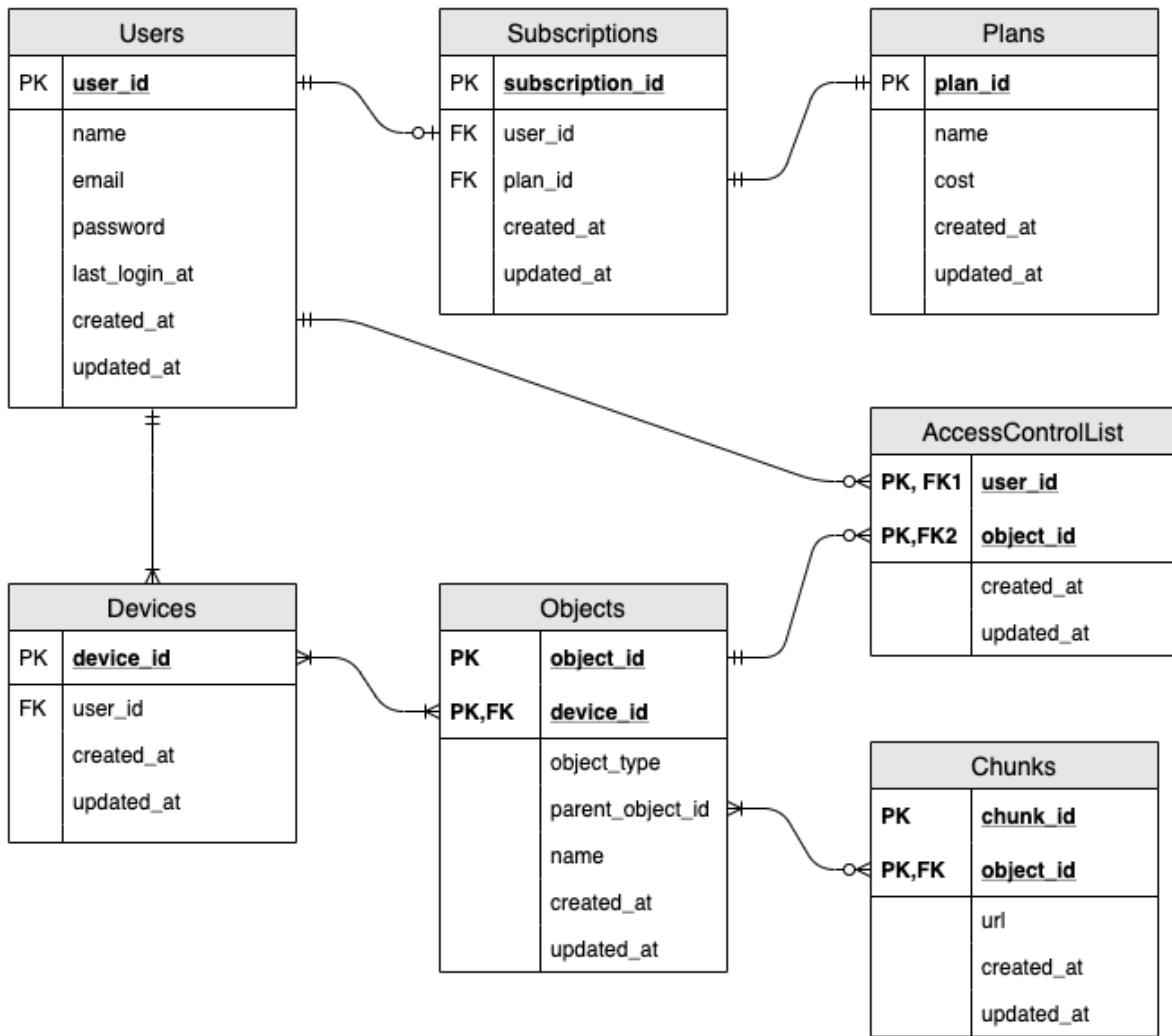
HTTP Long Polling is a better choice as server keeps the connection hanging till a data is available for client. Once data is available, server sends the data closing the connection. Once the connection is closed, client has to again establish a new connection. Generally, for each long poll request, there is a timeout associated with it and client must establish a new connection post timeout.

In Server Sent Events, client establishes a long term persistent connection with server. This connection is used to send events from server to client. There is no timeout and connection remains alive till the client remains on network. This fits our use case perfectly and would be a good choice for designing Notification Service. Though the Server Sent Events are not supported in all browsers, it's not a concern for us as we have our custom built desktop and mobile clients where we can utilize it.

Notification Service before sending the data to clients, reads the message from a message queue. This queue can be implemented using RabbitMQ, Apache ActiveMQ or Kafka. Message queue provides a asynchronous medium of communication between Meta Service and Notification Service and thus Meta Service need not to wait till notification is sent to clients. Notification service can keep on consuming the messages at its own pace without affecting the performance Meta Service. This decoupling also allows us to scale both services independently.

Database Schema

The database schema containing most important tables is illustrated below:



Let's quickly sum up above database schema:

- A user may subscribe for paid service.
- Each subscription must have one plan.
- Each user must have at-least one device.
- Each device will have at-least one object (file or folder). Once user registers, we create a root folder for him/her making sure he/she has at-least one object.
- Each object may have chunks. Only files can have chunk, folders can't have chunks.
- Each object may be shared with one or multiple users. This mapping is maintained in **AccessControlList**.

APIs

The service will expose API for uploading file and downloading file. Other API like user sign-up, sign-in, sign-out, subscribing, unsubscribing etc. have already been discussed in [this article](#).

Download Chunk

This API would be used to download the chunk of a file.

Request:

```
GET /api/v1/chunks/:chunk_id  
X-API-Key: api_key  
Authorization: auth_token
```

Response:

```
200 OK  
Content-Disposition: attachment; filename=<chunk_id>  
Content-Length: 4096000
```

The response will contain Content-Disposition header as attachment which will instruct the client to download the chunk. Note that Content-Length is set as 4096000 as each chunk is of 4 MB.

Upload Chunk

This API would be used to upload the chunk of a file.

Request:

```
POST /api/v1/chunks/:chunk_id  
X-API-Key: api_key  
Authorization: auth_token  
Content-Type: application/octet-stream  
/path/to/chunk
```

Content-Type header is set as application/octet-stream to tell server that a binary date is being sent.

Response:

```
200 OK
```

On successful upload, the server will return HTTP response code 200. Below are some possible failure response codes:

```
401 Unauthorized  
400 Bad request  
500 Internal server error
```

Get Objects

This API would be used by clients to query Meta Service for new files/folders when they come online.

Request:

```
GET /api/v1/objects?local_object_id=<Max object_id present locally>&device_id=<Unique  
Device Id>  
X-API-Key: api_key  
Authorization: auth_token
```

Client will pass the maximum object id present locally and the unique device id.

Response:

200 OK

```
{  
  new_objects: [  
    {  
      object_id:  
      object_type:  
      name:  
      chunk_ids: [  
        chunk1,  
        chunk2,  
        chunk3  
      ]  
    }  
  ]  
}
```

Meta Service will check the database and return an array of objects containing name of object, object id, object type and an array of chunk_ids. Client calls the Download Chunk API with these chunk_ids to download the chunks and reconstruct the file.

Performance

Chunking of files and data deduplication while uploading files boosts the performance a lot. Let's look in to both of these in detail.

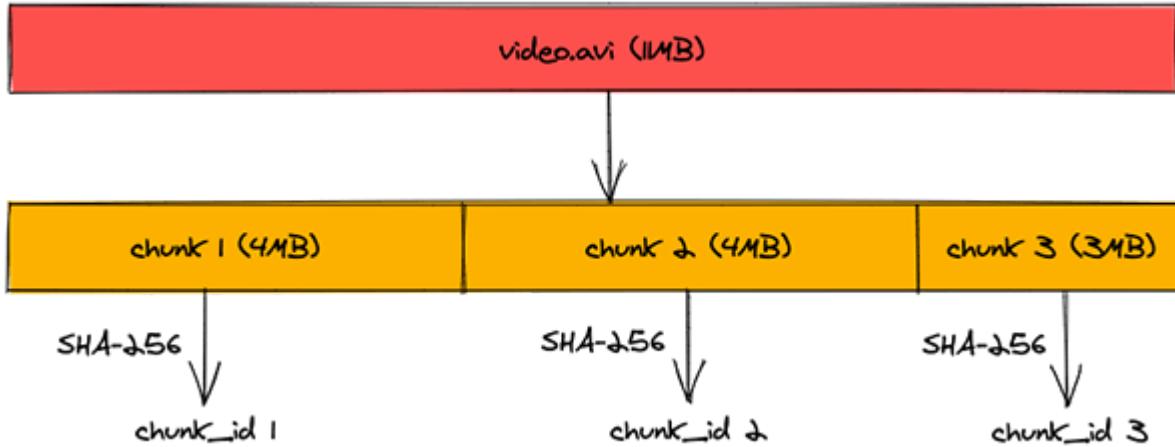
Chunking Files

Instead of uploading entire file in one go, we are chunking files in blocks of 4 MB and then uploading each chunk. This helps in improving performance in following ways:

- Multiple chunks can be uploaded in parallel thereby reducing the time for upload.
- If there is a small change in file, only the affected chunk is uploaded and downloaded by other clients, saving us bandwidth and time.
- If file upload is interrupted due to network connectivity loss, instead of uploading/downloading entire file again, we can just resume after the last chunk already uploaded/downloaded. This again saves our bandwidth and time.

Data Deduplication

The files are getting chunked in to blocks of 4 MB and each chunk is assigned a chunk_id which is SHA-256 hash of that chunk.



When a client tries to upload a chunk whose chunk_id is already present in Metadata DB, Meta Service just adds a row in Chunks table containing the new object_id and chunk_id. This eliminates the need to re-uploading the data thereby saving bandwidth and time. This also helps us in saving the space of block storage as there won't be multiple copies of one chunk in same data center.

Caching

We are using an in-memory caching to reduce the number of hits to Metadata DB. In-memory caches like Redis and Memcached cache the data from database in key-value pair.

From Meta Service servers, before hitting the Metadata DB, we are checking if data exists in cache or not. If it exists then we return the value from there itself bypassing a database trip. However if data is not present in cache, we hit the database, getting the data and populating the same in cache too. Hence subsequent requests won't hit the database and get the data from cache itself. This caching strategy is known as write-around caching.

Least Recently Used (LRU) eviction policy is used for caching data as it allows us to discard the keys which are least recently fetched.

Scalability

Our architecture is highly scalable owing to following facts:

Horizontal Scaling

We can add more servers behind the load balancer to increase the capacity of each service. This is known as Horizontal Scaling and each service can be independently scaled horizontally in our design.

Database Sharding

Metadata DB is sharded based on object_id. Our hash function will map each object_id to a random server where we can store the file/folder metadata. To query for a particular object_id, service can determine the database server using same hash function and query for data. This approach will distribute our database load to multiple servers making it scalable.

Cache Sharding

Similar to Metadata DB Sharding, we are distributing the cache to multiple servers. In-fact Redis has out of box support for [partitioning](#) the data across multiple Redis instances. Usage of [Consistent Hashing](#) for distributing data across instances ensures that load is equally distributed if one instance goes away.

Security

Our system is highly secure due to following:

HTTPS

The traffic between client and server is encrypted over HTTPS. This ensures that no one in the middle is able to see the data, especially the file contents.

Authentication

For each API request post log-in, we are doing authentication by checking the validity of auth_token in Authorization HTTP header. This makes sure that requests which originate from clients are legitimate.

Resiliency

Our system highly resilient owing to following:

Distributed Block Storage

Files are split up in to chunks and these chunks are replicated within data center for durability. Also these chunks are distributed across data centers in multiple geographic regions for redundancy. This makes sure that enough copies of chunks are available within data center if one machine goes down. Also if entire data center goes down, chunks can be served from a data center in other geographical location.

Queuing

We are using queuing in our system for sending out the notification to clients. Hence if any worker dies, message in a queue isn't acknowledged and other worker picks up the task again.

Load Balancing

Since we are putting multiple servers behind the load balancer, there is redundancy. Load Balancer is continuously doing health check on servers behind it. If any server dies, load balancer stops forwarding the traffic to it and removes it from cluster. This makes sure that requests don't fail due to a unresponsive server.

Geo-redundancy

We are deploying exact replica of our services in data-centers across multiple geographical locations. This ensures that if one data-center goes down due to some reason, the traffic could still be served from remaining data-centers.

7. Designing a Web Crawler

Design a Web Crawler scalable service that collects information (crawl) from the entire web and fetches hundreds of millions of web documents.

Things to discuss and analyze:

- Approach to find new web pages.
- Approach to prioritize web pages that change dynamically.
- Ensure that crawler is not unbounded on the same domain.

Solution 1:

1. What is a Web Crawler?

A web crawler is a software program which browses the World Wide Web in a methodical and automated manner. It collects documents by recursively fetching links from a set of starting pages. Many sites, particularly search engines, use web crawling as a means of providing up-to-date data. Search engines download all the pages to create an index on them to perform faster searches.

Some other uses of web crawlers are:

- To test web pages and links for valid syntax and structure.
- To monitor sites to see when their structure or contents change.
- To maintain mirror sites for popular Web sites.
- To search for copyright infringements.
- To build a special-purpose index, e.g., one that has some understanding of the content stored in multimedia files on the Web.

2. Requirements and Goals of the System

Let's assume we need to crawl all the web.

Scalability: Our service needs to be scalable such that it can crawl the entire Web and can be used to fetch hundreds of millions of Web documents.

Extensibility: Our service should be designed in a modular way with the expectation that new functionality will be added to it. There could be newer document types that need to be downloaded and processed in the future.

3. Some Design Considerations

Crawling the web is a complex task, and there are many ways to go about it. We should be asking a few questions before going any further:

Is it a crawler for HTML pages only? Or should we fetch and store other types of media, such as sound files, images, videos, etc.? This is important because the answer can change the design. If we are writing a general-purpose crawler to download different media types, we might want to break down the parsing module into different sets of modules: one for HTML, another for images, or another for videos, where each module extracts what is considered interesting for that media type.

Let's assume for now that our crawler is going to deal with HTML only, but it should be extensible and make it easy to add support for new media types.

What protocols are we looking at? HTTP? What about FTP links? What different protocols should our crawler handle? For the sake of the exercise, we will assume HTTP. Again, it shouldn't be hard to extend the design to use FTP and other protocols later.

What is the expected number of pages we will crawl? How big will the URL database become? Let's assume we need to crawl one billion websites. Since a website can contain many, many URLs, let's assume an upper bound of 15 billion different web pages that will be reached by our crawler.

What is 'RobotsExclusion' and how should we deal with it? Courteous Web crawlers implement the Robots Exclusion Protocol, which allows Webmasters to declare parts of their sites off limits to crawlers. The Robots Exclusion Protocol requires a Web crawler to fetch a special document called robot.txt which contains these declarations from a Web site before downloading any real content from it.

4. Capacity Estimation and Constraints

If we want to crawl 15 billion pages within four weeks, how many pages do we need to fetch per second?

$$15B / (4 \text{ weeks} * 7 \text{ days} * 86400 \text{ sec}) \approx 6200 \text{ pages/sec}$$

What about storage? Page sizes vary a lot, but, as mentioned above since, we will be dealing with HTML text only, let's assume an average page size of 100KB. With each page, if we are storing 500 bytes of metadata, total storage we would need:

$$15B * (100KB + 500) \approx 1.5 \text{ petabytes}$$

Assuming a 70% capacity model (we don't want to go above 70% of the total capacity of our storage system), total storage we will need:

$$1.5 \text{ petabytes} / 0.7 \approx 2.14 \text{ petabytes}$$

5. High Level design

The basic algorithm executed by any Web crawler is to take a list of seed URLs as its input and repeatedly execute the following steps.

1. Pick a URL from the unvisited URL list.
2. Determine the IP Address of its host-name.
3. Establish a connection to the host to download the corresponding document.
4. Parse the document contents to look for new URLs.
5. Add the new URLs to the list of unvisited URLs.
6. Process the downloaded document, e.g., store it or index its contents, etc.
7. Go back to step 1

How to crawl?

Breadth first or depth first? Breadth-first search (BFS) is usually used. However, Depth First Search (DFS) is also utilized in some situations, such as, if your crawler has already established a connection with the website, it might just DFS all the URLs within this website to save some handshaking overhead.

Path-ascending crawling: Path-ascending crawling can help discover a lot of isolated resources or resources for which no inbound link would have been found in regular crawling of a particular Web site. In this scheme, a crawler would ascend to every path in each URL that it intends to crawl. For example, when given a seed URL of <http://foo.com/a/b/page.html>, it will attempt to crawl /a/b/, /a/, and /.

Difficulties in implementing efficient web crawler

There are two important characteristics of the Web that makes Web crawling a very difficult task:

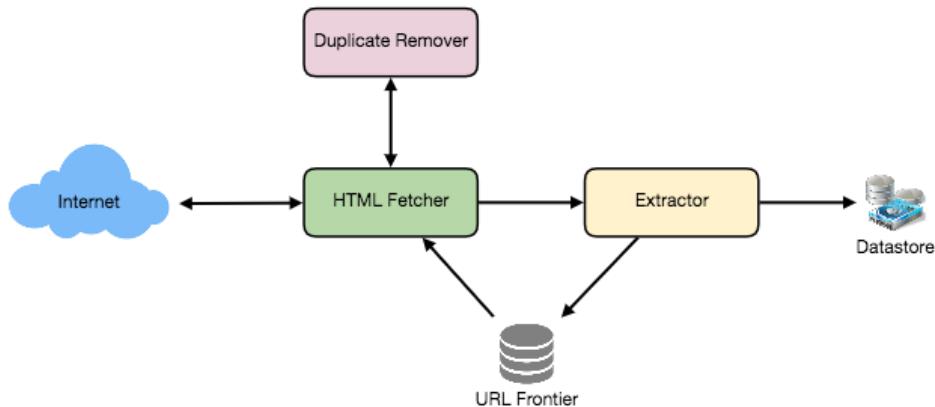
1. Large volume of Web pages: A large volume of web pages implies that web crawler can only download a fraction of the web pages at any time and hence it is critical that web crawler should be intelligent enough to prioritize download.

2. Rate of change on web pages. Another problem with today's dynamic world is that web pages on the internet change very frequently. As a result, by the time the crawler is downloading the last page from a site, the page may change, or a new page may be added to the site.

A bare minimum crawler needs at least these components:

1. URL frontier: To store the list of URLs to download and also prioritize which URLs should be crawled first.

2. **HTTP Fetcher:** To retrieve a web page from the server.
3. **Extractor:** To extract links from HTML documents.
4. **Duplicate Eliminator:** To make sure the same content is not extracted twice unintentionally.
5. **Datastore:** To store retrieved pages, URLs, and other metadata.



6. Detailed Component Design

Let's assume our crawler is running on one server and all the crawling is done by multiple working threads where each working thread performs all the steps needed to download and process a document in a loop.

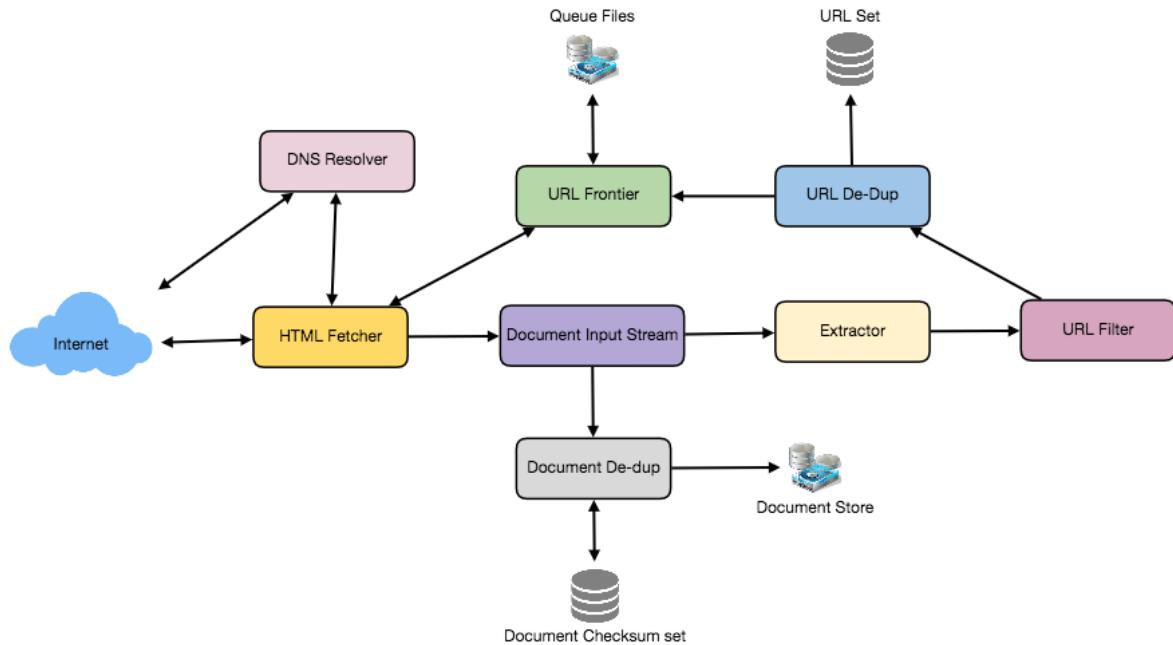
The first step of this loop is to remove an absolute URL from the shared URL frontier for downloading. An absolute URL begins with a scheme (e.g., "HTTP") which identifies the network protocol that should be used to download it. We can implement these protocols in a modular way for extensibility, so that later if our crawler needs to support more protocols, it can be easily done. Based on the URL's scheme, the worker calls the appropriate protocol module to download the document. After downloading, the document is placed into a Document Input Stream (DIS). Putting documents into DIS will enable other modules to re-read the document multiple times.

Once the document has been written to the DIS, the worker thread invokes the dedupe test to determine whether this document (associated with a different URL) has been seen before. If so, the document is not processed any further and the worker thread removes the next URL from the frontier.

Next, our crawler needs to process the downloaded document. Each document can have a different MIME type like HTML page, Image, Video, etc. We can implement these MIME schemes in a modular way, so that later if our crawler needs to support more types, we can easily implement them. Based on the downloaded document's MIME type, the worker invokes the process method of each processing module associated with that MIME type.

Furthermore, our HTML processing module will extract all links from the page. Each link is converted into an absolute URL and tested against a user-supplied URL filter to determine if

it should be downloaded. If the URL passes the filter, the worker performs the URL-seen test, which checks if the URL has been seen before, namely, if it is in the URL frontier or has already been downloaded. If the URL is new, it is added to the frontier.



Let's discuss these components one by one, and see how they can be distributed onto multiple machines:

1. The URL frontier: The URL frontier is the data structure that **contains all the URLs that remain to be downloaded**. We can crawl by performing a breadth-first traversal of the Web, starting from the pages in the seed set. Such traversals are easily implemented by using a FIFO queue.

Since we'll be having a huge list of URLs to crawl, we can **distribute our URL frontier into multiple servers**. Let's assume on each server we have multiple worker threads performing the crawling tasks. Let's also assume that our **hash function maps each URL to a server which will be responsible for crawling it**.

Following politeness requirements must be kept in mind while designing a distributed URL frontier:

1. Our crawler should not overload a server by downloading a lot of pages from it.
2. We should not have multiple machines connecting a web server.

To implement this politeness constraint our crawler can have a collection of distinct FIFO sub-queues on each server. Each worker thread will have its separate sub-queue, from which it removes URLs for crawling. When a new URL needs to be added, the FIFO sub-queue in which it is placed will be determined by the URL's canonical hostname. Our hash function can map each hostname to a thread number. Together, these two points imply

that, at most, one worker thread will download documents from a given Web server and also, by using FIFO queue, it'll not overload a Web server.

How big will our URL frontier be? The size would be in the hundreds of millions of URLs. Hence, we need to store our URLs on a disk. We can implement our queues in such a way that they have separate buffers for enqueueing and dequeuing. Enqueue buffer, once filled, will be dumped to the disk, whereas dequeue buffer will keep a cache of URLs that need to be visited; it can periodically read from disk to fill the buffer.

2. The fetcher module: The purpose of a fetcher module is to download the document corresponding to a given URL using the appropriate network protocol like HTTP. As discussed above, webmasters create robot.txt to make certain parts of their websites off limits for the crawler. To avoid downloading this file on every request, our crawler's HTTP protocol module can maintain a fixed-sized cache mapping host-names to their robot's exclusion rules.

3. Document input stream: Our crawler's design enables the same document to be processed by multiple processing modules. To avoid downloading a document multiple times, we cache the document locally using an abstraction called a Document Input Stream (DIS).

A DIS is an input stream that caches the entire contents of the document read from the internet. It also provides methods to re-read the document. The DIS can cache small documents (64 KB or less) entirely in memory, while larger documents can be temporarily written to a backing file.

Each worker thread has an associated DIS, which it reuses from document to document. After extracting a URL from the frontier, the worker passes that URL to the relevant protocol module, which initializes the DIS from a network connection to contain the document's contents. The worker then passes the DIS to all relevant processing modules.

4. Document Dedupe test: Many documents on the Web are available under multiple, different URLs. There are also many cases in which documents are mirrored on various servers. Both of these effects will cause any Web crawler to download the same document multiple times. To prevent processing of a document more than once, we perform a dedupe test on each document to remove duplication.

To perform this test, we can calculate a 64-bit checksum of every processed document and store it in a database. For every new document, we can compare its checksum to all the previously calculated checksums to see the document has been seen before. We can use MD5 or SHA to calculate these checksums.

How big would be the checksum store? If the whole purpose of our checksum store is to do dedupe, then we just need to keep a unique set containing checksums of all previously processed document. Considering 15 billion distinct web pages, we would need:

$$15B * 8 \text{ bytes} \Rightarrow 120 \text{ GB}$$

Although this can fit into a modern-day server's memory, if we don't have enough memory available, we can keep smaller LRU based cache on each server with everything backed by persistent storage. The dedupe test first checks if the checksum is present in the cache. If not, it has to check if the checksum resides in the back storage. If the checksum is found, we will ignore the document. Otherwise, it will be added to the cache and back storage.

5. URL filters: The URL filtering mechanism provides a customizable way to control the set of URLs that are downloaded. This is used to blacklist websites so that our crawler can ignore them. Before adding each URL to the frontier, the worker thread consults the user-supplied URL filter. We can define filters to restrict URLs by domain, prefix, or protocol type.

6. Domain name resolution: Before contacting a Web server, a Web crawler must use the Domain Name Service (DNS) to map the Web server's hostname into an IP address. DNS name resolution will be a big bottleneck of our crawlers given the amount of URLs we will be working with. To avoid repeated requests, we can start caching DNS results by building our local DNS server.

7. URL dedupe test: While extracting links, any Web crawler will encounter multiple links to the same document. To avoid downloading and processing a document multiple times, a URL dedupe test must be performed on each extracted link before adding it to the URL frontier.

To perform the URL dedupe test, we can store all the URLs seen by our crawler in canonical form in a database. To save space, we do not store the textual representation of each URL in the URL set, but rather a fixed-sized checksum.

To reduce the number of operations on the database store, we can keep an in-memory cache of popular URLs on each host shared by all threads. The reason to have this cache is that links to some URLs are quite common, so caching the popular ones in memory will lead to a high in-memory hit rate.

How much storage we would need for URL's store? If the whole purpose of our checksum is to do URL dedupe, then we just need to keep a unique set containing checksums of all previously seen URLs. Considering 15 billion distinct URLs and 4 bytes for checksum, we would need:

$$15B * 4 \text{ bytes} \Rightarrow 60 \text{ GB}$$

Can we use bloom filters for deduping? Bloom filters are a probabilistic data structure for set membership testing that may yield false positives. A large bit vector represents the set. An element is added to the set by computing 'n' hash functions of the element and setting the corresponding bits. An element is deemed to be in the set if the bits at all 'n' of the element's hash locations are set. Hence, a document may incorrectly be deemed to be in the set, but false negatives are not possible.

The disadvantage of using a bloom filter for the URL seen test is that each false positive will cause the URL not to be added to the frontier and, therefore, the document will never be downloaded. The chance of a false positive can be reduced by making the bit vector larger.

8. Checkpointing: A crawl of the entire Web takes weeks to complete. To guard against failures, our crawler can write regular snapshots of its state to the disk. An interrupted or aborted crawl can easily be restarted from the latest checkpoint.

7. Fault tolerance

We should use consistent hashing for distribution among crawling servers. Consistent hashing will not only help in replacing a dead host, but also help in distributing load among crawling servers.

All our crawling servers will be performing regular checkpointing and storing their FIFO queues to disks. If a server goes down, we can replace it. Meanwhile, consistent hashing should shift the load to other servers.

8. Data Partitioning

Our crawler will be dealing with three kinds of data: 1) URLs to visit 2) URL checksums for dedupe 3) Document checksums for dedupe.

Since we are distributing URLs based on the hostnames, we can store these data on the same host. So, each host will store its set of URLs that need to be visited, checksums of all the previously visited URLs and checksums of all the downloaded documents. Since we will be using consistent hashing, we can assume that URLs will be redistributed from overloaded hosts.

Each host will perform checkpointing periodically and dump a snapshot of all the data it is holding onto a remote server. This will ensure that if a server dies down another server can replace it by taking its data from the last snapshot.

9. Crawler Traps

There are many crawler traps, spam sites, and cloaked content. A crawler trap is a URL or set of URLs that cause a crawler to crawl indefinitely. Some crawler traps are unintentional. For example, a symbolic link within a file system can create a cycle. Other crawler traps are introduced intentionally. For example, people have written traps that dynamically generate an infinite Web of documents. The motivations behind such traps vary. Anti-spam traps are designed to catch crawlers used by spammers looking for email addresses, while other sites use traps to catch search engine crawlers to boost their search ratings.

Solution 2:

What Is A Web Crawler

Web crawling or web indexing is a program that collects webpages on the internet and stores them in a file, making them easier to access. Once it is fed with the initial reference pages, or “seed URLs”, it indexes the web links on those pages. Next, the indexed web pages are traversed and the web links within them are extracted for traversal. The crawler discovers new web links by recursively visiting and indexing new links in the already indexed pages.

Most Popular Applications

Search engines, including Google, DuckDuckGo, Bing, and Yahoo, have their own web crawlers to find and display pages. Other applications include **price comparison portals**, **data mining**, **malware detection**, and **web analysis tools**. **Automated maintenance** of websites will also require a web crawler.

What Features Are Involved?

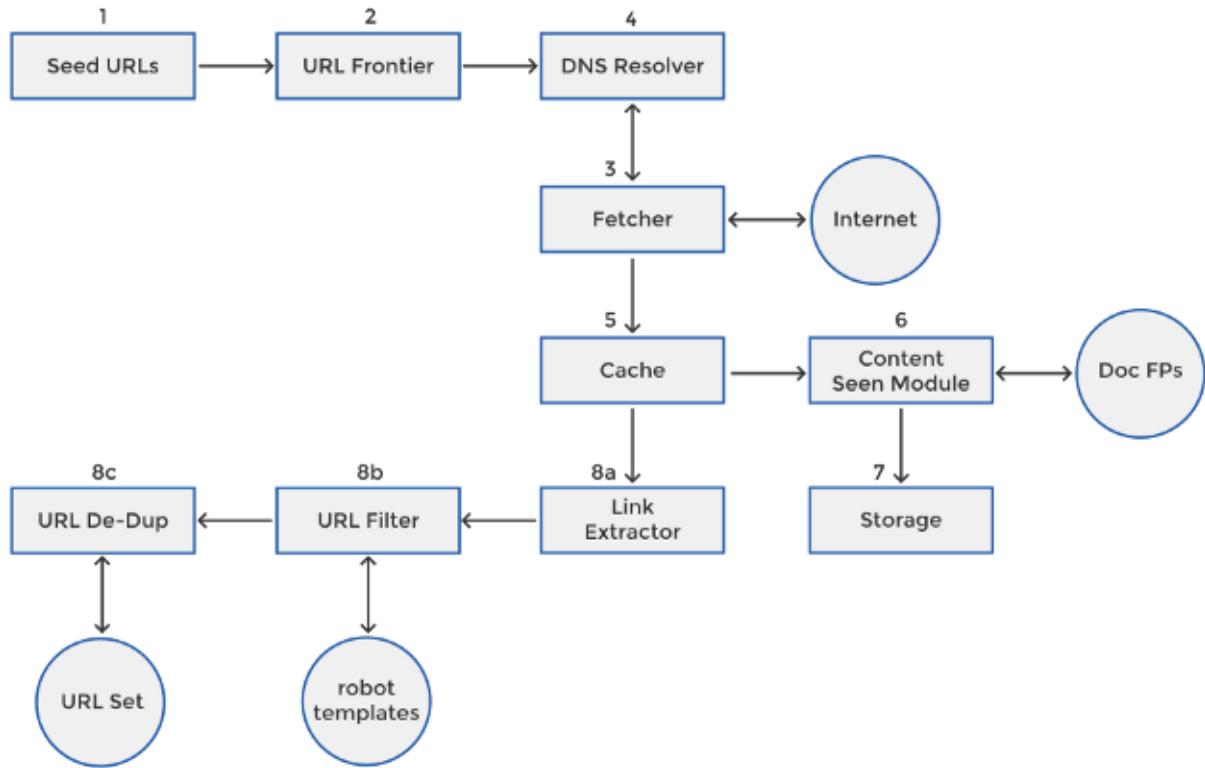
Before actually building a crawler, there are some considerations involved. Let's take a look:

- **Crawling Frequency:** Also known as crawl rate, or crawl frequency refers to how often you want to crawl a website. You can have different crawl rates for different websites. For example, news websites might need to be crawled more often.
- **Dedup:** Where multiple crawlers are used, they may add duplicate links to the same URL pool. Dedup or duplicate detection involves the use of a space-efficient system, like Bloom Filter, to detect duplicate links, so your design isn't crawling the same sites.
- **Protocols:** Think about the protocols that your crawler will cater to. A basic crawler can handle HTTP links, but you can also modify the application to work over STMP or FTP.
- **Capacity:** Each page that is crawled will carry several URLs to index. Assume an estimate of around 50 billion pages. Assuming an average page size of 100kb:
$$50 \text{ B} \times 100 \text{ KBytes} = 5 \text{ petabytes}$$

You would need around 5 petabytes of storage, give or take 2 petabytes, to hold the information on the web.

You can compress the documents to save storage since you'll not need to refer to it every time. For certain applications, such as search engines, you may only need to extract the metadata information before compressing it. When you do need the entire content of the page, you can access it through the cached file.

Design Diagram



Design Diagram

Overview

As you can see in the system design diagram, the loop is initiated through a set of 'seed URLs' that is created and fed into the URL frontier. The URL frontier applies algorithms to build URL queues based on certain constraints, prioritization and politeness, which we'll discuss in detail further in the post.

Module 3, which is the URL fetcher receives the URLs waiting in the queue one by one, receives the address against it from the DNS resolver and downloads the content from that page. The content is cached by module 5 for easier access to the processors. It is also compressed and stored after going through the document De-Dup test at module 6. This test checks to see if the content has already been crawled.

The cached pages are processed, going through different modules (8a, 8b and 8c) in the process. All the links on the page are extracted, filtered based on certain protocols and passed through the URL-Dedup test to see if multiple URLs point to the same document and discard repetitions. The unique set of URLs received through the processing modules are fed back to the URL frontier for the next crawl cycle.

Design Components

1. Input Seed URLs

Firstly, your crawler will need ‘seed URLs’. Once it has the initial input, it will continue extracting and storing data recursively. This list of seed URLs or absolute URLs is fed to the ‘URL frontier’.

2. URL Frontier

The job of module 2, the URL frontier, is to build and store a list of URLs to be downloaded from the internet. For focused or topical web crawlers, the URL frontier will also prioritize the URLs in the queue.

3. Fetching Data

Whenever the URL frontier is requested for a URL, it will send the next URL in the priority queue to module 3, the HTML fetcher. The HTML fetcher then downloads the document against the fetched URL, once the DNS resolver gives it the IP address (find details under the next heading). The crawler downloads the file based on the network protocol that the file is running. Your crawler can also have multiple protocol modules to download different file types. The fetcher, also called the worker, will invoke the appropriate protocol module to download the page on the URL.

4. DNS Resolver

Before the HTML fetcher can actually download the page content, an additional step is required. This is where the role of a DNS resolver comes in. A DNS resolver, or a DNS lookup tool, component 4 in the diagram, maps a hostname to its IP address.

Though DNS resolution can be requested from the server, it will take a lot of time to complete the step, given the big number of URLs to be crawled. Instead, the better option is to create a customized DNS resolver, as you can see in the diagram, to complement the basic crawler design. So your custom DNS resolver will give HTML fetcher the IP address of the hostname that is to be fetched. Once it has the IP address, the fetcher downloads the content on the page available on that address.

5. Caching

Next, the content downloaded from the internet by the fetcher is cached. Since the data is typically stored after being compressed and can be time-consuming to retrieve, an open-source data structure store, such as Redis can be used to cache the document. This makes it easier for other processors in your web crawler design to fetch the data and re-read it without consuming unnecessary time.

6. Content Seen Module

Another aspect to consider is whether the URL content is already seen by the crawler. Sometimes, multiple URLs can have the same content in them. If the document is already in the crawler database, you’re going to discard it here without sending it to storage.

We’ll use module 6, the content seen module or the document De-Dup test, so that the crawler doesn’t download the same document multiple times. Fingerprint mechanisms, like [checksum](#) or [shingles](#), can be used to detect duplication. The checksum of the current

document is compared to all the checksums present in a store called ‘Doc FPs’ to see if the file has already been crawled. If the checksum already exists, the document is discarded at this point.

7. Storage

If the document passed the content seen test in the previous module, it's saved into the persistent storage.

8. Processing Data

You can have multiple processors in your customized web crawler design, depending on what you plan on doing with the crawler. All the processing is carried out on the cached document, rather than the stored database since it's easier to retrieve. The three most common processors that are almost always present include:

8a. Link Extractor

URL extractor or link extractor can be taken as the default processor. A copy of the crawled page is fed into the link extractor from Redis or any other in-memory data store. The extractor will parse the network protocol and extract all the links on the page. The links could either be pointing to

- a specific location on the same page,
- a different page on the same website,
- or a different website.

A set of normalization techniques will need to be incorporated to make the link list more manageable. The links in the list should follow a standard format to make them easily understandable by the crawler modules. You can:

- Map all child domains to the main domain. For example, links from mail.yahoo.com and music.yahoo.com can all be tagged under www.yahoo.com.
- If the components of the link are uppercased, convert them to lowercase
- Add network protocol to the beginning of the link, if it's missing
- Add/remove backslashes at the end of the link

8b. URL Filtering

URL filter receives the set of unique, standardized URLs from the link extractor module. Next, depending on how you're using the web crawler, the URL filter will filter out the files that are required and discards the rest.

You can design a URL filter that filters by filetype. For example, a web crawler that crawls only jpg files will keep all the links that end with ‘.jpg’ and discard the rest. Other than the filetype, you can also filter links by their prefix or domain name. For example, if you don't want to crawl Wikipedia links, you can design your URL filter to ignore the links pointing to Wikipedia.

It is at this point that we can implement the robot exclusion protocol. Since the URL fetcher will already have fetched a document called robot.txt and mapped the off-limit pages to the URL list, the URL filter will discard all the links that the website does not permit downloading. We'll discuss the need for robot exclusion protocol later in the post.

The output of the URL filter is all the URLs that we want to keep and pass on to the URL frontier after some further processing.

8c. URL De-Dup

URL De-Dup is typically implemented after the URL filter module. The stream of URLs coming out of the URL filter might not be unique. You may have multiple URLs in the stream that point to the same document. We wouldn't want to crawl the same document twice, so a De-Dup test is performed on each filtered link before passing it ahead.

Your crawler should ideally store a database of all the crawled URLs — let's call it the URL set. Each URL to be tested is mapped onto each of the URLs in the set to detect a repetition.

The Bloom Filter

The Bloom filter is a space-efficient option to check if a URL is already present in the database. If it's already in the crawled database, it's discarded, and if not, it's passed on to the next module. However, the Bloom filter isn't always reliable. Though a false negative isn't possible, there are chances of a false positive. If the Bloom filter decides, falsely, that the URL is already present in the set, the URL will not be passed to the URL frontier. It's not a major issue because you might encounter the same URL in one of the next iterations of crawling and it may be added to the list one of those times.

Cycle Repeats

After being passed through the URL De-Dup test, the unique URLs are saved to the URL set and also pushed to the URL frontier to repeat the cycle.

Design Considerations

Now that we understand a bit about the system components, there are certain considerations that you also need to look into, to build an efficient and functional crawler. These considerations decide the order in which URLs are returned by the URL frontier. Both these functions, prioritization and politeness will be implemented in the frontier since that's the module that positions the URLs in the queue.

How To Prioritize

Since there are countless web pages on the internet, your web crawler should be able to prioritize which pages to download for its efficient operation. Priority decides how often the webpage will be recrawled. The high-quality pages that change their content frequently will need to be crawled more often. Priority of crawled webpages, therefore, is based on two factors:

1. The quality of a page
2. The change rate of the page

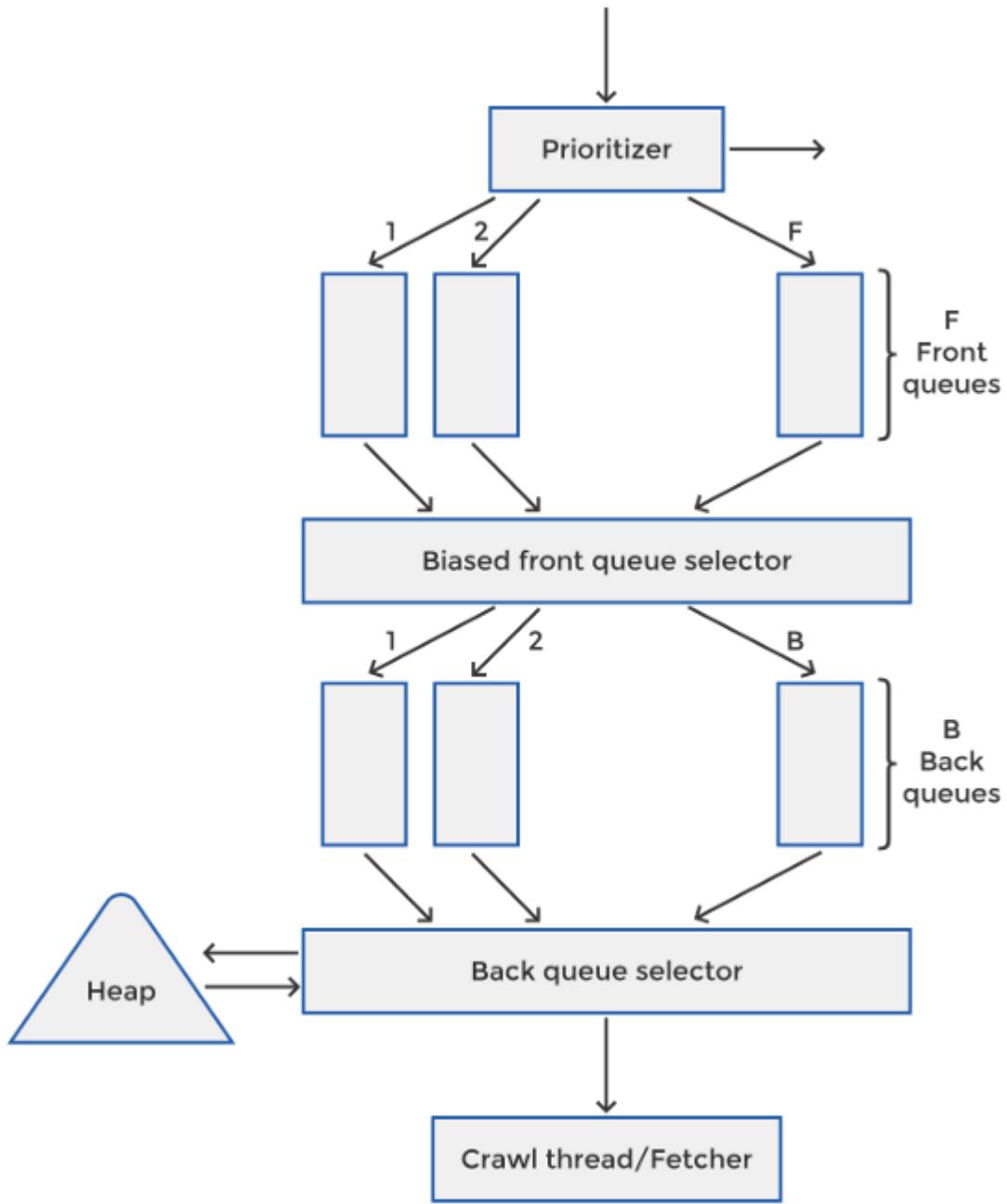
If you're only going to prioritize by change rate, your crawler will encounter several spam pages that change their content completely each time they're fetched. It's, therefore, essential to put some form of quality check as well.

For example, for a news page that updates content frequently, your crawler will need to recrawl the link every few minutes to incorporate any changes.

How To Implement Politeness

The second consideration, also implemented at the URL frontier, is politeness. Your crawler should ensure that it's not overwhelming the host's server at any time. There needs to be a delay between repeated fetching requests to a particular server. The delay must be greater than the time taken for the latest page fetched from the host. Also, at any given time, make sure that only a single connection is open between the crawler and the host.

URL Frontier Architecture Diagram



URL Frontier Architecture Components

The diagram explains how these considerations are implemented by the URL frontier.

The URL frontier architecture makes sure:

1. Pages are crawled according to their priority.
2. Only one connection is established to a host at any given time.
3. A wait time is maintained between successive requests to a host.

Two sets of queues are maintained — F number of front queues, as shown in the upper portion of the diagram, and B number of back queues, shown in the lower portion of the diagram. All the queues follow the FIFO protocol.

The front queue implements the prioritization, while the back queue implements the politeness. Back queues will ensure that only a single connection is open to a host and also that there is a wait time between successive requests to a host.

Front Queues

When a URL is received by the prioritizer, it assigns it a priority number between 1 and F based on its crawl history. For example, if the URL receives a priority i it's added to the ith front queue. Each of the F queues will have multiple URLs to crawl, they can also be empty.

Biased Front Queues Selector

Based on their queue number, the URLs are pushed by the biased front queue selector to the back queues. The choice of the selector is biased since it fetches links from higher priority queues first.

Back Queues

There are two features that the back queues maintain:

1. The back queues can not be empty while the crawl is in progress. As soon as a queue becomes empty, it will ask the front queue selector for the next URL, which could be from a different host, from one of the front queues.
2. Each back queue will have URLs from a single host only. This feature ensures that no more than a single connection is established between the crawler and the host.

Note: We can only set multiple connections with the host if the host website agrees to it.

Back Queue Table

A table is maintained to address the back queue number to the host such as the one below. With this table we know that, for example, back queue number 4 is reserved for pages from the host wikipoedia.com only.

Whenever a back queue is empty, the biased front queue selector pushes a URL from one of the front queues, from a different host, to the back queues and the table is updated.

Host	Back queue
wikipedia.com	4
thespruce.com	1
...	B

Back Queue Heap

In addition to the table, there is also a “Heap” which carries information for each back queue. This information is the minimum time to wait before a host can be contacted again. When the fetcher contacts the back queue selector for a URL, the heap will tell which back queue to pick the link from. Once a URL is crawled, the time against the back queue is updated in the heap.

Back Queue Selector

When a crawling thread is free, the fetcher asks the back queue selector for a link to crawl. The back queue selector contacts the heap to check which back queue can the URL be fetched from and at what time. The crawler may have to wait for the amount of time corresponding to that queue in the heap. The back queue picks the URL from that queue based on FIFO and gives it to the fetcher.

Robot Exclusion Protocol

Most web crawlers implement the robot exclusion protocol, allowing webmasters to put certain restrictions on the content of their website that can be accessed by the crawler. The host carries a special file called ‘robot.txt’ which is required to be fetched by the crawler. It contains information regarding the website pages that are off-limits for the crawler and also the frequency at which the host can be contacted.

‘Robot.txt’ is used by the crawler to check whether each URL fetched from the host passes the robot restrictions. This check takes place at the URL filter module.

Detecting Updates

If our crawler can know how frequently a website is updating (or changing content) it can adjust the crawl rate accordingly. One way to do so is this:

Instead of downloading an entire page, our crawler can simply make a head request. This request returns the last modified time for the page. If this time is the same as the last crawl time of the page, then the crawler won’t download it.

Detecting Duplicates

Plenty of duplicate content exists on the internet. Web Pages may not be exactly the same, but it's very likely that your crawler will encounter multiple sites with almost the same content. Spammers who steal content from a blog to make a few changes and add it to a different blog or page are one such example. To save storage space and time, we need our crawler to be intelligent enough to detect such duplicates.

Now, with millions of pages on the internet, it may not be feasible to check them word by word to see if it's a copy. Our content seen module already calculates and compares the checksum for every page. However, since the entire content may not be the same, checksum, or hashing won't detect pages that are almost duplicates. More suitable techniques to detect near duplicate pages include Simhash, which is the one that Google search engine uses.

Crawling VS Scraping

Crawling and scraping are often confused, so it's good to understand the difference before you head to a major interview. A crawler scans a page, indexes the URLs on it and then goes to the next page to crawl.

Web scrapers, often used in conjunction with crawlers, collect specific information on web pages or other sources to do some processing over it. For example, the price list can be extracted from different sources and saved into an excel spreadsheet using a web scraper.

A major feature that sets web scrapers apart from web crawlers is the focused approach. Rather than scanning the entire content, web scrapers are after a specific portion of the document, from where the information needs to be pulled. Crawler, on the other hand, extracts all the data from pages and indexes the links.

1. how to distribute work ?

Basically each machine should be responsible for a subset of urls, we need to divide the entire url set into subsets, such that :

- Machines might be heterogeneous. that means some machines might be faster than other machines. Ideally the partition algorithm should be able to assign partitions according to a machine's capacity.
- Maximize network usage. If we assign urls belonging to the same domain to the same machine, we may save time rebuilding TCP connections; also, in real world that the servers most likely have some rate limit mechanism, which means we need to control the speed for sending requests to the same server. Sending urls belonging to the same domain on the same machine makes it much easier to control such speed. This may not be a problem if we are asked to download Wiki pages (since all of those urls belong to the same domain).
- Tolerable to membership change. In a real system when network partition occurs, or a hardware fails, some machines might not be available; on the other hand, we can

always add more machines to even the workload. The partition algorithm should try to minimize the work when membership change happens.

In practice we can first map each url to a non-negative 64-bit integer, and divide the whole range into Q sections ($Q \gg M$, the number of machines). Then we randomly assign each section to machines based on machine's capacity. When a new machine joins, it "steals" some partitions from existing machines. The "stealth" is a good chance of balancing the workload of the cluster—for example, it can simply steal partitions from the machines with the highest workload.

Note that a common term mentioned in literature or distributed system blogs is "consistent hashing". However, it's very easy to get misused without tuning it carefully.

Also note that the even distribution of urls may not result in even workload on different machines, because the size of the page, the geo-location of the servers, and the workload of the servers may all be different. To achieve a real even workload, we need to rebalance workload when necessary. Technically the process of "rebalance" can be the same as "stealth" (when membership change happens), and it's a manual process in a lot real-world systems.

2. how to coordinate ?

Each machine may download urls that need to be handled by other machines. When it happens, how to forward such urls to other machines ?

The answer to this question would affect the architecture of the system. In a master-slave system where a single master exists for all the coordination work, while all the slave nodes can be simply stateless, all urls should be forwarded to the master, and the master will forward urls to machines. There's no communication between slave nodes at all; On the other hand, in a peer to peer system, there is no single master, all nodes have to know where to forward a url.

It would be a long list to compare the pros and cons of these two architectures. However, the problem gives a good hint : minimize the traffic between nodes. This indicates that a peer-to-peer architecture would be a better one since its overall traffic is only half of a master-slave architecture.

3. how to make sure each URL is downloaded exactly once ?

Since we already decide to forward url based on hash partition, that means the same url will always be handled by the same machine (given no machine failure). In a single machine we can use either a hash table or a bloom filter (if the number of urls to download is huge) to achieve this.

So far we got a rough design that works. There are a few follow-up domains to further discuss (you can briefly discuss with interviewers during interview). In this article I will focus on a common follow-up problem : fault tolerance. i.e., what happens if a node fails ?

4. how to make it fault tolerant ?

A node can fail at any time, and recover later (ex., restart, temporary network partition). When such failure happens, we need to find other nodes to continue the work of the failed node; also for those nodes who are forwarding urls to the failed node, they should forward to a different one.

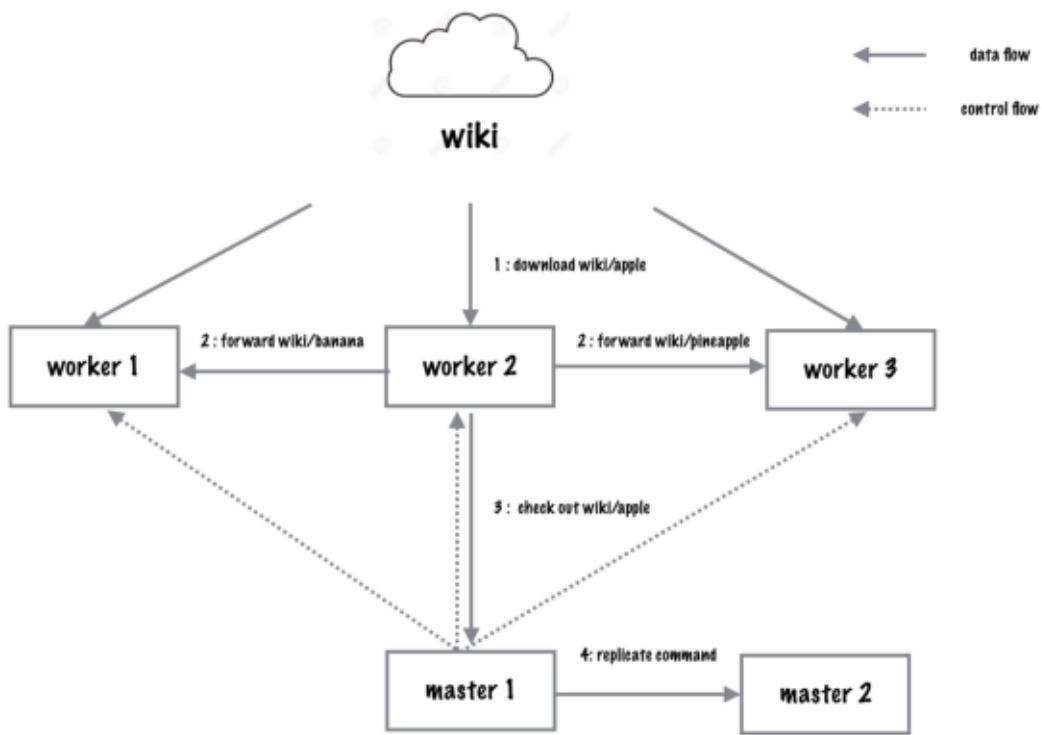
The second case is relatively easy to solve. We can borrow the idea of “consumer group” from [3], i.e., two or more nodes can form a group, ex., node 1, 2, 3 forms a group. Normally node 4 is sending urls to node 1, when it finds that node 1 is unreachable, it may choose another node from the group, say node 2, and always forward new urls to 2 going forward (even though node 1 recovers later). This preserves the “exactly once downloading” property.

The first case is more difficult : how can we find a node to continue node 1’s work when it fails ? and what should we do when it recovers ? Unfortunately, to detect a node failure, and to find a new node to replace the failed node while preserving the “exactly once downloading” property, we need a master node. A master node plays two roles, first, it sends the heartbeat periodically to all worker nodes, so it can detect a node failure; second, each worker node should check out its current progress to some external system (or to master node), so its work can be continued when it fails. Keeping a master node simplifies the architecture.

Note that “exactly once downloading”, as described in the problem cannot be achieved accurately with the architecture described above. In practice we cannot distinguish between “downloaded the page successfully, but failed to check out the progress” and “failed to download the page”. It’s better to avoid using the term “exactly once” here.

Put it together

The description above yields a simple architecture below :



0. start up stage : deploy metadata to all workers. meta data include seed urls, and decisions about how to divide urls to partitions, assign partitions to each worker, and which workers form a group.

1. worker 2 downloads the page *wiki/apple*, and extract *wiki/banana* and *wiki/pineapple*.
2. based on the meta data, worker 2 decides to forward *wiki/banana* to worker 1, and forward *wiki/pineapple* to worker 3. If either worker 2 or worker 3 does not respond, worker 2 marks them as “fail” locally, and choose another group member to forward.
3. after downloading *wiki/apple*, worker 1 checks this progress out to master 1.
4. master 1 replicates the data to master 2, in case itself fails.
5. master 1 sends heartbeat to all workers regularly to detect node failure. Once detected, it assigns the work left on the failed node to another node, and update meta data on all workers.

8. Designing Instagram

refer:<https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8IG>

You need to design a social media service for billions of users. Most of the interviewers spend time discussing news feed generation services in these apps.

Features to be considered:

- Some of the specific Twitter/Facebook/Instagram features to be supported.
- Privacy controls around each tweet or post.
- Users should be able to post tweets also the system should support replies to tweets/grouping tweets by conversations.
- Users should be able to see trending tweets/posts.
- Direct messaging
- Mentions/Tagging.
- The user should be able to follow another user.

Things to analyze:

- The system should be able to handle the huge amount of traffic for billions of users.
- Number of followers
- The number of times the tweet has been favorited.

Components:

- News feed generation.
- Social graph (Friend connection networking between users or who follows whom?—? especially when millions of users are following a celebrity)
- Efficient storage and search for posts or tweets.

Solution 1:

System design is one of the **most significant** parts of software engineering. It is intimidating to start designing a system. One of the main reasons is that the terminologies described in software architecture books are hard to understand. And there are no fixed guidelines. Everywhere you check seems to have a different approach. It becomes confusing for a new learner.

Definition of the System:

We need to clarify the goal of the system. *System design is such a vast topic; if we don't narrow it down to a specific purpose, it will become complicated to design the system, especially for newbies.*

Instagram is a photo-sharing service that enables its users to upload and share their photos and videos with other users, mostly their followers. As this is an exercise, we will design a simpler version of the main Instagram service.

In this service, a user can share photos and follow other users. There will be a newsfeed for each user. The 'News Feed' consists of top photos of all the people the user follows.

Requirements of The System:

In this part, we select the features of the system. The system requirements are divided into two parts:

Functional Requirements:

This type of requirement what the system has to deliver. You may say it is the main goal of the system.

Firstly, for Instagram, the users should be able to upload/download/view photos. Users may perform searches for images based on photo/video titles. One User can follow other users. Another essential feature is that the system should display a user's News Feed that consists of photos of all the people the user follows.

As this is a practice for the system, we are not considering tags in photos, tag-based search, like and comment features, etc.

Non-functional Requirements

Performance, availability, consistency, scalability, reliability, etc., are important quality requirements in system design. We need to analyze these requirements for the system.

As a system designer, we might want to have a design that will be highly available, very high performant, top-notch consistency in the system, highly secured system, etc. But it's not possible to achieve all these targets in one system. We need to have requirements that will work as restrictions on the design of a system. So, let's define our NFRs:

Our system should be highly available. In the case of any web service, it's a mandatory requirement. Home page generation latency should be at most 200 msec. If the home page generation takes too long, users will be dissatisfied, which is not acceptable.

As we choose for the system's high availability, we should keep in mind that may hamper consistency across the system. The system should also be highly reliable, which means any uploaded photo or video by users should never be lost.

In this system, photos search and views would be more than uploading. As the system would have more read-heavy operations, we will focus on building a system that can retrieve photos quickly. While viewing photos, latency needs to be as low as possible.

Data flow:

If you are not sure where to start in a system design, always start with the data storing system. It will help to keep your focus aligning with the requirements of the system.

We need to support two scenarios at a high-level, one is to upload photos, and another is to view/search photos. Our system would need some [object storage](#) servers to store photos and some database servers to store metadata information.

Defining the database schema is the first phase of understanding the data flow between different components of the system. We need to store user profile data like the follower list, uploaded photos by users. There should be a table that stores all data regarding photos. If we want to show recent photos first, we need to index (PhotoID, CreationDate). Tables in the DB could be:

Table Name : table indexe1,index2 etc

Photo: PhotoID (pk), UserID, PhotoLocation, CreationDate

User: UserID(pk), Name, Email, DOB, LastLoginTime

Follow: UserID1, UserID2 (paired pk)

FeedItem: FeedID(pk), UserID, Contents, PhotoID, CreationDate

****pk = primary key**

We may store the above tables in an RDBMS like MySQL as we will require joins between tables. But relational databases have some challenges of their own. Though it is possible to scale a relational database across multiple servers, it is a challenging process. We may store photos in a distributed file storage like [HDFS](#) or [S3](#).

When users upload photos, storing them in the database is a slow process as they are stored on a disk. But in the case of reads, the process will be faster, especially if we serve them from the cache.

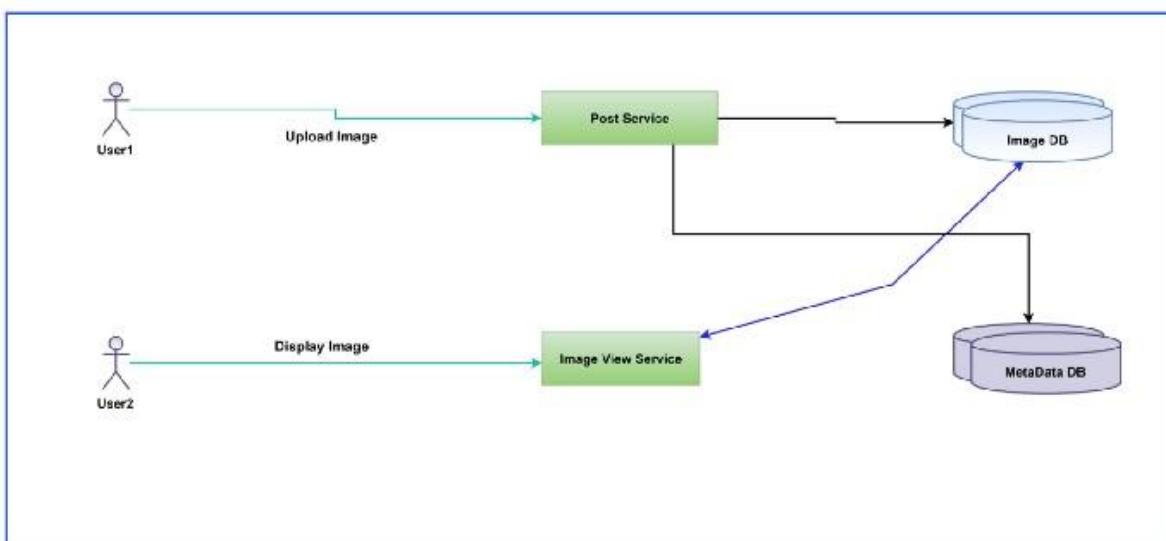


Figure: Separate Upload and Download service (Image by Author)

As uploading is a slow process, if we put both write and read photos on the same server, the system may get too busy with all the 'write' requests. We have to consider web servers have a connection limit before designing our system.

Let's say a server can have a maximum of 5000 connections at any time. So, it can't have more than 5000 concurrent uploads or downloads. To handle this type of bottleneck, we can use **responsibility segregation**.

We can split reads and writes into two separate services. So, we will have some servers for reads and different servers for writes. This separation will also give us a chance to scale each operation independently.

Availability and Reliability:

We can not lose the photos uploaded by the users. Reliability is a huge factor for this system. So, we need to have more than one copy of each file. In that case, if one server is dead, we still have other copies of those photos.

We can not have a single point of failure in a system. If we keep multiple copies of a component, it may remove the single point of failure in the system. For making a highly available system, we need to keep multiple replicas of the services in the system. So, even if some of the servers are down, the system may remain available. Creating redundancy in a system also provides a backup in a crisis.

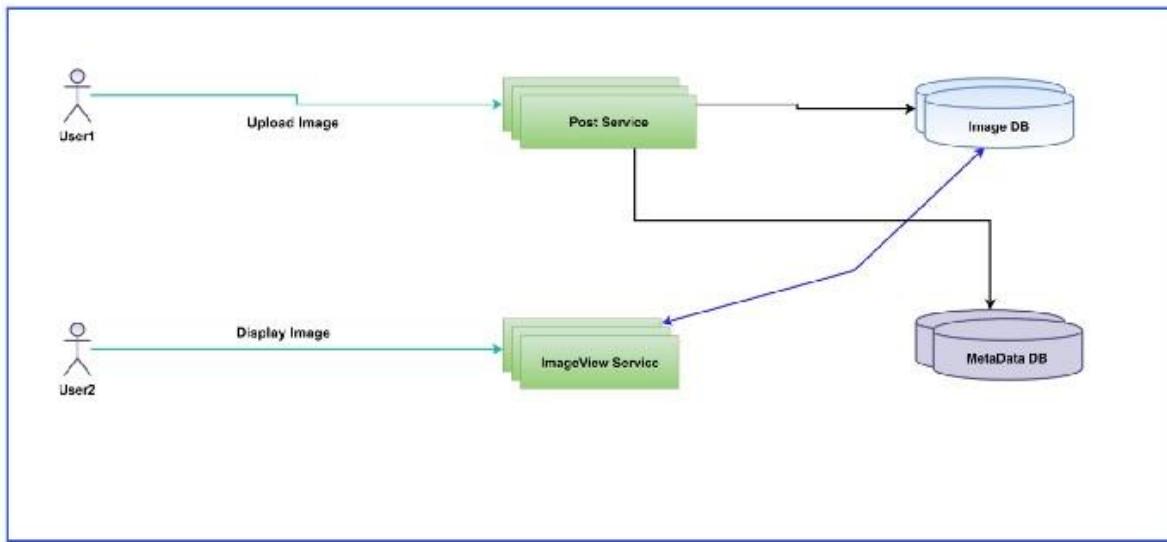


Figure: Multiple copies of services and databases for availability(Image by Author)

Scalability:

For supporting millions of users, we need to partition our database to divide and store our data into different DB servers. We can use database sharding for metadata.

Partitioning based on UserID:

If we partition metadata DB based on 'UserID,' we may keep all the user photos in the same shard. One User seems to have almost 3 TB of data. If one DB shard is 1TB, we will need three shards of data of each User.

For better performance and scalability, we may keep ten shards. In that case, we can find the shard number by $User\text{Id} \% 10$ and store data there.

We may face some problems in this approach if a user is a popular celebrity. That portion of shards will be hit frequently.

Besides, some users may upload more photos compared to others. In that case, the distribution might not be uniform.

More importantly, if we can not store all user pictures in one shard, we may need to distribute photos into different shards.

And if a shard is down, we may face an unavailability issue for a user.

Partitioning based on Photoid:

Another way of partitioning DB can be by using Photoid. We need to create a unique ID for each photo. Each DB shard needs to have its auto-increment sequence for Photoids, but then each shard will have the same photo iD in its portion. We can add ShardId with each photoid; this can make the photoid unique within the system.

Load balancer:

As we use multiple copies of a server, we need to distribute the user traffic to those servers efficiently. The load balancer will distribute the user requests to various servers uniformly. We can use IP-based routing for the Newsfeed service, as the same user requests go to the same server, and caching can be used to get a faster response.

If there are no such requirements, round-robin should be a simple and good solution for the server selection strategy of load balancers.

API GateWay:

We have a lot of services for our system. Some will generate newsfeed, some help storing photos, some viewing the photos, etc. We need to have a single entry point for all clients. This entry point is API Gateway.

It will handle all the requests by sending them to multiple services. And for some requests, it will just route to the specific server. The API gateway may also implement security, such as verifying the client's permission to perform the request.

NewsFeed Generation:

The News Feed for any user contains the latest, most popular, and relevant photos of the people that the User follows. If we generate the newsfeed in real-time, it would be very high latency. So, we will try to pre-generate the newsfeed. So, the Post service will store the photos in the database. Newsfeed service precomputes feed for all the users.

We need to have specific servers which would continuously generate user newsfeed and store them in the database. So, when the User needs the latest photos for the newsfeed, we need to query the table.

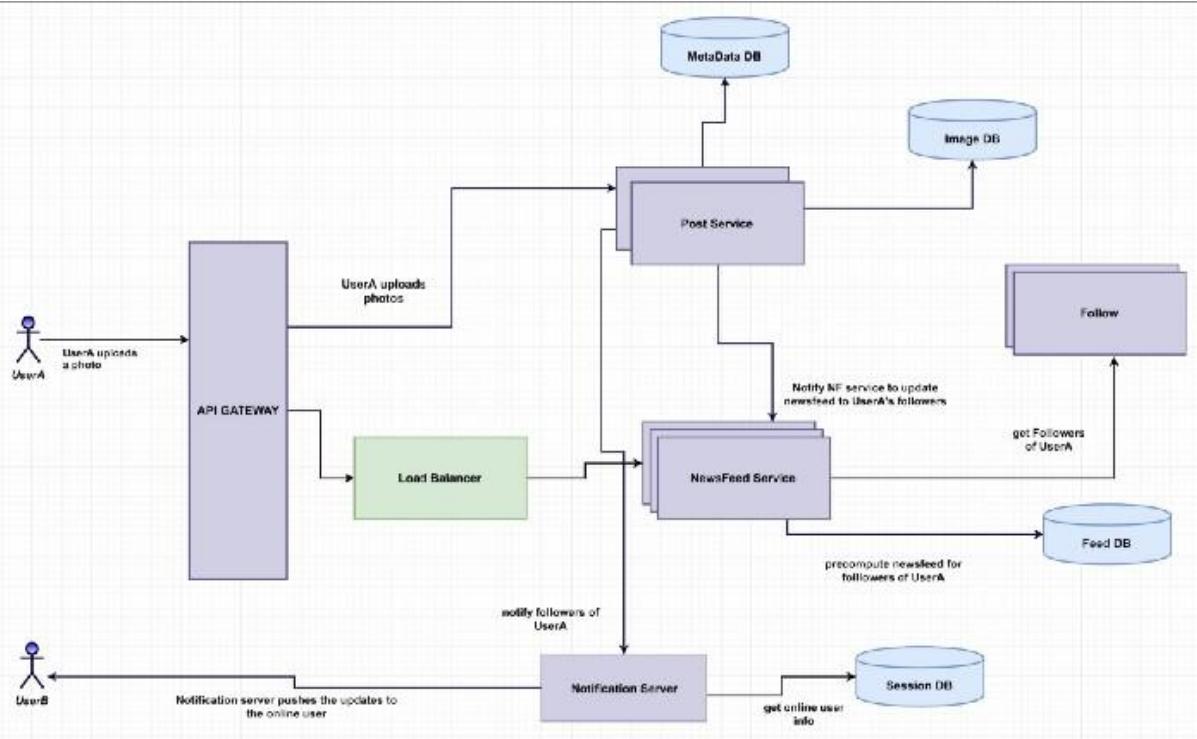


Figure: Newsfeed precomputation and notification(Image by Author)

Now, how does a user get the latest newsfeed from the server? We may consider below approaches:

Pull-Based Approach

In this approach, each User may poll the server after a regular interval to check if any friend has a new update. The server has to find all the user connections and check for each friend to create a new post. If there are new posts, the database's query will get all the recent posts created by a user's connections and show them on their home page.

The users may ping the Newsfeed servers for pre-generated feeds after a regular interval. The lower the time interval, the more recent data will be found in the feed.

This approach is time-consuming. As users are polling the database after a regular interval, it will put a huge amount of unnecessary load on the database.

Push-Based Approach

In this approach, servers can push new data to the users as soon as it is available. Users have to maintain a long polling request with the server for receiving the updates.

As the Newsfeed service is pre generating the feeds, the Notification service will notify the active users about the new posts. Session DB will keep data of the user's connections who are online.

You may check the various notification approaches at [this link](#).

We may implement a pull-based model for all the users who have a high number of followers. And we can use a push-based approach for those who have a few hundred followers.

Pagination:

The newsfeed of users can be a large response. So, we may design the API to return a single page of the feed. Let's say we are sending at most 50 posts every time a feed request is made.

The user may send another request for the next page of feeds for the next time. And within that period, if there is not enough feed, Newsfeed may generate the next page. This technique is called pagination.

Solution 2:

1. What is Instagram?

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

We plan to design a simpler version of Instagram for this design problem, where a user can share photos and follow other users. The 'News Feed' for each user will consist of top photos of all the people the user follows.

2. Requirements and Goals of the System

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

Functional Requirements

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

Non-functional Requirements

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

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

3. Some Design Considerations

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

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

4. Capacity Estimation and Constraints

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

$2M * 200KB \Rightarrow 400 \text{ GB}$

- Total space required for 10 years:

$400\text{GB} * 365 \text{ (days a year)} * 10 \text{ (years)} \approx 1425\text{TB}$

5. High Level System Design

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

6. Database Schema

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

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

Photo		User		UserFollow	
PK	<u>PhotoID: int</u>	PK	<u>UserID: int</u>	PK	<u>FollowerID: int</u> <u>FolloweeID: int</u>
	UserID: int PhotoPath: varchar PhotoLatitude: int PhotoLongitude: int UserLatitude: int UserLongitude: int CreationDate: datetime		Name: varchar Email: varchar DateOfBirth: datetime CreationDate: datetime LastLogin: datetime		

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

We can store photos in a distributed file storage like [HDFS](#) or [S3](#).

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

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

7. Data Size Estimation

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

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

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

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

500 million * 68 ≈ 32GB

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

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

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

$2M * 284 \text{ bytes} \approx 0.5\text{GB per day}$

For 10 years we will need 1.88TB of storage.

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

$500 \text{ million users} * 500 \text{ followers} * 8 \text{ bytes} \approx 1.82\text{TB}$

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

$32\text{GB} + 1.88\text{TB} + 1.82\text{TB} \approx 3.7\text{TB}$

8. Component Design

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

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

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

9. Reliability and Redundancy

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

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

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

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

10. Data Sharding

Let's discuss different schemes for metadata sharding:

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

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

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

What are the different issues with this partitioning scheme?

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

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

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

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

KeyGeneratingServer1:

```
auto-increment-increment = 2  
auto-increment-offset = 1
```

```
KeyGeneratingServer2:  
auto-increment-increment = 2  
auto-increment-offset = 2
```

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

Alternately, we can implement a ‘key’ generation scheme.

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

11. Ranking and News Feed Generation

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

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

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

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

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

- 1. Pull:** Clients can pull the News-Feed contents from the server at a regular interval or manually whenever they need it. Possible problems with this approach are a) New data might not be shown to the users until clients issue a pull request b) Most of the time, pull requests will result in an empty response if there is no new data.
- 2. Push:** Servers can push new data to the users as soon as it is available. To efficiently manage this, users have to maintain a [Long Poll](#) request with the server for receiving the updates. A possible problem with this approach is a user who follows a lot of people or a celebrity user who has millions of followers; in this case, the server has to push updates quite frequently.
- 3. Hybrid:** We can adopt a hybrid approach. We can move all the users who have a high number of followers to a pull-based model and only push data to those who have a few hundred (or thousand) follows. Another approach could be that the server pushes updates to all the users not more than a certain frequency and letting users with a lot of updates to pull data regularly.

12. News Feed Creation with Sharded Data

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

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

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

$86400 \text{ sec/day} * 365 \text{ (days a year)} * 50 \text{ (years)} \Rightarrow 1.6 \text{ billion seconds}$

We would need 31 bits to store this number. Since, on average, we are expecting 23 new photos per second, we can allocate 9 additional bits to store the auto-incremented sequence. So every second, we can store $(29 \Rightarrow 512)(2^9 \Rightarrow 512)(29 \Rightarrow 512)$ new photos. We are allocating 9 bits for the sequence number which is more than what we require; we are doing this to get a full byte number (as $40\text{bits}=5\text{bytes}$ $40\text{ bits} = 5 \text{ bytes}$). We can reset our auto-incrementing sequence every second.

We will discuss this technique under 'Data Sharding' in [Designing Twitter](#).

13. Cache and Load balancing

Our service would need a massive-scale photo delivery system to serve globally distributed users. Our service should push its content closer to the user using a large number of geographically distributed photo cache servers and use CDNs (for details, see [Caching](#)).

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

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

Solution 3:

1. Understand the System

Before we go and work on the design, let's pause for a minute and focus your minds about what we are asked to design. In this case, do you know what Instagram is? What does this company or application do?

It was very important that you have experience in using or at least have some knowledge about the system you are about to design else you have to start asking the right question to get to know about the system first. You can signup for Instagram and get to experience some of the features and inner working off the platform first hand.

What is Instagram?

Instagram is a social media app that allows users to share photos and videos with others. The visibility of these posts (photos/videos) can be set as private and public by the original poster. Users can like posts and comment on the posts. Users can follow other users and view their News Feed (a collection of posts from the users they are following).

Also, users can search for posts across the platform. Instagram also supports other features like image editing, Location tagging, Private messaging, Push notifications, Group messaging, Hashtags, Filters and more.

Instagram reported 500+ million daily active Stories users worldwide, Stories is a feature of the app allowing users post photo and video sequences that disappear 24 hours after being posted. Most of the interactions on the platform happen through the Android and iOS mobile apps instead of their website.

Let's gather the requirements and define the scope of the design problem. Ask questions to clarify use cases and constraints. Discuss any assumptions that you can think of.

2. Define Functional Requirements

We just outlined what the platform does at a very high level, but trust me, the actual system has a lot many features and functionalities that we cannot think right away or cater to here. Keep in mind that you only have 30 minutes to 45 minutes to present your design from start to finish in the actual interview. So it is very important that you focus on key features that are essential to the platform and more importantly look for guidance from your interview about which all features and aspects he/she likes you to cover.

Use Cases

The following use-cases are in scope:

1. Users should be able to upload their photos and videos.
2. Users should be able to like a post posted by other users.
3. Users should be able to follow other users.
4. Users should be able to view NewsFeed from other users they are following.

Not in Scope

The following use-cases are out scope:

- Push Notification to users for updates.
- Users are able to search Post across the platform.
- Photo Filters, Post Comments & Location-based tagging.
- User stories automatically expiring at 24 hours.
- User messaging to an individual user or a group.

3. Define Non-Functional Requirements

Now let's define the non-functional requirements as assumptions and constraints.

Assumptions

- Traffic is evenly distributed across geography and time of the day.
- The system is read-heavy as most of the users are consuming content and not uploading content. Read Write ratio of 80 / 20 i.e. 80% read v/s 20% writes.
- Users can upload one photo or video at one time.
- News Feed is limited to Top liked posts from the users that the user is following.
- Data storage for the user and related data records is very small compared to the data storage required for photos and videos.

Constraints

- The system should be performant i.e. have very low latency for display photos, videos and News Feed to users.
- The system should be durable, any photos or videos uploaded by the user should not get lost.
- The system should be highly available and support failure over.

- The system should be able to scale to support 500+ million monthly active users.
- The system may not be consistent (eventual consistent) e.g. there could be a delay in showing photos/videos or the contents of the NewsFeed may not include all content.

4. Perform Estimations

We take relatively low numbers of users compared to the actual numbers so as to do calculations bit quickly. However, this does not change our design or approach.

Let's perform back the envelope of calculations for our system:

Assume the total users = 100 million.

Let total monthly active users (MAU) = 60 % = 60 million.

Daily active users = 60million / 30 days = 2million per day.

Assuming that each user uploads 2 photos on average, so total photos upload in day = $2 \times 2\text{million} = 4\text{ million}$.

Assuming that each user uploads 0.5 videos on average, so total videos upload in day = $0.5 \times 2\text{million} = 1\text{million}$.

Throughput Estimations

Assuming there is some API to allow to upload photos & videos and there is some API to view each one or as a collection.

Total write transactions in day = 4 million photos + 1 million videos = 5 million.

Since the read: write ratio is 80: 20, total read transactions = $5\text{ million} \times 80/20 = 20\text{ million}$.

Thus total TPS (transactions per second) : $5\text{ million} + 20\text{ million} = 25\text{ million per day} \sim 300\text{ per second}$.

Storage Estimations

Photo

Assume each photo is 200kb, so total storage for photos per day is $4\text{ million} \times 200\text{k} \sim 800\text{ GB per day}$.

Total photo storage for 10 years = $10\text{ years} \times \sim 300\text{ days a year} \times 800\text{ GB per day} \sim 2,400\text{ TB} \sim 3\text{ PB}$

Video

Assume each video is 50MB, so total storage for videos per day is $1\text{million} \times 50\text{MB} \sim 50\text{ TB per day}$.

Total video storage for 10 years = 10 years x ~ 300 days a year x 50 TB per day ~ 150 PB.

Total Storage = 3 PB (Photos) + 150 PB (Videos) = 153 PB in next 10 years

Bandwidth Capacity

Upload Bandwidth

Monthly upload bandwidth = 30 (days) x (3 TB (Photos) + 50 TB(Videos)) ~ 1.6 PB ~ 2PB per month.

Download Bandwidth

Use the read-write ratio of 80: 20 to calculate the download capacity.

Monthly download bandwidth = 2PB (Monthly Upload capacity) x 80 / 20 = 8 PB per month.

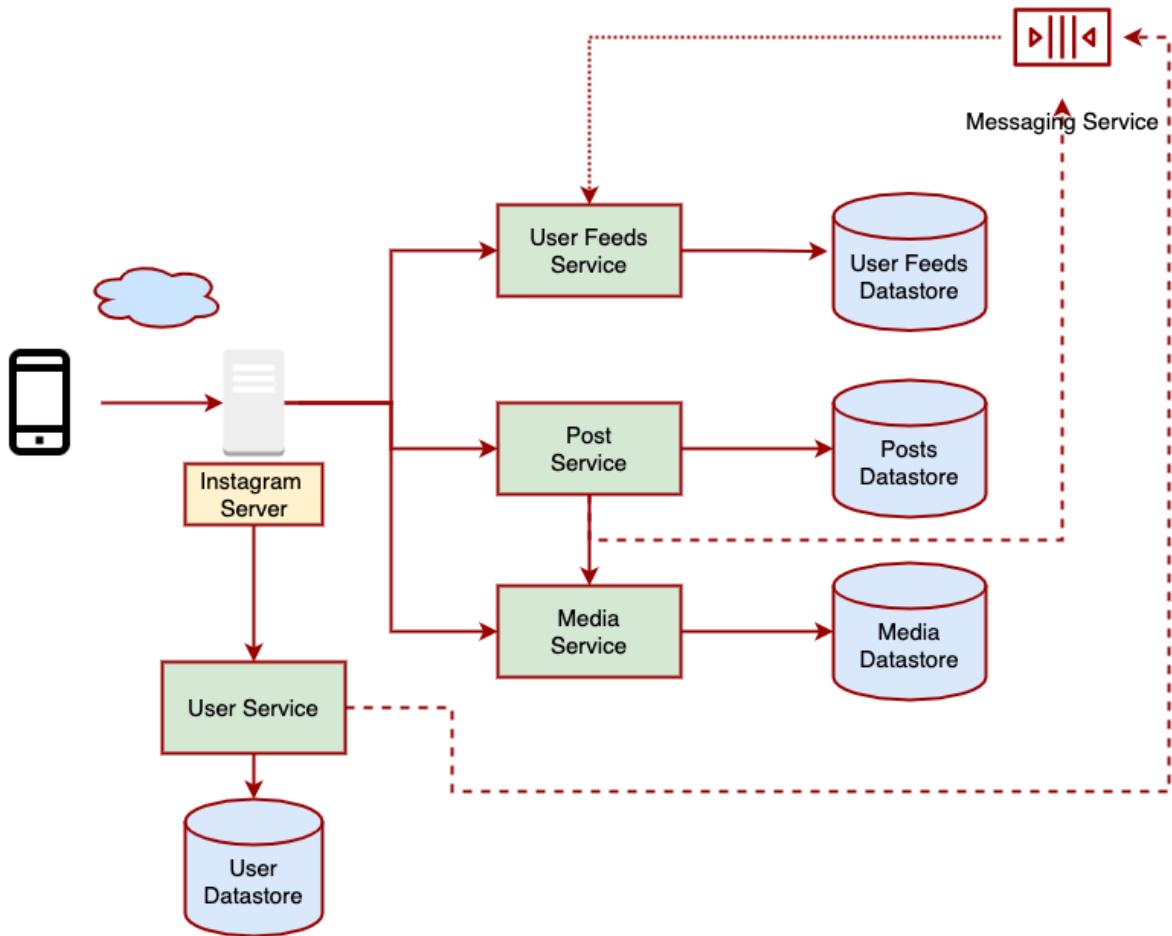
5. Build a High-Level Design

Let's work on the high-level design of our system that meets our functional requirements (use-cases) and then we can review the design and make sure it can meet our non-functional requirements (assumptions and constraints).

At this point focus on the key component and services that you need to build your solution meeting all the agreed-upon requirements. Draw an outline of how these components talk to each other use-case by use-case. Include the data design, API design and any special algorithm that may be needed as part of your services and components.

Architecture

The overall high-level architecture of our platform is as follows:



The platform exposes a set of services and each service performs a dedicated function. Each of these services exposes a set of APIs to allow interactions from the client app (i.e. iPhone/Android-based app the user uses) or from other services to support all the use cases in scope. Further, each of these services uses a dedicated data store to store the data belonging to the services. When a service receives a request e.g. new post creation, following a user, etc. at a high level it does perform the following two operations:

- a. It validates the request and persists (inserts or updates) the data in its data store.
- b. Optionally, the service sends a message to the messaging service in the form of an event that other services can consume and update their state.

The messaging service provides decoupling between various services leading an event-based architecture and will help us to scale the platform to support a large user base of 500m+.

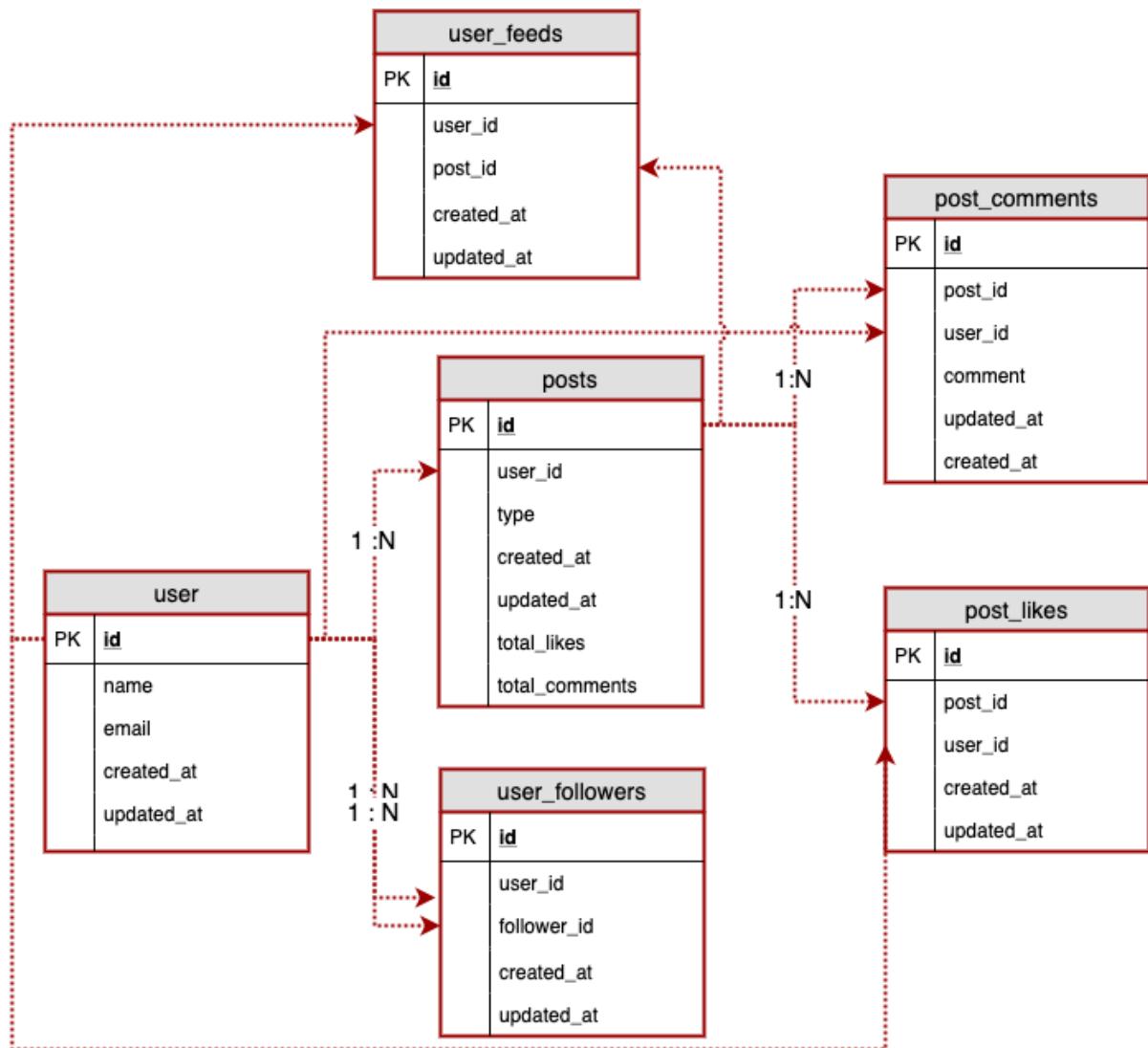
Datastore

- For the data store, both SQL and NoSQL solutions can be considered. But given the scalability challenges surrounding the SQL solutions and the large number of the user base to support we will choose to go with NoSQL solution for the data store.

- To store the entities like the user, followers, etc. we can choose a graph-based data store solution like Neo4J. Each instance of the entity will become a node and each node will have various child nodes.
- For other entities like posts, comments, likes, etc we can use a wide column data store like Cassandra. Each instance of the entity can have a large number of related attributes that can be easily supported as columns in wide column datastore.
- For storing the actual contents of the photos, videos we can use general storage solutions like Amazon S3 or any other kind of flat file or large object-store.

Data Model

Let's model our data layer to store the metadata about the users and their photos and videos and engagement.



Components

The following are the key system components for our platform:

User Service: Provides API to manage users and interactions where a user follows or unfollows another user. A separate User Follower Service can be defined instead of making the follow/unfollow use case part of the User Service itself.

Posts Service: Provides API to allows various posts (photo, video, etc.) to be created and stored in the data store. Also, it handles the ability for a user to add comments and like of posts as well. The Post Service can further be broken down into Post Comments and Post Like Services if required.

Media Service: Provides API to allows storage and retrieval of photos and videos from its underlying data store.

Messaging Service: The messaging service is a bus or a steaming event service. The various services like User Service, Posts Service, etc. are produces and post events like user liked a photo, user followed another user, etc. to relevant topics. On the other hand, various consumer services like User News Feed subscribe to these topics and take action based on the ents published by other services.

Each of these services can be built as microservices performing dedicated tasks talking other services directly (e.g. Post Service taking to Media Service for storage) or services talking to each other via event exchanges via the Messaging Server (e.g. User Feeds Service listening to event posted by the Posts Service)

6. Use Case Design

The following section presents a high-level design for each use case in scope:

Design: Upload Photos and Videos

API

We can create two API (can be REST-based) as follows:

Upload Photo - Photoid uploadPhoto(UserID, Base 64 Encoded Photo) : /photo/upload

Upload Video - Videoid uploadVideo(UserID, Base64 Video Payload) : /video/upload

Each API returns a unique post id associated with the upload. The type attribute stores the type of content (photo/video). The upload services take in the data (photo, video) contents and create an entry in the post datastore. It then talks to the media service and persists the contents of the post into the media data store. Finally, it creates an event (e.g. Post Created) with all the required post data (e.g. postID, userID, etc. and sends an event to the messaging service. The consumer (job) associated with the message service gets notified about the event and its associated data. This job then triggers the pre-computation or update of the user feed for all the followers of the user who just posted the post. This job updates the new feed for followers by adding the post that was just created into their feed list.

Optimization: The job looks at the number of followers that the user has. If the user is very famous (e.g. celebrity, company, etc.), very possibly the user may have millions of followers. In this case, the job skips the pre-computation of new feed for the follower users to avoid an extensively large IO operation. For such users with a large following, it is best to compute the feed of followers on the fly when the feed is accessed by the follower.

Storage

Since the photos, esp. the video sizes are large, instead of storing these objects in a database, we can store them in the same sort of object storage like Amazon S3, etc. for efficient storage and retrieval.

Design: Like a Photo or Video

Data Model

The relationship between the post and its like is 1: N as shown in our data model diagram.

API

We can create two APIs (can be REST-based) as follows:

Like a Photo - Result likePhoto(UserID, Photoid) : /photo/like

Like a Video -Result likeVideo(UserID, Photoid) : /video/like

In our system thus far we have modeled both the photos and videos both as posts so that we can easily display photos and videos in single-stream based on some sorting criteria e.g. time, likes, etc. If we split these into separate entities, we have to merge these from two sources and then sort it as required, this will increase the response time for the API. Each post also maintains a count of likes, so when a post is liked, the Post Service adds an entry against the post to reflect the user who liked it, likewise, it also goes ahead and also increments the like count of that post by 1.

Design: Follow/Un Follow a user

Data Model

The relationship between the user and its followers is 1: N as shown in our data model diagram above.

API

We can create two API (can be REST-based) as follows:

Follow a User - Result followUser(UserID, FollowerID) : /user/follow

Unfollow a User -Result unFollowUser(UserID, FollowerID) : /user/unfollow

The Follow API simply adds a FollowerID into the list of followers of the UserID. The Unfollow API does the reverse, it simply removes the FollowerID entry from the list of followers of the UserID.

Design: User News Feed

Data Model

The relationship between the user and its feed post is 1: N as shown in our data model diagram above.

API

We can create a single API (can be REST-based) as follows:

User New Feed - Post[] newsFeed(UserID) : /user/feed

The User News Feed is pre-computed and stored in the user feeds datastore. Depending on the number of posts from the users a particular user, the size of feed varies a lot. But since we have an overall huge user base it is most likely a user is following a good number of users (e.g. in average case it could be 200). The approach precomputes the user news feed once and simply update it with the latest at all time. As such the job the User News Feed API simply is to return a stream of posts in reverse chronological order to the client. This API will also support the pagination of the news feed items e.g. the API can return 50 posts in one request and then the client can ask for the next 50 posts and so on. Further, as we scale the design, we can cache the output of this API to increase the performance and load of the system.

Security + Optimization: The news feed contains a list of posts and location of the post contents (.e.g. the actual photo or video stored in S3.) This poses a security risk if somehow a user can determine the URL of that contents, the link can be shared and anyone can view the content as it would be publically accessible. We can mitigate that by building an additional service that checks and applies access controls before serving the photos and videos. Also, since our system is 80% read, its best to have a separate cluster of nodes that focus on serving the content while having a separate cluster of nodes supports the uploads thus increasing the overall system scalability.

7. Scale the Design

In the following section, we will evaluate the various optimization techniques that we can use to scale our design for performance, scalability, redundancy, etc. to meet our non-functional requirements.

Data Sharding

As the number of users and posts is huge, it's possible to store all the data from a data store on to a single machine, is best to start to define a partitioning strategy. The partition strategy helps to divide the data stored for a given service into many partitions. Each of the partition

is replicated to another host in the cluster. This technique of distributing data based on given criteria is called as data sharding. Data sharding helps to improve the performance of a data node as it has to search a document within a smaller subset of nodes.

Let's look at some of the possible ways in which we can partition our data.

By User

In this case, all the data (posts) for a given user will be stored in a single shard. We cannot have unlimited shards nor there can be 1 shard per user since we have a huge number of users. The approach would define a reasonable number of shards keeping the number of users, size and frequency of content produced by the users, and finally the size of the single shard that we can support. Generally, the average shard size ranges between 20- 40 GB.

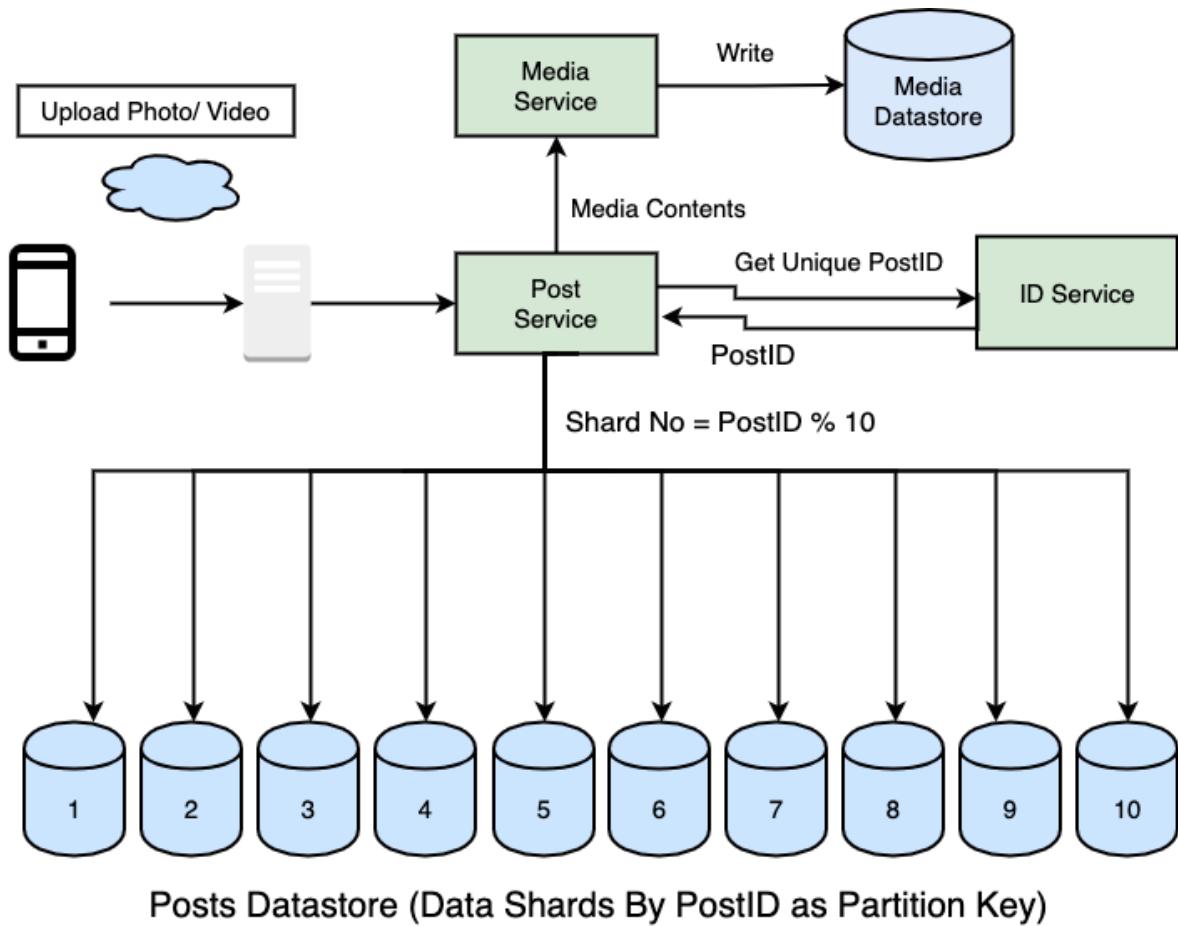
We will use UserID (which is unique for each user) is used as a partition key. Let's say we divide our database into 100 shards, so by doing UserID % 100, we can find in which shard the data for that user will reside. Further, the Post ID can be generated either by using the local counter to shard and appending the shard ID to it.

In this approach, we face the following problems:

- Uneven shard growth size as some users may have more and some have fewer data.
- The shards belonging to users who are popular (have a following) will be used excessively compared to other shards and would become hot.

By Post

In this approach, we partition the data by PostID. Now the PostID has to be unique and must have uniformly distributed so that we can use an effectively use it as a partition key. We cannot use the timestamp or some other logic there off to generate a unique Post ID on each node, rather we have to build some central service that can return a unique PostID across the system that we can use. Then using the same modulus approach of PostID % 100 we can store the post and its metadata on that shard.



This approach solves the problem associated with partitioning by UserId but no matter which every way we partition, sharding will present some of the following issues which we need to understand and deal it.

Sharding Issues:

Complex queries have to merge data from one to another and, sort and merge.

Join across shards are expensive or not possible.

Fixed no. of sharding segments, difficult to add new later.

Caching

Caching is a very essential aspect that would help to scale our system. Caching helps with increasing the performance as the precomputed data stored in the cache (memory) is returned rather than compute the data for each request. But caching can introduce some inconsistency into the system e.g. it may take time for a new user post to be reflected in the user feeds for their followers. Further, the duration of inconsistency can be reduced by reducing the cache time to live or updating the cache realtime based on events rather than waiting for the cache to re-computed on TTL expiration.

The User Feed Service is the ideal component where we can introduce caching and improve the performance of the system. We can use any caching solution like Redis, Memcache, etc. to store the user feed for each user. Further, we can define an optimization strategy based on LRU (Least recently used) method to evict certain user feed for a user who may not be active for a while. Also, If the feed cache will be updated in realtime based on events, we can limit the number of entries in the cache to a certain number, automatically removing the oldest posts. Finally, as the user browses through the feed and loads more posts, we can preemptively fetch the next set of feed data for that user from the store and add it to the cache, reducing the latency and improving the user experience.

Optimization: Beyond that, the photos and videos corresponding to the posts that are much in demand can also be added to the cache at source or at CDN to further speed up the serving of the content.

Load Balancing

We can use DNS load balancing to distribute the load (requests) coming from various geographic locations to evenly spread the request across multiple data centers. Each service will be front-ended by a load balancer to evenly divide the incoming request to various service nodes as determined by the capacity estimation. A number of load balancing strategies like round-robin, least connection, etc. can be used. Further, since all our services are stateless, we can easily add or remove nodes from the cluster without losing any state.

CDN

We can put a CDN in front have all the media (photos, videos) be served from edge servers that are physically located I to user. Using a CDN (Content Delivery Network) greatly reduces the latency and offloads mundane IO work to edge servers helping our solution to scale to millions of concurrent users.

Video Streaming

The videos uploaded by the users can be optimized for various devices and bandwidth. When a video is uploaded it can be converted to various video files specifically supporting key device resolutions and internet bandwidth. A separate service called Video Service will have to be designed to designed which creates all such video artifacts and uploads them to media storage. If a CDN is used, then these video artifacts will have to be distributed to CDN as well. Further, the Media Playback Service would need to be enhanced to detect the client device type and internet speed so as to serve the corresponding optimized video artifact to avoid latency, buffering, and improve the user watching experience.

Redundancy & Disaster Recovery

Each datastore in the cluster can have a replication schema where the data on a given shard of the node is replicated to 1-3 other nodes. In the event, if any node has some hardware/software issues dies and falls of the cluster, the replica nodes take over and operations continue but with reduced capacity. Laster, when a new node is added to replace the failed node, the data from other replica nodes are replicated to this node the capacity

goes back to normal. The same technique can be applied to replicate our photo and video storage system so that we don't lose the critical data. Further, we can replicate the data from a given datastore residing in a data center to another data store located in another datacenter. Doing so, even if one data center goes down, the operation can be moved or taken over by another data center.

Solution 4:

Problem Statement

Design a photo-sharing platform similar to Instagram where users can upload their photos and share it with their followers. Subsequently, the users will be able to view personalized feeds containing posts from all the other users that they follow.

Gathering Requirements

In Scope

The application should be able to support the following requirements.

- Users should be able to upload photos and view the photos they have uploaded.
- Users should be able to follow other users.
- Users can view feeds containing posts from the users they follow.
- Users should be able to like and comment the posts.

Out of Scope

- Sending and receiving messages from other users.
- Generating machine learning based personalized recommendations to discover new people, photos, videos, and stories relevant one's interest.

High Level Design

Architecture

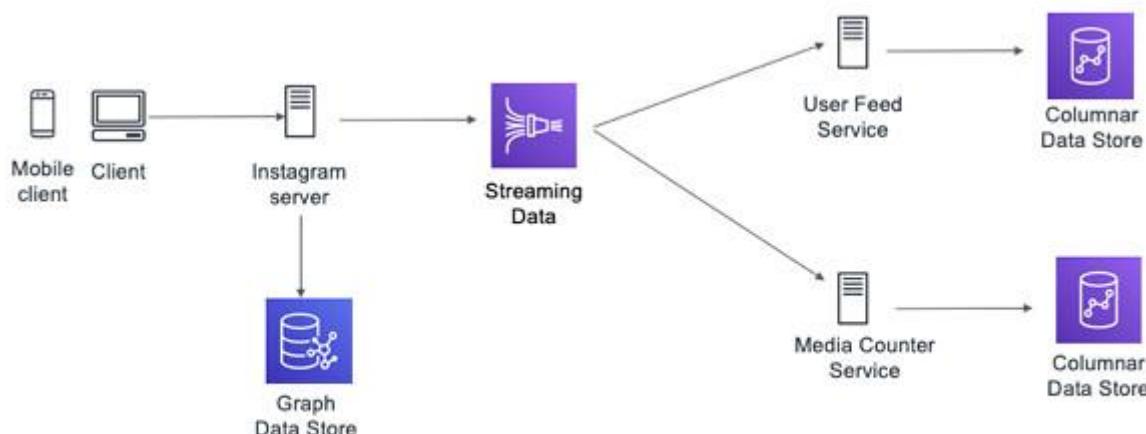


Fig 0: Architecture of Photo Sharing application

When the server receives a request for an action (post, like etc.) from a client it performs two parallel operations: i) persisting the action in the data store ii) publish the action in a streaming data store for a pub-sub model. After that, the various services (e.g. User Feed Service, Media Counter Service) read the actions from the streaming data store and performs their specific tasks. The streaming data store makes the system extensible to support other use-cases (e.g. media search index, locations search index, and so forth) in future.

FUN FACT: In this [talk](#), Rodrigo Schmidt, director of engineering at Instagram talks about the different challenges they have faced in scaling the data infrastructure at Instagram.

System Components

The system will comprise of several micro-services each performing a separate task. We will use a graph database such as Neo4j to store the information. The reason we have chosen a graph data-model is that our data will contain complex relationships between data entities such as users, posts, and comments as nodes of the graph. After that, we will use edges of the graph to store relationships such as follows, likes, comments, and so forth. Additionally, we can use columnar databases like Cassandra to store information like user feeds, activities, and counters.

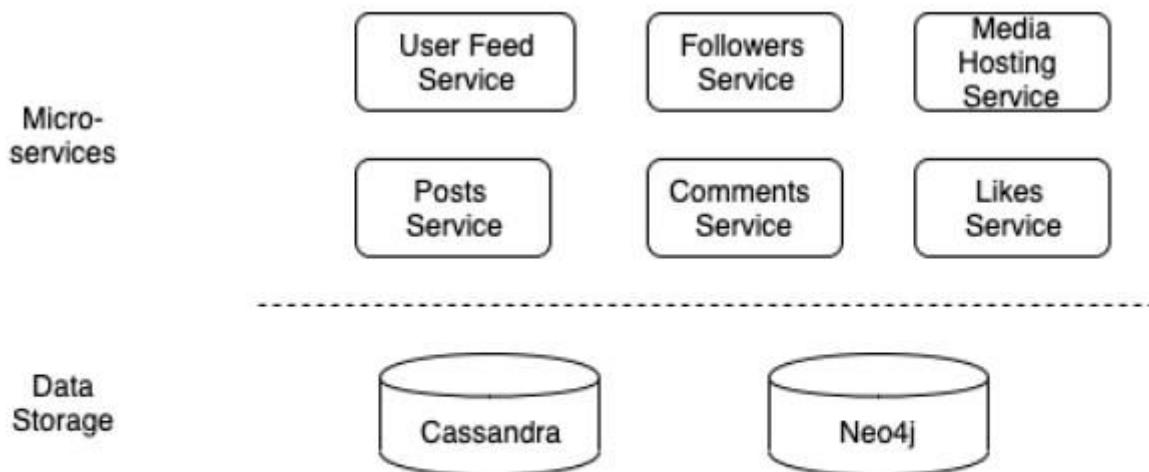


Fig 1: High Level System Components

Component Design

Posting on Instagram

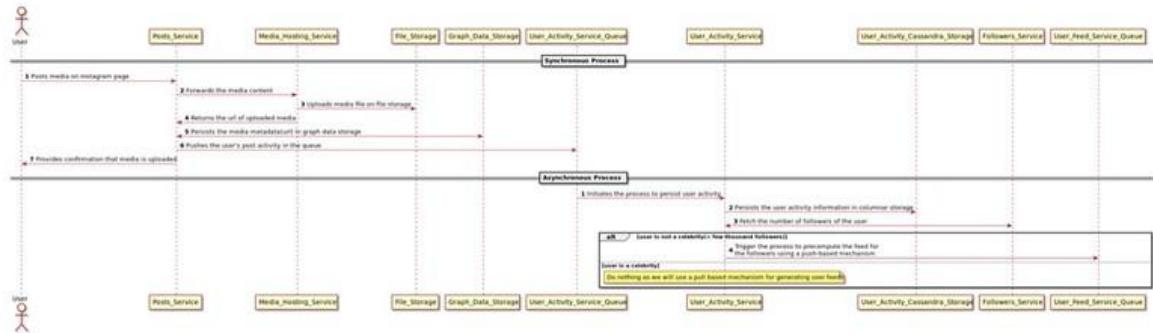


Fig 2: Synchronous and Asynchronous process for posting on Instagram

There are two major processes which gets executed when a user posts a photo on Instagram. Firstly, the synchronous process which is responsible for uploading image content on file storage, persisting the media metadata in graph data-storage, returning the confirmation message to the user and triggering the process to update the user activity. The second process occurs asynchronously by persisting user activity in a columnar data-storage(Cassandra) and triggering the process to pre-compute the feed of followers of non-celebrity users (having few thousand followers). We don't pre-compute feeds for celebrity users (have 1M+ followers) as the process to fan-out the feeds to all the followers will be extremely compute and I/O intensive.

API Design

We have provided the API design of posting an image on Instagram below. We will send the file and data over in one request using the multipart/form-data content type. The [MultiPart/Form-Data](#) contains a series of parts. Each part is expected to contain a content-disposition header [RFC 2183] where the disposition type is "form-data".

URL:

POST /users/<user_id>/posts

Sample Request Body:

Content-Type: ***multipart/form-data***; boundary=ExampleFormBoundary

--ExampleFormBoundary

Content-Disposition: form-data; name="file"; filename="test.png"

Content-Type: image/png

{image file data bytes}

--ExampleFormBoundary--

Sample Response:

{

```

    > "type": "ImageFile",
      "id": "667",
      "createdAt": "1466623229",
      "createdBy": "9",
      "name": "test.png",
      "updatedAt": "1466623230",
      "updatedBy": "9",
      "fullImageUrl": "https://mybucket.s3.amazonaws.com/myfolder/test.jpg"
    },
    "size": {
      "type": "Size",
      "width": "316",
      "height": "316"
    }
  }
}

```

Precompute Feeds

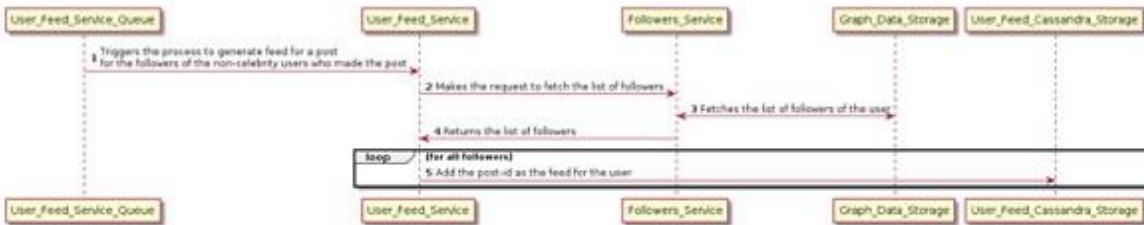


Fig 3: Pre-compute feeds from non-celebrity users

This process gets executed when non-celebrity users makes a post on Instagram. It's triggered when a message is added in the User Feed Service Queue. Once the message is added in the queue, the User Feed Service makes a call to the Followers Service to fetch the list of followers of the user. After that, the post gets added to the feed of all the followers in the columnar data storage.

Fetching User Feed

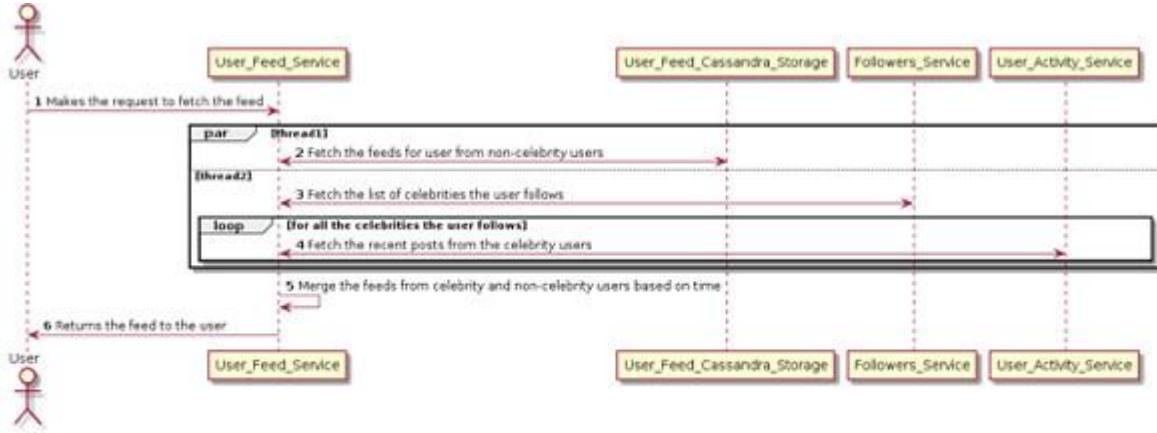


Fig 4: Sequence of operations involved in fetching user feed

When a user requests for feed then there will be two parallel threads involved in fetching the user feeds to optimize for latency. The first thread will fetch the feeds from non-celebrity users which the user follow. These feeds are populated by the fan-out mechanism described in the PreCompute Feeds section above. The second thread is responsible for fetching the feeds of celebrity users whom the user follow. After that, the User Feed Service will merge the feeds from celebrity and non-celebrity users and return the merged feeds to the user who requested the feed.

API Design

URL:

GET /users/<user_id>/feeds

Sample Response:

The response below can be mapped directly to the graph data model mentioned in the next section.

```
{
  "feeds": [
    {
      "postId": "post001",
      "postOwnerId": "user001",
      "postOwnerName": "Bill Gates",
      "mediaURL": "https://mybucket.s3.amazonaws.com/myfolder/test.jpg",
      "numberOfLikes": 400000
      "numberOfComments": 19789
      "comments": [
        {
          "commentId": "comment001",
          "commentText": "Great post!",
          "commenterId": "user002",
          "commenterName": "John Doe",
          "commentTime": "2023-01-01T12:00:00Z"
        }
      ]
    }
  ]
}
```

```

        "commenterUserId": "user004",
        "commenterName": "JeffBezos"
        "comment": "Amazing!"
        "likes": [
            {
                "likerId": "user003",
                "likerUserName": "Bob"
            }
        ]
    }
]
}

```

Data Models

Graph Data Models

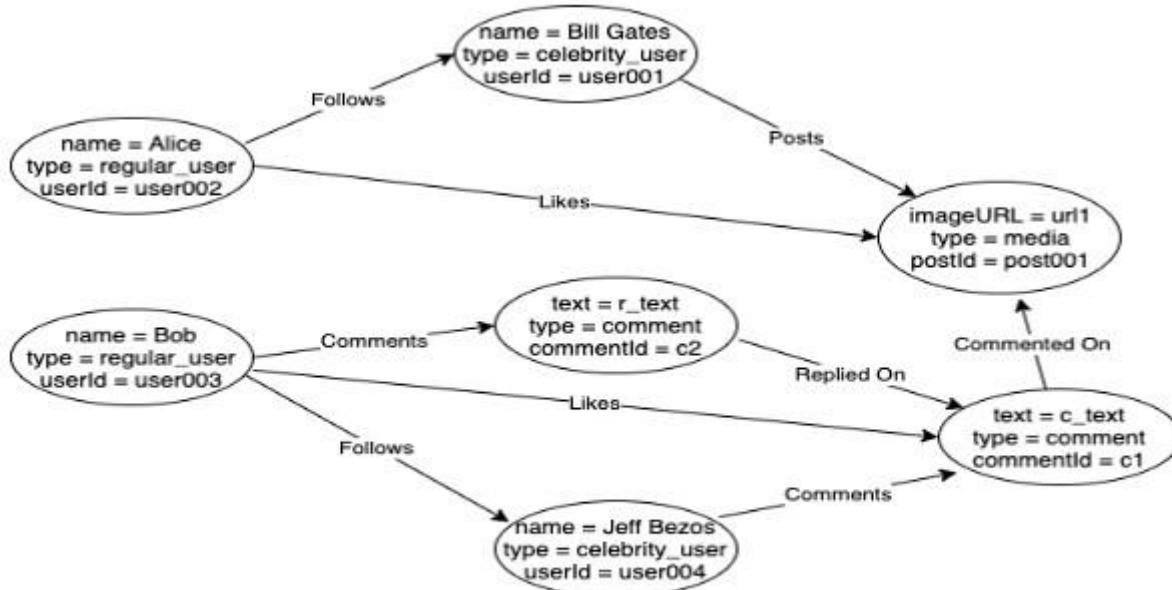


Fig 5: Graph representation of users, posts, and comments

We can use a graph database such as Neo4j which stores data-entities such as user information, posts, comments, and so forth as nodes in the graph. The edges between the nodes are used to store the relationship between data entities such as followers, posts, comments, likes, and replies. All the nodes are added to an index called nodeIndex for faster lookups. We have chosen this NoSQL based solution over relational databases as it

provides the scalability to have hierarchies which go beyond two levels and extensibility due to the schema-less behavior of NoSQL data storage.

Sample Queries supported by Graph Database

Fetch all the followers of Jeff Bezos

```
Node jeffBezos = nodeIndex.get("userId", "user004");
List jeffBezosFollowers = new ArrayList();

for (Relationship relationship: jeffBezos.getRelationships(INGOING, FOLLOWS)) {
    jeffBezosFollowers.add(relationship.getStartNode());
}
```

Fetch all the posts of Bill Gates

```
Node billGates = nodeIndex.get("userId", "user001");
List billGatesPosts = new ArrayList();

for (Relationship relationship: billGates.getRelationships(OUTGOING, POSTS)) {
    billGatesPosts.add(relationship.getEndNode());
}
```

Fetch all the posts of Bill Gates on which Jeff Bezos has commented

```
List commentsOnBillGatesPosts = new ArrayList<>();

for(Node billGatesPost : billGatesPosts) {
    for (Relationship relationship: billGatesPost.getRelationships(INGOING, COMMENTED_ON))
    {
        commentsOnBillGatesPosts.add(relationship.getStartNode());
    }
}

List jeffBezosComments = new ArrayList();

for (Relationship relationship: jeffBezos.getRelationships(OUTGOING, COMMENTS)) {
    jeffBezosComments.add(relationship.getEndNode());
}

List jeffBezosCommentsOnBillGatesPosts =
commentsOnBillGatesPosts.intersect(jeffBezosComments);
```

Columnar Data Models

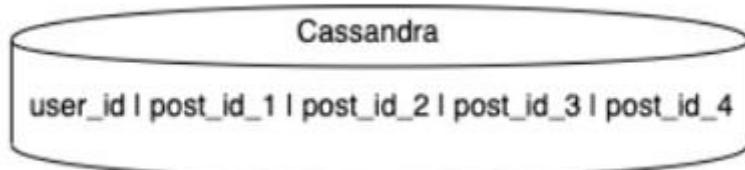


Fig 6: Columnar Data Model for user feed and activities

We will use columnar data storage such as Cassandra to store data entities like user feed and activities. Each row will contain feed/activity information of the user. We can also have a TTL based functionality to evict older posts. The data model will look something similar to:

User_id -> List

FUN FACT: In this [talk](#), Dikang Gu, a software engineer at Instagram core infra team has mentioned about how they use Cassandra to serve critical usecases, high scalability requirements, and some pain points.

Streaming Data Model

We can use cloud technologies such as [Amazon Kinesis](#) or [Azure Stream Analytics](#) for collecting, processing, and analyzing real-time, streaming data to get timely insights and react quickly to new information(e.g. a new like, comment, etc.). We have listed below the de-normalized form of some major streaming data entities and action.

<pre>A. class User { String id; String userName; String fullName; boolean isPrivate; ... }</pre>	<pre>B. class Post { String id; String ownerId; MediaContentType contentType; ... }</pre>
<pre>C. class LikeEvent { String likerId; String postId; String postOwnerId; Post post; User liker; User postOwner; boolean isFollowingMediaOwner ... }</pre>	<pre>D. class Follow { User follower; User followedUser; boolean isFollowedUserCelebrity; ... }</pre>

Table: De-normalized major data-entities and actions

The data entities **A** and **B** above show the containers which contain denormalized information about the Users and their Posts. Subsequently, the data entities **C** and **D** denote the different actions which users may take. The entity **C** denotes the event where a user likes a post and entity **D** denotes the action when a user follows another user. These actions are read by the related micro-services from the stream and processed accordingly. For

instance, the *LikeEvent* can be read by the *Media Counter Service* and is used to update the media count in the data storage.

Optimization

We will use a cache having an LRU based eviction policy for caching user feeds of active users. This will not only reduce the overall latency in displaying the user-feeds to users but will also prevent re-computation of user-feeds.

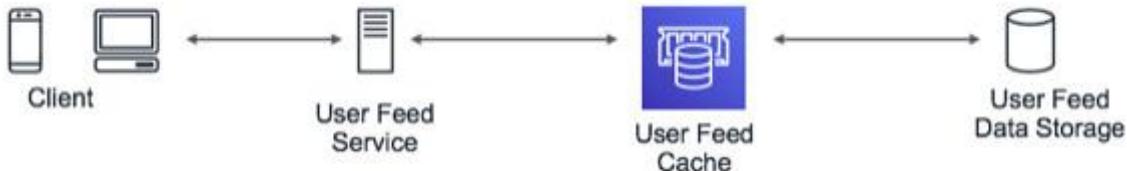


Fig 7: LRU based cache for storing user feeds of active users

Another scope of optimization lies in providing the best content in the user feeds. We can do this by ranking the new feeds (the ones generated after users last login) from those who the user follows. We can apply machine learning techniques to rank the user feeds by assigning scores to the individual feeds which would indicate the probability of click, like, comment and so forth. We can do this by representing each feed by a feature vector which contains information about the user, the feed and the interactions which the user has had with the people in the feed (e.g. whether the user had clicked/liked/commented on the previous feeds by the people in the story). It's apparent that the most important features for feed ranking will be related to social network. Some of the keys of understanding the user network are listed below.

- Who is the user a close follower of? For example, one user is a close follower of Elon Musk while another user can be a close follower of Gordon Ramsay.
- Whose photos the user always like?
- Whose links are most interesting to the user?

We can use [deep neural networks](#) which would take the several features (> 100K dense features) which we require for training the model. Those features will be passed through the n-fold layers, and will be used for predicting the probability of the different events (likes, comments, shares, etc.).

FUN FACT: In this [talk](#), Lars Backstrom, VP of Engineering @ Facebook talks about the machine learning done to create personalized news feeds for users. He talks about the classical machine learning approach they used in the initial phases for personalizing News Feeds by using decision trees and logistic regression. He then goes to talk about the improvements they have observed in using neural networks.

9. Designing Uber backend

Design a service where a user requests a ride from the app, and a driver arrives to take them to their destination. A frequently asked interview question in the system design round of interviews.

Architecture: Monolithic/Microservices (real-time service, Front-end (Application), and database)

Things to analyze and discuss:

- The backend is primarily serving mobile phone traffic. uber app talks to the backend over mobile data.
- How dispatch system works (GPS/ location data is what drives the dispatch system)? How efficiently can the user match request with the nearby drivers?
- How do maps and routing work in Uber? How ETAs are calculated?
- An efficient approach to store millions of geographical locations for drivers/riders who are always on the move.
- Approach to handle millions of updates to driver location.
- Dispatch is mostly built using [Node.js](#)
- Services: Business logic services are mostly written in python.
- Databases: Postgres, Redis, MySQL.

Solution 1:

Uber backend: problem overview

Uber aims to make transportation easy and reliable. The popular smartphone app handles high traffic and complex data and systems. Uber enables customers to book affordable rides with drivers in their personal cars. The customers and drivers communicate using the Uber app.

There are two kinds of users that our system should account for: **Drivers** and **Customers**.

What we know about our system requirements is:

- Drivers must be able to frequently notify the service regarding their current location and availability
- Passengers should be able to see all nearby drivers in real-time
- Customers can request a ride using a destination and pickup time.
- Nearby drivers should be notified when a customer needs to be picked up.
- Once a ride is accepted, both the driver and customer must see the other's current location for the entire duration of the trip.
- Once the drive is complete, the driver completes the ride and should then be available for another customer.

Similar services: Lyft, Didi, Via, Sidecar, etc.

Difficulty level: Hard

Helpful prerequisite: [Designing Yelp](#)

Constraints

Constraints (i.e. capacity estimations and system limitations) are some of the most important considerations to make before designing Uber's backend system. Constraints will generally differ depending on time of day and location.

We'll design our build with the following **constraints** and **estimations**:

- 300 million customers and one million drivers in the system
- One million active customers and 500 thousand active drivers per day
- One million rides per day
- All active drivers notify their current location every three seconds
- System contacts drivers in real time when customer puts in a ride request

Uber backend: System design

For our Uber solution, we will be referencing our answer to another popular system design interview question: [Designing Yelp](#). Please take a minute to check our [Yelp solution](#) if you're not familiar with it.

We would need to make modifications to align with our Uber system and its requirements. For instance, our QuadTree must be adapted for **frequent updates**.

A few issues arise if we use the Dynamic Grid solution from our Yelp problem:

- We need to update data structures to reflect active drivers' reported locations every three seconds. To update a driver to a new location, we must find the right grid based on the driver's previous location.
- If the new position doesn't belong to the current grid, we remove the driver from the current grid and reinsert them to the right grid. If the new grid reaches a maximum limit, we have to repartition it.
- We need a quick mechanism to propagate the current location of nearby drivers to customers in the area. Our system needs to notify both the driver and customer on the car's location throughout the ride's duration.

In these cases, a **QuadTree is not ideal** because we can't guarantee the tree will be updated as quickly as our system requires. The QuadTree must be updated with every driver's update so that the system only uses fresh data reflecting everyone's current location.

We could keep the most recent driver position in a hash table and update our QuadTree less frequently. We want to guarantee that a driver's current location is reflected in the QuadTree **within 15 seconds**. We maintain a hash table that will store the current driver location. We can call it DriverLocationHT.

DriverLocationHT

We need to store DriverID in the hash table, which reflects a driver's current and previous location. This means that we'll need 35 bytes to store each record:

1. DriverID (3 bytes for 1 million drivers)
2. Old latitude (8 bytes)
3. Old longitude (8 bytes)
4. New latitude (8 bytes)
5. New longitude (8 bytes)

This totals to 35 bytes.

Since we assume one million drivers, we'll require the following memory:

$$1 \text{ million} * 35 \text{ bytes} \Rightarrow 35 \text{ MB}$$

Now let's discuss **bandwidth**. If we get the DriverID and location, it will require (3+16=>19 bytes)(3+16 => 19 bytes)(3+16=>19 bytes). This information is received every three seconds from 500 thousand daily active drivers, so we receive 9.5MB every three seconds.

To randomize distribution, we could distribute DriverLocationHT on multiple servers based on the DriverID. This will help with scalability, performance, and fault tolerance. We will refer to the machines holding this information as the *Driver Location servers*.

These servers will do two more things. Once the server receives an update on a driver's location, it will broadcast that information to relevant customers. The server will also notify the respective QuadTree server to refresh the driver's location.

Broadcasting driver locations

We'll need to broadcast driver locations to our customers. We can use a Push Model so that the server pushes positions to relevant users. We can use a Notification Service and build it on the **publisher/subscriber model**.

When a customer opens the Uber app, they'll query the server to find nearby drivers. On the server side, we subscribe the customer to all updates from nearby drivers. Each update in a driver's location in DriverLocationHT will be broadcast to all subscribed customers. This ensures that each driver's current location is displayed.

We've assumed one million active customers and 500 thousand active drivers per day. Let's assume that five customers subscribe to one driver. We'll store this information in a **hash table for quick updates**.

We need to store both driver and customer IDs. We need three bytes for DriverID and eight bytes for CustomerID, so we will need 21MB of memory.

$$(500,000 * 3) + (500,000 * 5 * 8) = 21 \text{ MB}$$
$$(500,000 * 3) + (500,000 * 5 * 8) = 21 \text{ MB}$$

Now for bandwidth. For every active driver, we have five subscribers. In total, this reaches:

$$5 * 500,000 \Rightarrow 2.5\text{million}$$

We need to send DriverID (3 bytes) and their location (16 bytes) every second, which requires:

$$2.5\text{ million} * 19\text{ bytes} \Rightarrow 47.5\text{MB/s}$$

$$2.5\text{million} * 19\text{bytes} \Rightarrow 47.5\text{MB/s}$$

Notification service

To efficiently implement the Notification service, we can either use HTTP long polling or push notifications. Customers are subscribed to nearby drivers when they open the Uber app for the first time.

When new drivers enter their areas, we need to add a new customer/driver subscription dynamically. To do so, we track the area that a customer is watching. However, this would get extra complicated.

Instead of pushing this information, we can design the system so **customers pull the information from the server**. Customers will send their current location so that the server can find nearby drivers from our QuadTree. The customer can then update their screen to reflect drivers' current positions.

Customers should be able to query every five seconds. This will limit the number of round trips to the server.

To **repartition**, we can create a cushion so that each grid grows beyond the limit before we decide to partition it. Assume our grids can grow or shrink by an *extra 10%* before we partition them. This will decrease the load for a grid partition.

Use case: request ride

We'll summarize how this use case works below.

- The customer enters a ride request.
- One of the Aggregator servers accepts the request and asks the QuadTree servers to return nearby drivers.
- The Aggregator server collects the results and sorts them by ratings.
- The Aggregator server sends a notification to the top drivers simultaneously.
- The first driver to accept will be assigned that ride. The other drivers will receive a cancellation.
- If the drivers do not respond, the Aggregator will request a ride from the next drivers on our list.
- The customer is notified once a driver accepts a request.

Advanced issues and considerations

Fault tolerance and replication

We will need **server replicas** in case the Driver Location or Notification servers die. A secondary server can take control when a primary server dies. We can also store data in persistent storage like solid state drives (SSDs) to provide fast input and output. We can quickly use this persistent storage to recover data in the event that both primary and secondary servers die.

Ranking

Uber also provides a ranking system for drivers. A customer can rate a driver according to wait times, courtesy, and safety. Let's say we want to **rank search results by popularity** or relevance as well as proximity.

To do this, we need to return top-rated drivers within a given radius. Assume we track drivers' ratings in a database and QuadTree. An aggregated number will represent popularity in the system based on these ratings. The system will search for the top 10 drivers in a given radius, while we ask each partition of the QuadTree to return the top drivers with a specified rating. The aggregator server will determine the top 10 drivers among all drivers returned by different partitions.

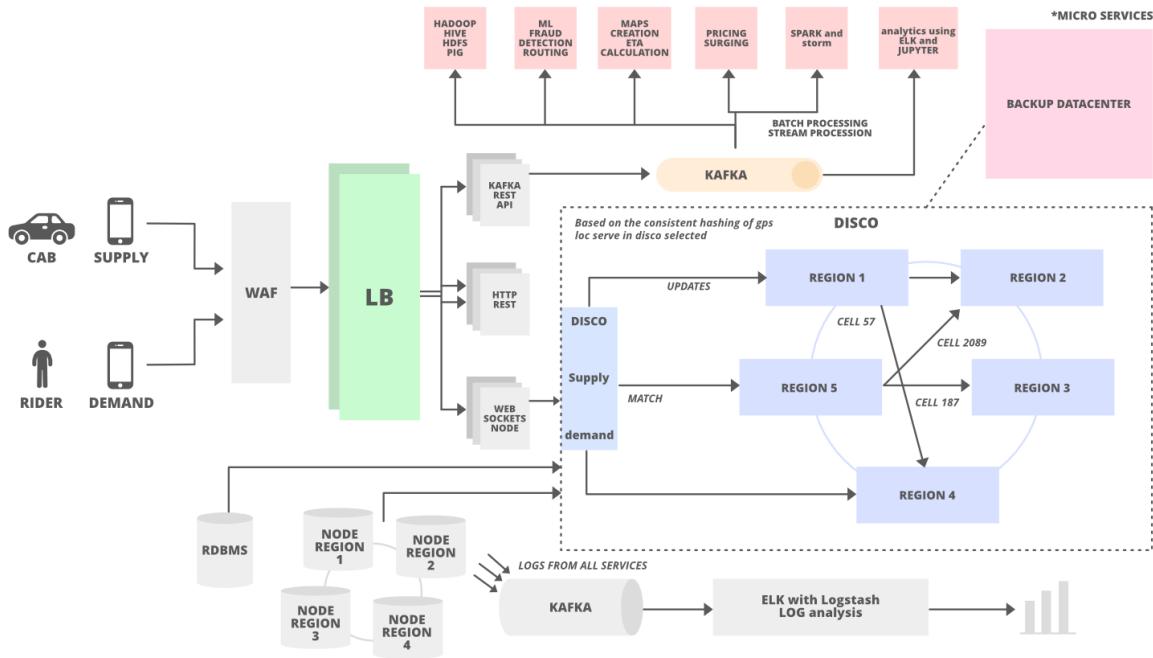
Other issues to consider

- Clients using slow or disconnecting networks
- Clients that are disconnected during the duration of a ride
- How to handle billing if a ride is disconnected
- How to [implement machine learning components in your system](#)

Solution 2:

Uber System Architecture

We all are familiar with Uber services. A user can request a ride through the application and within a few minutes, a driver arrives nearby his/her location to take them to their destination. Earlier Uber was built on the "**monolithic**" software architecture model. They had a backend service, a frontend service, and a single database. They used Python and its frameworks and SQLAlchemy as the ORM layer to the database. This architecture was fine for a small number of trips in a few cities but when the service started expanding in other cities Uber team started facing the issue with the application. After the year 2014 Uber team decided to switch to the "**service-oriented architecture**" and now Uber also handles food delivery and cargo.



1. Talk About the Challenges

One of the main tasks in Uber service is to match the rider with cabs which means we need two different services in our architecture i.e.

- Supply Service (for cabs)
- Demand Service (for riders)

Uber has a **Dispatch system** (Dispatch optimization/DISCO) in its architecture to match supply with demands. This dispatch system uses mobile phones and it takes the responsibility to match the drivers with riders (supply to demand).

2. How Dispatch System Work?

DISCO must have these goals...

- Reduce extra driving.
- Minimum waiting time
- Minimum overall ETA

The dispatch system completely works on maps and location data/GPS, so the first thing which is important is to model our maps and location data.

- Earth has a spherical shape so it's difficult to do summarization and approximation by using latitude and longitude. To solve this problem Uber uses the **Google S2 library**. This library divides the map data into tiny cells (for example 3km) and gives the unique ID to each cell. This is an easy way to spread data in the distributed system and store it easily.
- S2 library gives coverage for any given shape easily. Suppose you want to figure out all the supplies available within a 3km radius of a city. Using the S2 libraries you can

draw a circle of 3km radius and it will filter out all the cells with IDs that lie in that particular circle. This way you can easily match the rider to the driver and you can easily find out the number of cars(supply) available in a particular region.

3. Supply Service And How it Works?

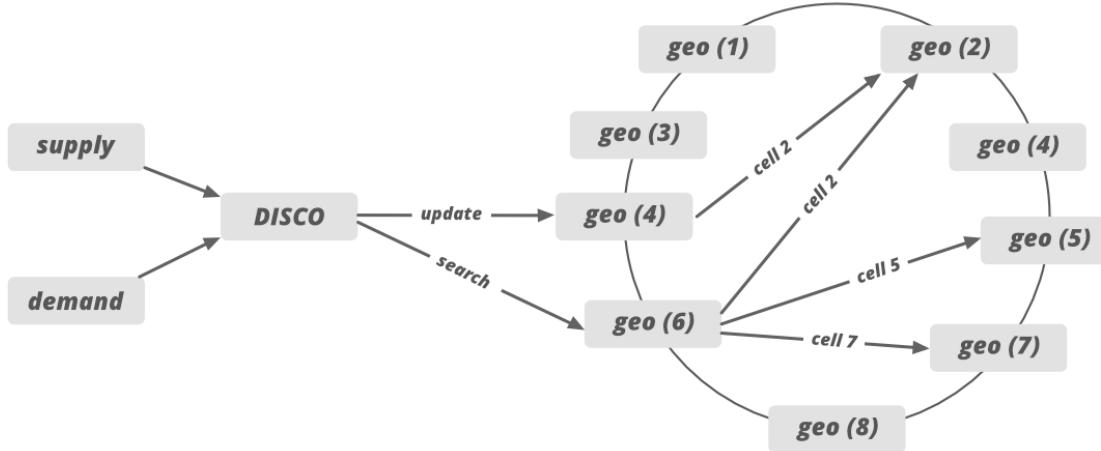
- In our case cabs are the supply services and they will be tracked by geolocation (latitude and longitude). All the active cabs keep on sending the location to the server once every 4 seconds through a web application firewall and load balancer. The accurate GPS location is sent to the data center through Kafka's Rest APIs once it passes through the load balancer. Here we use **Apache Kafka** as the data hub.
- Once the latest location is updated by Kafka it slowly passes through the respective worker nodes' main memory.
- Also, a copy of the location (state machine/latest location of cabs) will be sent to the database and to the dispatch optimization to keep the latest location updated.
- We also need to track a few more things such as the number of seats, presence of a car seat for children, type of vehicle, can a wheelchair be fit, and allocation (for example, a cab may have four seats but two of those are occupied.)

4. Demand Service And How it Works?

- Demand service receives the request of the cab through a web socket and it tracks the GPS location of the user. It also receives different kinds of requirements such as the number of seats, type of car, or pool car.
- Demand gives the location (cell ID) and user requirement to supply and make requests for the cabs.

5. How Dispatch System Match the Riders to Drivers?

- We have discussed that DISCO divides the map into tiny cells with a unique ID. This ID is used as a sharding key in DISCO. When supply receives the request from demand the location gets updated using the cell ID as a shard key. These tiny cells' responsibilities will be divided into different servers lies in multiple regions (consistent hashing). For example, we can allocate the responsibility of 12 tiny cells to 6 different servers (2 cells for each server) lying in 6 different regions.



- Supply sends the request to the specific server based on the GPS location data. After that, the system draws the circle and filters out all the nearby cabs which meet the rider's requirement.
- After that, the list of the cab is sent to the ETA to calculate the distance between the rider and the cab, not geographically but by the road system.
- The sorted ETA is then sent back to the supply system to offer to a driver.

If we need to handle the traffic for the newly added city then we can increase the number of servers and allocate the responsibilities of newly added cities' cell IDs to these servers.

6. How To Scale Dispatch System?

- The dispatch system (including supply, demand, and web socket) is built on **NodeJS**. NodeJS is the asynchronous and event-based framework that allows you to send and receive messages through WebSockets whenever you want.
- Uber uses an open-source **ringpop** to make the application cooperative and scalable for heavy traffic. Ring pop has mainly three parts and it performs the below operation to scale the dispatch system.
 1. It maintains the **consistent hashing** to assign the work across the workers. It helps in sharding the application in a way that's scalable and fault-tolerant.
 2. Ringpop uses **RPC (Remote Procedure Call) protocol** to make calls from one server to another server.
 3. Ringpop also uses a **SWIM membership protocol/gossip protocol** that allows independent workers to discover each other's responsibilities. This way each server/node knows the responsibility and the work of other nodes.
 4. Ringpop detects the newly added nodes to the cluster and the node which is removed from the cluster. It distributes the loads evenly when a node is added or removed.

7. How does Uber Defines a Map Region?

Before launching a new operation in a new area, Uber onboarded the new region to the map technology stack. In this map region, we define various subregions labeled with grades A, B, AB, and C.

Grade A: This subregion is responsible to cover the urban centers and commute areas. Around 90% of Uber traffic gets covered in this subregion, so it's important to build the highest quality map for subregion A.

Grade B: This subregion covers the rural and suburban areas which are less populated and less traveled by Uber customers.

Grade AB: A union of grade A and B subregions.

Grade C: Covers the set of highway corridors connecting various Uber Territories.

8. How does Uber Builds the Map?

Uber uses a third-party map service provider to build the map in their application. Earlier Uber was using [Mapbox](#) services but later Uber switched to Google Maps API to track the location and calculate ETAs.

1. Trace coverage: Trace coverage spot the missing road segments or incorrect road geometry. Trace coverage calculation is based on two inputs: map data under testing and historic GPS traces of all Uber rides taken over a certain period of time. It covers those GPS traces onto the map, comparing and matching them with road segments. If we find missing road segments (no road is shown) on GPS traces then we take some steps to fix the deficiency.

2. Preferred access (pick-up) point accuracy: We get the pickup point in our application when we book the cab in Uber. Pick-up points are a really important metric in Uber, especially for large venues such as airports, college campuses, stadiums, factories, or companies. We calculate the distance between the actual location and all the pickup and drop-off points used by drivers.

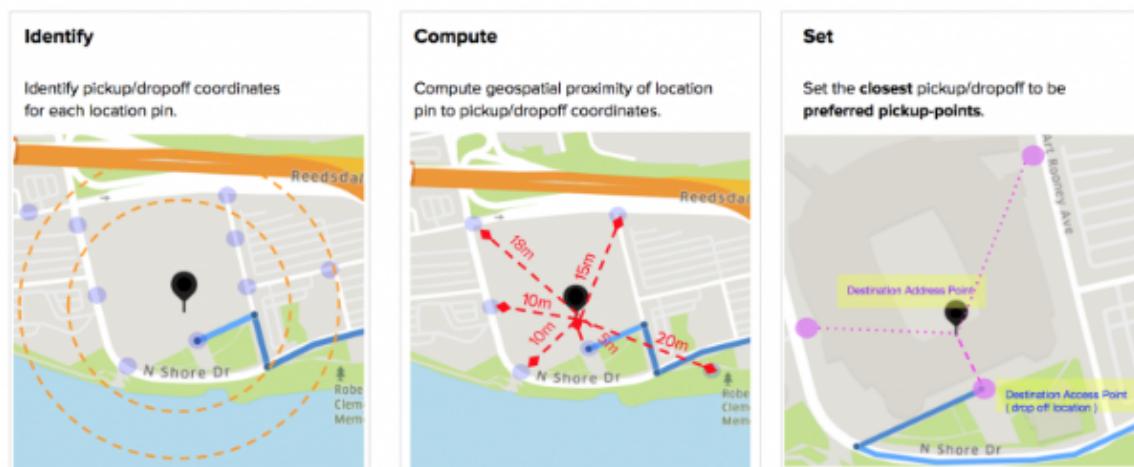


Figure 4: Refining access points is composed of three main steps: identify actual pick-up and drop-off locations used by drivers for a place or address (left), compute the distances from those locations to the place or address (middle), and then set the preferred access point based on the shortest distance (right).

Image Source: <https://eng.uber.com/maps-metrics-computation/>

The shortest distance (closest pickup point) is then calculated and we set the pin to that location as a preferred access point on the map. When a rider requests the location indicated by the map pin, the map guides the driver to the preferred access point. The calculation continues with the latest actual pick-up and drop-off locations to ensure the freshness and accuracy of the suggested preferred access points. Uber uses machine learning and different algorithms to figure out the preferred access point.

9. How ETAs Are Calculated?

ETA is an extremely important metric in Uber because it directly impacts ride-matching and earnings. ETA is calculated based on the road system (not geographically) and there are a lot of factors involved in computing the ETA (like heavy traffic or road construction). When a rider requests a cab from a location the app not only identifies the free/idle cabs but also includes the cabs which are about to finish a ride. It may be possible that one of the cabs which are about to finish the ride is more close to the demand than the cab which is far away from the user. So many uber cars on the road send GPS locations every 4 seconds, so to predict traffic we can use the driver's app's GPS location data.

We can represent the entire road network on a graph to calculate the ETAs. We can use AI-simulated algorithms or simple **Dijkstra's algorithm** to find out the best route in this graph. In that graph, nodes represent intersections (available cabs), and edges represent road segments. We represent the road segment distance or the traveling time through the edge weight. We also represent and model some additional factors in our graph such as one-way streets, turn costs, turn restrictions, and speed limits.

Once the data structure is decided we can find the best route using Dijkstra's search algorithm which is one of the best modern routing algorithms today. For faster performance, we also need to use OSRM (Open Source Routing Machine) which is based on contraction hierarchies. Systems based on contraction hierarchies take just a few milliseconds to compute a route — by preprocessing the routing graph.

10. Databases

Uber had to consider some of the requirements for the database for a better customer experience. These requirements are...

- The database should be horizontally scalable. You can linearly add capacity by adding more servers.
- It should be able to handle a lot of reads and writes because once every 4-second cabs will be sending the GPS location and that location will be updated in the database.
- The system should never give downtime for any operation. It should be highly available no matter what operation you perform (expanding storage, backup, when new nodes are added, etc).

Earlier Uber was using the RDBMS PostgreSQL database but due to scalability issues uber switched to various databases. Uber uses a NoSQL database (schemaless) built on top of the MySQL database.

- Redis for both caching and queuing. Some are behind Twemproxy (which provides scalability of the caching layer). Some are behind a custom clustering system.
- Uber uses schemaless (built in-house on top of MySQL), Riak, and Cassandra. Schemaless is for long-term data storage. Riak and Cassandra meet high-availability, low-latency demands.
- MySQL database.
- Uber is building their own distributed column store that's orchestrating a bunch of MySQL instances.

11. Analytics

To optimize the system, minimize the cost of the operation and for better customer experience uber does log collection and analysis. Uber uses different tools and frameworks for analytics. For log analysis, Uber uses multiple Kafka clusters. Kafka takes historical data along with real-time data. Data is archived into Hadoop before it expires from Kafka. The data is also indexed into an Elastic search stack for searching and visualizations. Elastic search does some log analysis using Kibana/Graphana. Some of the analyses performed by Uber using different tools and frameworks are...

- Track HTTP APIs
- Manage profile
- Collect feedback and ratings
- Promotion and coupons etc
- Fraud detection
- Payment fraud
- Incentive abuse by a driver
- Compromised accounts by hackers. Uber uses historical data of the customer and some machine learning techniques to tackle this problem.

12. How To Handle The Datacenter Failure?

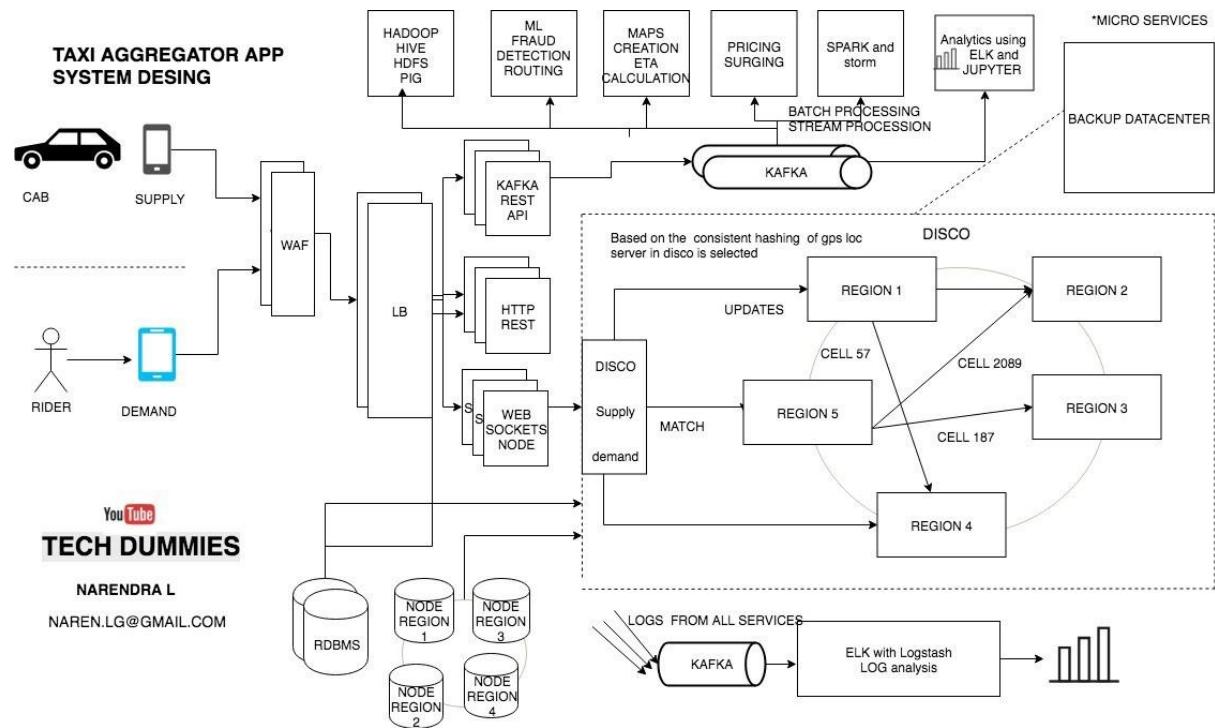
Datacenter failure doesn't happen very often but Uber still maintains a backup data center to run the trip smoothly. This data center includes all the components but Uber never copies the existing data into the backup data center.

Then how does Uber tackle the data center failure??

It actually uses driver phones as a source of trip data to tackle the problem of data center failure.

When The driver's phone app communicates with the dispatch system or the API call is happening between them, the dispatch system sends the encrypted state digest (to keep track of the latest information/data) to the driver's phone app. Every time this state digest will be received by the driver's phone app. In case of a data center failure, the backup data center (backup DISCO) doesn't know anything about the trip so it will ask for the state digest from the driver's phone app and it will update itself with the state digest information received by the driver's phone app.

Solution 3:



Uber's technology may look simple but when A user requests a ride from the app, and a driver arrives to take them to their destination.

But Behind the scenes, however, a giant infrastructure consisting of thousands of services and terabytes of data supports each and every trip on the platform.

Like most web-based services, the Uber backend system started out as a “monolithic” software architecture with a bunch of app servers and a single database

The system was mainly written in Python and used [SQLAlchemy](#) as the [ORM](#)-layer to the database. The original architecture was fine for running a relatively modest number of trips in a few cities.

After 2014 the architecture has evolved into a Service-oriented architecture with about 100s of services

Uber's backend is now not just designed to handle taxies, instead, it can handle taxi, food delivery and cargo also

The backend is primarily serving mobile phone traffic. uber app talks to the backend over mobile data.

The challenging thing is to supply demand with a variable supply!!

Uber's Dispatch system acts like a real-time market platform that matches drivers with riders using mobile phones.

So we need two services

1. Supply service
2. Demand service

going forward I will be using supply for cabs and demand for riders while explaining

Supply service:

- The Supply Service tracks cars using geolocation (lat and lang) Every cab which is active keep on sending lat-long to the server every 5 sec once
- The state machines of all of the supply also kept in memory
- To track vehicles there are many attributes to model: number of seats, type of vehicle, the presence of a car seat for children, can a wheelchair be fit, and so on.
- Allocation needs to be tracked. A vehicle, for example, may have three seats but two of those are occupied.

Demand service

- The Demand Service tracks the GPS location of the user when requested
- It tracks requirements of the orders like Does a rider require small car/big car or pool etc
- demand requirements must be matched against supply inventory.

Now we have supply and demand. all we need a service which matches they demand to a supply and that service in UBER is called as DISCO

DISCO — DISPATCH optimization

This service runs on hundreds of processes.

Core requirements of the dispatch system

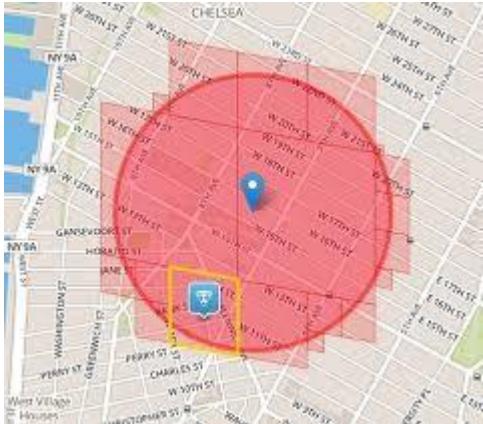
1. reduce extra driving.
2. reduce waiting time
3. lowest overall ETA

How does the Dispatch system work?? How riders are matched to drivers ??

GPS/ location data is what drive dispatch system, that means we have to model our maps and location data

1. The earth is a sphere. It's hard to do summarization and approximation based purely on longitude and latitude. So Uber divides the earth into tiny cells using the Google S2 library. Each cell has a unique cell ID.

2. S2 can give the coverage for a shape. If you want to draw a circle with a 1km radius centered on London, S2 can tell what cells are needed to completely cover the shape



1. Since each cell has an ID the ID is used as a sharding key. When a location comes in from supply the cell ID for the location is determined. Using the cell ID as a shard key the location of the supply is updated. It is then sent out to a few replicas.
2. To match riders to drivers or just display cars on a map, DISCO sends a request to geo by supply
3. the system filters all cabs by rider's GPS location data to get nearby cabs that meet riders requirements Using the cell IDs from the circle area all the relevant shards are contacted to return supply data.
4. Then the list and requirements are sent to routing / ETA to compute the ETA of how nearby they are not geographically, but by the road system.
5. Sort by ETA then sends it back to supply system to offer it to a driver.

How To Scale Dispatch System?

There are many ways you can build, but @ uber

1. Dispatch is built using node.js the advantage with using node is the asynchronous and event-based framework. also, it enables you to send and receive messages over WebSockets
2. so anytime client can send the message to server or server can send and whenever it wants to.
3. Now how to distribute dispatch computation on the same machine and to multiple machines?
4. The solution to scaling is Node js with *ringpop*, it is faster RPC protocol with gossip using SWIM protocol along with a consistent hash ring.
5. Ringpop is a library that brings cooperation and coordination to distributed applications. It maintains a consistent hash ring on top of a membership protocol and provides request forwarding as a routing convenience. It can be used to shard your application in a way that's scalable and fault tolerant
6. SWIM is used gossip/to know what node does what and who takes which geo's computation's responsibility.
7. so with gossip it's easy to add and remove nodes and hence scaling is easy

8. Gossip protocol SWIM also combines health checks with membership changes as part of the same protocol.

How supply sends messages and saved?

Apache **Kafka** is used as the data hub

supply or cabs uses Kafka's APIS to send there accurate GPS locations to the datacenter.

Once the GPS locations are loaded to Kafka they are slowly persisted to the respective worker notes main memory and also to the DB when the trip is happening.

How do Maps and routing work?

Before Uber launches operations in a new area, we define and onboard a new region to our map technology stack. Inside this map region, we define subregions labeled with grades A, B, AB, and C, as follows:

Grade A: A subregion of Uber Territory covering urban centers and commute areas that makeup approximately 90 percent of all expected Uber traffic. With that in mind, it is of critical importance to ensure the highest map quality of grade A map regions.

Grade B: A subregion of Uber Territory covering rural and suburban areas that might be less populated or less traveled by Uber customers.

Grade AB: A union of grade A and B subregions.

Grade C: A set of highway corridors connecting various Uber Territories.

GeoSpatial design:

The earth is a sphere. It's hard to do summarization and approximation based purely on longitude and latitude.

So Uber divides the earth into tiny cells using the Google S2 library. Each cell has a unique cell ID.

When DISCO needs to find the supply near a location, a circle's worth of coverage is calculated centered on where the rider is located.

The read load is scaled through the use of replicas. If more read capacity is needed the replica factor can be increased.

How uber builds the Map?

1. Trace coverage: A comparative coverage metric, trace coverage identifies missing road segments or incorrect road geometry. The computation uses two inputs: map data under testing and historic GPS traces of all Uber rides taken over a certain period of time. We overlay those GPS traces onto the map, comparing and matching them with road segments. If we find GPS traces where no road is shown, we can infer that our map is missing a road segment and take steps to fix the deficiency.

2. Preferred access (pick-up) point accuracy: Pick-up points are an extremely important metric to the rider experience, especially at large venues such as airports and stadiums. For this metric, we compute the distance of an address or place's location, as shown by the map pin in Figure 4, below, from all actual pick-up and drop-off points used by drivers. We then set the closest actual location to be the preferred access point for the said location pin. When a rider requests the location indicated by the map pin, the map guides the driver to the preferred access point. We continually compute this metric with the latest actual pick-up and drop-off locations to ensure the freshness and accuracy of the suggested preferred access points.

How ETAs are calculated?

that means disco should track the cabs available to ride the riders.

but IT shouldn't just handle currently available supply, i.e. cabs which are ready to ride customer but also tracks the cars about to finish a ride.

for example:

1. a cab which is about to finish near the demand(rider) is better than allocating the cab which is far away from the demand.
2. Sometimes revising a route of an ongoing trip because some cab near to demand came online.

when uber started every cities data was separated by creating separated tables/DB this was not easy

now all the cities computation happens in the same system, since the workers the DBnodes are distributed by regions the demand request will be sent to the nearest datacenter.

Routing and Calculating ETA is important component in uber as it directly impacts ride matching and earnings.

so it uses historical travel times to calculate ETAs

you can use AI simulated algorithms or simple Dijkstra's also to find the best route

Also you can use Driver's app's GPS location data to easily predict traffic condition at any given road as there are so many uber cars on the road which is sending GPS locations every 4 seconds

The whole road network is modeled as a graph. Nodes represent intersections, and edges represent road segments. The edge weights represent a metric of interest: often either the road segment distance or the time take it takes to travel through it. Concepts such as one-way streets, turn restrictions, turn costs, and speed limits are modeled in the graph as well.

One simple example you can try at home is the [Dijkstra's search algorithm](#), which has become the foundation for most modern routing algorithms today.

OSRM is based on [contraction hierarchies](#). Systems based on contraction hierarchies achieve fast performance — taking just a few milliseconds to compute a route — by preprocessing the routing graph.

Databases:

A lot of different databases are used. The oldest systems were written in Postgres.

Redis is used a lot. Some are behind Twemproxy. Some are behind a custom clustering system.

MySQL: Built on these requirements

- linearly add capacity by adding more servers (Horizontally scalable)
- write availability with buffering using Redis
- Triggers should work when there is a change in the instance
- No downtime for any operation (expanding storage, backup, adding indexes, adding data, and so forth).

You can use, Google's [Bigtable](#) like any schema-less database

Trip data Storage in Schemaless

Uber is building their own distributed column store that's orchestrating a bunch of MySQL instances called schemaless

Schemaless is key-value store which allows you to save any JSON data without strict schema validation in a schemaless fashion (hence the name).

It has append-only sharded [MySQL](#) with buffered writes to support failing MySQL masters and a publish-subscribe feature for data change notification which we call triggers.

Schemaless supports global indexes over the data.

Trip data is generated at different points in time, from pickup drop-off to billing, and these various pieces of info arrive asynchronously as the people involved in the trip give their feedback, or background processes execute.

A trip is driven by a partner, taken by a rider, and has a timestamp for its beginning and end. This info constitutes the base trip, and from this we calculate the cost of the trip (the fare), which is what the rider is billed. After the trip ends, we might have to adjust the fare, where we either credit or debit the rider. We might also add notes to it, given feedback from the rider or driver (shown with asterisks in the diagram above). Or, we might have to attempt to bill multiple credit cards, in case the first is expired or denied.

Some of the Dispatch services are keeping state in Riak.

Geospatial data and trips DB

The design goal is to handle a million GPS points writes per second

Read is even more as for every rider we need to show at least 10 nearby cabs using Geo hash and Google s2 library all the GPS locations can be queried

Now let's talk about ANALYTICS

Log collection and analysis

Every micro-services or service logging services are configured to push logs to a distributed Kafka cluster and then using log stash we can apply filters on the messages and redirect them to different sources,

for example, Elastic search to do some log analysis using Kibana/Graphana

1. Track HTTP APIs
2. To manage profile
3. To collect feedback and ratings
4. Promotion and coupons etc
5. FRAUD DETECTION
6. Payment fraud
7. Incentive abuse
8. Compromised accounts

Load balance:

Layer 7, Layer 4 and Layer 3 Load Balancer

- Layer 7 in application load balancing
- layer 4 is based on IP + port/ TCP or DNS based load balance
- Layer 3 is based on only IP address

Post-trip actions:

once the trip is completed we need to do these actions by scheduling

- Collect ratings.
- Send emails.
- Update databases.
- Schedule payments.

PRICE AND SURGE:

The price is increased when there are more demand and less supply with the help of prediction algorithms.

According to UBER surge helps to meet supply and demand. by increasing the price more cabs will be on the road when the demand is more.

How To Handle Total Datacenter Failure?

- It doesn't happen very often, but there could be an unexpected cascading failure or an upstream network provider could fail.
- Uber maintains a backup data center and the switches are in place to route everything over to the backup datacenter.
- The problem is the data for in-process trips may not be in the backup datacenter. Rather than replicate data they use driver phones as a source of trip data.
- What happens is the Dispatch system periodically sends an encrypted State Digest down to driver phones. Now let's say there's a datacenter failover. The next time the driver phone sends a location update to the Dispatch system the Dispatch system will detect that it doesn't know about this trip and ask them for the State Digest. The Dispatch system then updates itself from the State Digest and the trip keeps on going like nothing happened.

Solution 4:

Similar Services: Lyft, Didi, Via, Sidecar, etc. Difficulty level: Hard Prerequisite: Designing Yelp

1. What is Uber?

Uber enables its customers to book drivers for taxi rides. Uber drivers use their personal cars to drive customers around. Both customers and drivers communicate with each other through their smartphones using the Uber app.

2. Requirements and Goals of the System

Let's start with building a simpler version of Uber.

There are two types of users in our system: 1) Drivers 2) Customers.

- Drivers need to regularly notify the service about their current location and their availability to pick passengers.
- Passengers get to see all the nearby available drivers.
- Customer can request a ride; nearby drivers are notified that a customer is ready to be picked up.
- Once a driver and a customer accept a ride, they can constantly see each other's current location until the trip finishes.
- Upon reaching the destination, the driver marks the journey complete to become available for the next ride.

3. Capacity Estimation and Constraints

- Let's assume we have 300M customers and 1M drivers with 1M daily active customers and 500K daily active drivers.
- Let's assume 1M daily rides.
- Let's assume that all active drivers notify their current location every three seconds.

- Once a customer puts in a request for a ride, the system should be able to contact drivers in real-time.

4. Basic System Design and Algorithm

We will take the solution discussed in [Designing Yelp](#) and modify it to make it work for the above-mentioned “Uber” use cases. The biggest difference we have is that our QuadTree was not built keeping in mind that there would be frequent updates to it. So, we have two issues with our Dynamic Grid solution:

- Since all active drivers are reporting their locations every three seconds, we need to update our data structures to reflect that. If we have to update the QuadTree for every change in the driver’s position, it will take a lot of time and resources. To update a driver to its new location, we must find the right grid based on the driver’s previous location. If the new position does not belong to the current grid, we have to remove the driver from the current grid and move/reinsert the user to the correct grid. After this move, if the new grid reaches the maximum limit of drivers, we have to repartition it.
- We need to have a quick mechanism to propagate the current location of all the nearby drivers to any active customer in that area. Also, when a ride is in progress, our system needs to notify both the driver and passenger about the current location of the car.

Although our QuadTree helps us find nearby drivers quickly, a fast update in the tree is not guaranteed.

Do we need to modify our QuadTree every time a driver reports their location? If we don’t update our QuadTree with every update from the driver, it will have some old data and will not reflect the current location of drivers correctly. If you recall, our purpose of building the QuadTree was to find nearby drivers (or places) efficiently. Since all active drivers report their location every three seconds, therefore there will be a lot more updates happening to our tree than querying for nearby drivers. So, what if we keep the latest position reported by all drivers in a hash table and update our QuadTree a little less frequently? Let’s assume we guarantee that a driver’s current location will be reflected in the QuadTree within 15 seconds. Meanwhile, we will maintain a hash table that will store the current location reported by drivers; let’s call this DriverLocationHT.

How much memory we need for DriverLocationHT? We need to store DriveID, their present and old location, in the hash table. So, we need a total of 35 bytes to store one record:

- DriverID (3 bytes - 1 million drivers)
- Old latitude (8 bytes)
- Old longitude (8 bytes)
- New latitude (8 bytes)
- New longitude (8 bytes) Total = 35 bytes

If we have 1 million total drivers, we need the following memory (ignoring hash table overhead):

1 million * 35 bytes => 35 MB

How much bandwidth will our service consume to receive location updates from all drivers? If we get DriverID and their location, it will be (3+16 => 19 bytes). If we receive this information every three seconds from 500K daily active drivers, we will be getting 9.5MB per three seconds.

Do we need to distribute DriverLocationHT onto multiple servers? Although our memory and bandwidth requirements don't require this, since all this information can easily be stored on one server, but, for scalability, performance, and fault tolerance, we should distribute DriverLocationHT onto multiple servers. We can distribute based on the DriverID to make the distribution completely random. Let's call the machines holding DriverLocationHT the Driver Location server. Other than storing the driver's location, each of these servers will do two things:

1. As soon as the server receives an update for a driver's location, they will broadcast that information to all the interested customers.
2. The server needs to notify the respective QuadTree server to refresh the driver's location. As discussed above, this can happen every 10 seconds.

How can we efficiently broadcast the driver's location to customers? We can have a **Push Model** where the server will push the positions to all the relevant users. We can have a dedicated Notification Service that can broadcast the current location of drivers to all the interested customers. We can build our Notification service on a publisher/subscriber model. When a customer opens the Uber app on their cell phone, they query the server to find nearby drivers. On the server side, before returning the list of drivers to the customer, we will subscribe the customer for all the updates from those drivers. We can maintain a list of customers (subscribers) interested in knowing the location of a driver and, whenever we have an update in DriverLocationHT for that driver, we can broadcast the current location of the driver to all subscribed customers. This way, our system makes sure that we always show the driver's current position to the customer.

How much memory will we need to store all these subscriptions? As we have estimated above, we will have 1M daily active customers and 500K daily active drivers. On average let's assume that five customers subscribe to one driver. Let's assume we store all this information in a hash table so that we can update it efficiently. We need to store driver and customer IDs to maintain the subscriptions. Assuming we will need 3 bytes for DriverID and 8 bytes for CustomerID, we will need 21MB of memory.

(500K * 3) + (500K * 5 * 8) ~= 21 MB

How much bandwidth will we need to broadcast the driver's location to customers? For every active driver, we have five subscribers, so the total subscribers we have:

5 * 500K => 2.5M

To all these customers we need to send DriverID (3 bytes) and their location (16 bytes) every second, so, we need the following bandwidth:

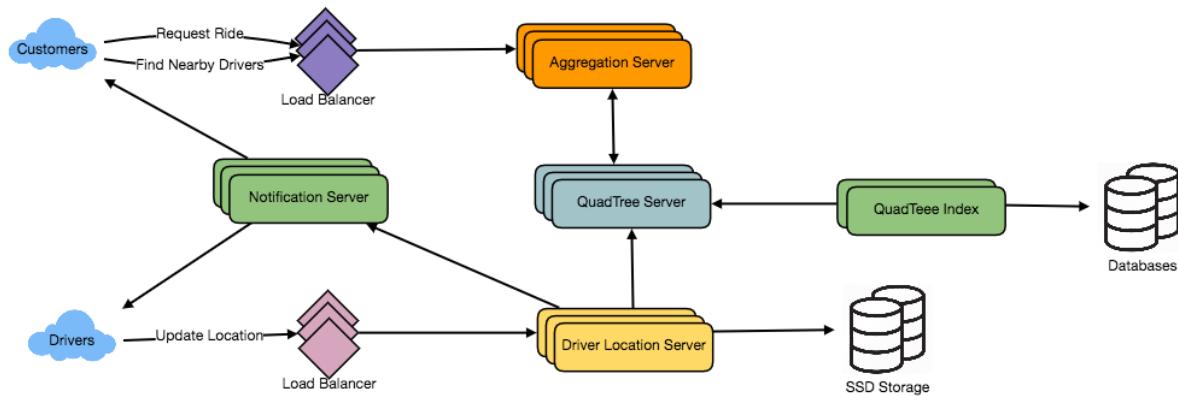
$2.5M * 19 \text{ bytes} \Rightarrow 47.5 \text{ MB/s}$

How can we efficiently implement Notification service? We can either use HTTP long polling or push notifications.

How will the new publishers/drivers get added for a current customer? As we have proposed above, customers will be subscribed to nearby drivers when they open the Uber app for the first time, what will happen when a new driver enters the area the customer is looking at? To add a new customer/driver subscription dynamically, we need to keep track of the area the customer is watching. This will make our solution complicated; how about if instead of pushing this information, clients pull it from the server?

How about if clients pull information about nearby drivers from the server? Clients can send their current location, and the server will find all the nearby drivers from the QuadTree to return them to the client. Upon receiving this information, the client can update their screen to reflect the current positions of the drivers. Clients can query every five seconds to limit the number of round trips to the server. This solution looks simpler compared to the push model described above.

Do we need to repartition a grid as soon as it reaches the maximum limit? We can have a cushion to let each grid grow a little bigger beyond the limit before we decide to partition it. Let's say our grids can grow/shrink an extra 10% before we partition/merge them. This should decrease the load for a grid partition or merge on high traffic grids.



How would “Request Ride” use case work?

1. The customer will put a request for a ride.
2. One of the Aggregator servers will take the request and asks QuadTree servers to return nearby drivers.
3. The Aggregator server collects all the results and sorts them by ratings.
4. The Aggregator server will send a notification to the top (say three) drivers simultaneously, whichever driver accepts the request first will be assigned the ride. The other drivers will receive a cancellation request. If none of the three drivers respond, the Aggregator will request a ride from the next three drivers from the list.

- Once a driver accepts a request, the customer is notified.

5. Fault Tolerance and Replication

What if a Driver Location server or Notification server dies? We would need replicas of these servers, so that if the primary dies the secondary can take control. Also, we can store this data in some persistent storage like SSDs that can provide fast IOs; this will ensure that if both primary and secondary servers die we can recover the data from the persistent storage.

6. Ranking

How about if we want to rank the search results not just by proximity but also by popularity or relevance?

How can we return top rated drivers within a given radius? Let's assume we keep track of the overall ratings of each driver in our database and QuadTree. An aggregated number can represent this popularity in our system, e.g., how many stars does a driver get out of ten? While searching for the top 10 drivers within a given radius, we can ask each partition of the QuadTree to return the top 10 drivers with a maximum rating. The aggregator server can then determine the top 10 drivers among all the drivers returned by different partitions.

7. Advanced Issues

- How will we handle clients on slow and disconnecting networks?
- What if a client gets disconnected when they are a part of a ride? How will we handle billing in such a scenario?
- How about if clients pull all the information, compared to servers always pushing it?

Solution 4:

1. Introduction

When it comes to getting about in cities, many people choose ride-sharing apps like Uber or Lyft. Indeed, such apps make short-range commute very easy by offering competitive pricing, reasonably short wait time, and high availability. From a technical point of view, systems like these are very interesting because nearest-neighbor searching is hard. In this article, I want to share my design of large-scale ride-sharing apps like Uber/Lyft.

Requirements

Modern ride-sharing apps offer complex features such as carpooling, trip sharing, real-time chat in addition to ride-matching. In this article, I want to keep the discussion brief and focus only on the core functionality — booking a ride. Below are the **functional requirements** of our system:

- Users can request a ride and should be matched to a driver in close proximity
- Users can see all nearby drivers (although not choosing which one)
- Drivers can answer/decline requests from nearby riders.

- When a trip is created, both parties see each other's real-time location.

Needless to say, the system should be scalable and highly available.

Traffic Estimation

Before talking about traffics, let's examine a unique feature of Uber — its data usage is local. Unlike apps such as Slack or Instagram, inter-region data communication is rarely needed. If you are physically in New York, you can't book a ride in London. Hence, locational data and trips are not replicated across different regions (less-frequently accessed data such as profiles are replicated globally, of course). In this sense, discussing global traffic is not as meaningful as regional traffic.

The amount of ride requests hitting our system varies depending on the app's popularity. Here we assume significant regional traffic for a generalized solution:

- Group cities by proximity into regions. We need maybe a dozen regions to cover the whole of U.S
- We expect ~100K active drivers in one region.
- We expect ~1M active users in one region.
- The total amount of users globally is 10M.

Assume both drivers and riders emit their location ~5s with each message consisting of longitude, latitude, and additional metadata. In addition, riders regularly check nearby drivers (~5s) and sometimes make a ride request. The amount of QPS and bandwidth we can expect are:

- We expect ~200K location updates/writes per second
- We expect ~200K queries per second, double it for peak traffic handling.
- Each location message consists of ID (8 bytes), longitude and latitude (16 bytes), and other info (64 bytes). The total upload bandwidth is just shy of 88MB/s

2. High-level Design

Database Design

The access pattern of our application determines what schema is used. Let's investigate the requests that hit the database:

Read Operations

- Given a user ID, retrieve its profile
- Given a user ID, retrieve all trips completed by the user
- Given longitude and latitude, query all drivers nearby

Write Operations

- Given a user ID, update its location
- Given a trip ID, the driver can either take/decline the request

Database Schema

Given the access patterns, sharded SQL is a good choice since no complex relationship queries are made. Note that NoSQL databases like Cassandra could also be used here if strong consistency is not a hard requirement.

Driver profile table

driver ID	name	vehicle	plate number	rating	other...
ih03pdp1	John Doe	Toyota Camery	CFY 999	4.7	other...

PK

figure 1. driver profile table, figure by author

Rider profile table

rider ID	name	picture	other...
jdfi5729	Jane Doe	s3://...	other...

PK

figure 2. rider profile table, figure by author

Trip details table

Trip ID	driver ID	rider ID	status	location stamps	other...
uhml1452	ih03pdp1	jdfi5729	FINISHED	[0, 0, 0]	other...

PK Secondary Index

figure 3. trip details table, figure by author

In addition to these database tables, we need another high-performance storage to hold frequently-updated location data. Since live location data is inherently ephemeral, persisting them onto disks does not make sense. A good alternative would be using an in-memory cache such as Redis or Memcache.

Cache Schema

Location truth cache

user ID	longitude	latitude	type	expire
ih03pdp1	12.78	59.02	driver	1646404651
duei12mg	12.79	59.12	rider	1646404658

figure 4. location truth cache, figure by author

This cache is the source of truth when it comes to user location. Phone apps send regular updates to maintain accuracy. If a user gets disconnected, its record expires in 30 seconds.

Driver proximity cache

geohash	drivers (sorted set)
dr5ru7	ih03pdp1, 1646404651, free, XL ifi1pspff, 1646404659, in ride, Black
dr5ru9	ih03pdp1, 1646404653, free, XL h0nvmw, 1646404659, in ride, Black

figure 5. driver proximity cache, figure by author

The proximity cache is crucial for nearby driver search. Given a location, we can use [GeoHash](#) to compute its location key and retrieve all drivers in the grid. I'll talk more about this in the Details section.

Architecture

With a clearer understanding of what data to store, now it's time for service-oriented designs!

- **Notification Service:** whenever the backend needs to send information to the clients, the notification service is used to deliver the messages.
- **Trip Management Service:** When a trip is initiated, this service is needed to monitor the locations of all parties as well as plan routes
- **Ride Matching Service:** This service handles ride requests. It finds nearby drivers and matchmakes based on driver responses (either accept or decline)
- **Location Service:** All users must regularly update their locations via this service.

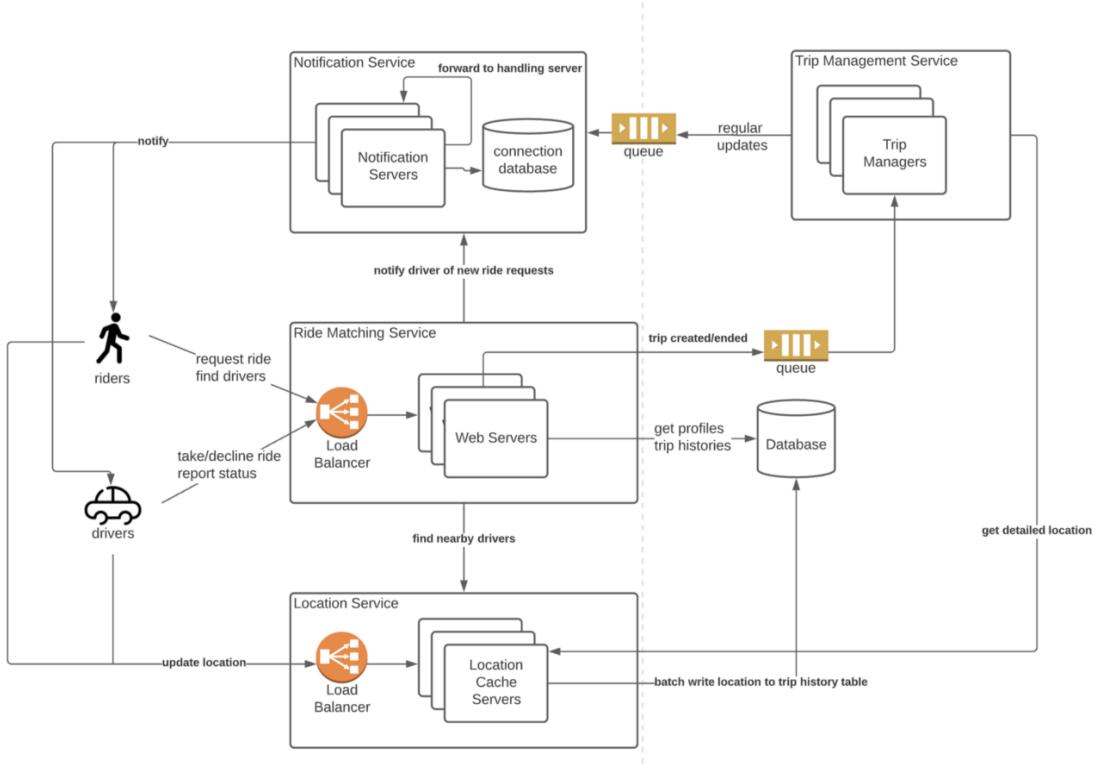


figure 6. high-level architecture, figure by author

Workflow

1. Bob sends a ride request to the Ride Matching service and hopes to pair with a driver.
2. The Rider Matching service contacts the Location service and finds all available drivers in the same region.
3. The Rider Matching service then conducts ranking/filtering and pushes the ride request to selected drivers via the Notification service.
4. Drivers can accept/decline the request. When received, the Ride Matching Service will send the trip details to the Trip Management service.
5. The Trip Management service monitors the trip progress and broadcasts driver&rider locations to all parties involved in the trip.

3. Details

The high-level architecture design is relatively straightforward. However, if you examine carefully, there are still a few wrinkles. For instance, I've ignored features such as trip pricing, carpooling, etc. In this article, I want to focus on location-related problems as they are the foundation of Uber/Lyft.

3.1 How to efficiently look up nearby drivers?

Nearest neighbor search is a hard problem and the scale of our system makes efficient lookup even more difficult. Instead of computing the distance between the rider and every driver in the database, we can use a technique called GeoHash to convert the user's location

to a unique key that corresponds to one cell in figure 7 and confine the search to a few adjacent cells only.



figure 7. location to GeoHash, PC credit: Movable Type Scripts

Figure 7 demonstrates the key property of GeoHash — a one-to-one mapping between keys and location cells. Therefore, we can use the following heuristic to quickly lookup drivers:

1. Given a user location, compute its GeoHash from longitude and latitude.
2. With the GeoHash key available, it's easy to compute the keys for the 8 nearby cells. ([see this post](#))
3. Query the Location service with all 9 keys; retrieve drivers in those cells.

The accuracy of GeoHash is determined by the key length. The longer the key is, the smaller each cell would be.

geohash length	cell size
2	1250Km x 625Km
3	156Km x 156Km
4	39Km x 19.5Km
5	4.89Km x 4.89Km
6	1.22Km x 0.61Km
7	153m x 153m

figure 8. geohash key length and cell size, figure by author

What would be a good key size to use? In practice, the key size is determined iteratively based on the number of drivers and riders. In my opinion, a good starting point would be size 6, as it covers a few city blocks.

Latitude /
 Longitude ,
 Precision
 Geohash

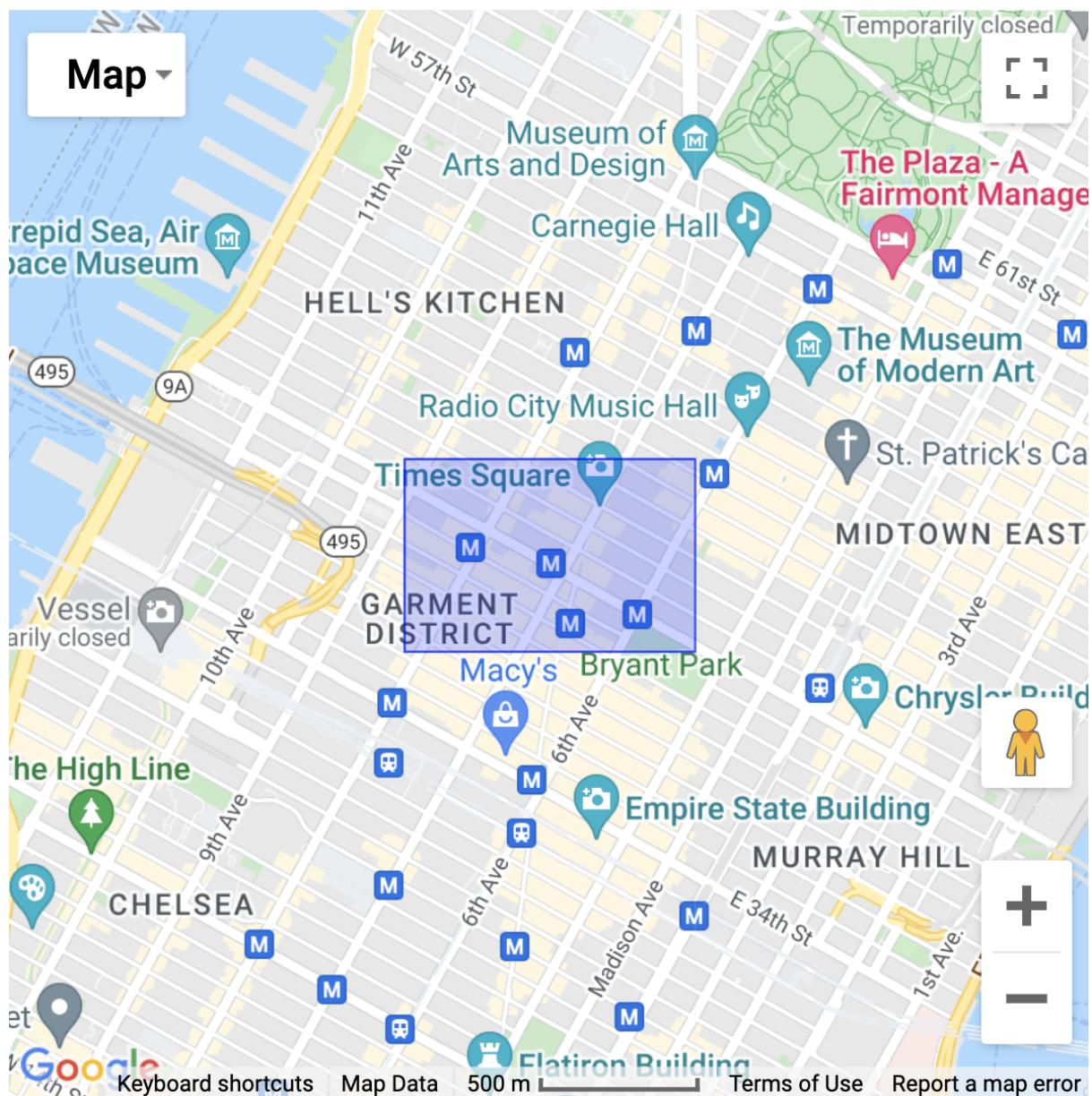


figure 9. size-6 GeoHash offers a good amount of coverage, figure by author

There are physical limitations for the maximum number of drivers and riders that can fit in one of these cells. Going back to figure 5, each cell will be an entry in the Driver Proximity cache in Redis.

You might be wondering if Redis can handle this level of traffic. If there are 1M active riders in the region, the number of requests hitting the cluster is around 200K writes/s (each user updates its location ~5s) and 2M reads/s (each user reads 9 Redis keys ~5s).

Single node results summary

The table below shows how we determined the optimal Redis shards and proxy threads configuration on the AWS c5.18xlarge instance in order to achieve the best possible throughput (in ops/sec) while keeping latency to sub-millisecond:

Shards	4	6	8	8	10
Proxy threads	16	24	28	28	32
Comment	Default	Default	Default	NUMA tuned	NUMA tuned
Latency (msec)	0.92	0.91	0.98	0.81	0.92
Throughput (M ops/sec)	2.9	3.89	4.2	4.8	5.3
Throughput per shard (K ops/sec)	725	648	525	600	530

Figure 10. Single Redis node capacity, PC by [Redis.com](https://redis.com)

With even one single AWS c5.18xlarge node with 32 threads, the system can handle the traffic. In practice, we can distribute the workload to dozens of computers and achieve ~100M level RPS capacity.

What about memory limitations? If we use size 6 geohash, there are potentially $32^6 \approx 10$ billion keys in the cache, which is crazy. However, we will never reach this amount of keys if **empty keys are removed** (figure 5). In practice, the number of cache entries is constrained by the number of cars because each car can only be in one cell! Hence, memory is not the bottleneck.

3.2 Location update

Hopefully I've convinced you of the viability of hosting the Location service on Redis. Now let's examine how users update their locations.

There are two tables in the cache — the location truth table and the driver proximity table. The usage of the location truth table is simple. The user's mobile app sends out its location as well as the user ID to the Location service every 5 seconds. Whenever a user's accurate location is needed, the system can query this table by user ID.

The proximity table is more interesting in nature. Drivers move around in their cars, which means they often cross one cell to the other. When the backend receives updates from a driver, it doesn't know what cells they have been in the past. Hence, it is hard to remove drivers from their old cells. The implication is obvious; the same driver can appear in multiple cells because of the stale data.

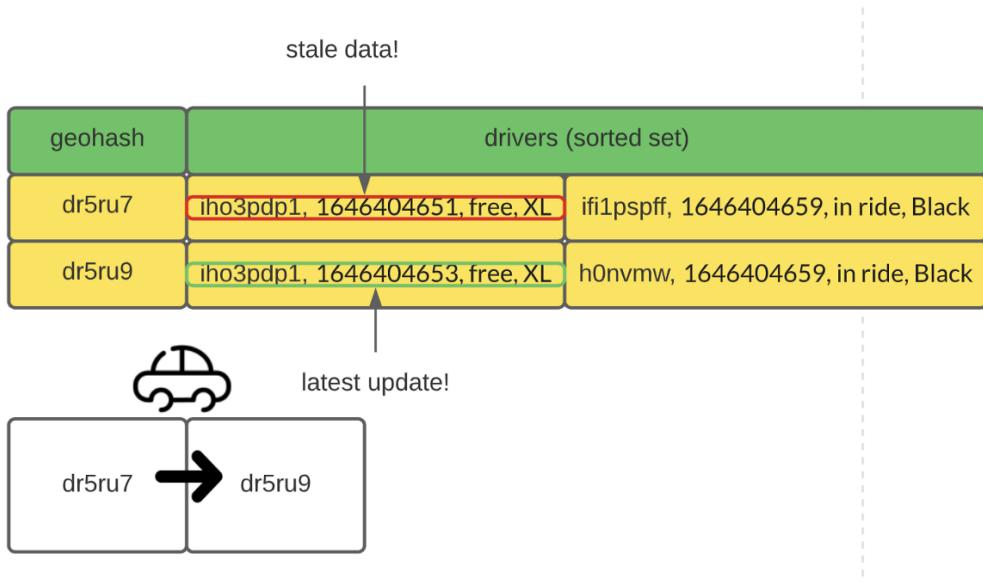


Figure 11. stale records in Redis, figure by author

To deal with this problem, we could introduce a timestamp to each record. With timestamps, it is easy to filter out stale location data. In practice, the sorted set data structure in Redis is an efficient way to achieve such a feature.

In addition to the driver's location, we can keep information such as vehicle type, trip status (free, in ride) in the cache. With the additional information, ride-matching can be made quickly since the round trip to the database (query vehicle information and trip status) is skipped.

3.3 Trip recording

Another important feature of our system is trip recording. For a completed trip, we want to store the route that was taken by the driver and make it available to the client for review.

There are many ways to achieve this feature; the simplest way would be relying on the drivers to report their location as well as their trip status. The Location service will batch write all location updates with *in-trip* status to the database for persistent storage.

Relying on the client app is risky. What if the client app loses all local data while on a trip? It will not know the driver's status before the reboot. To recover from failure, the client app should check if there's any unfinished trip in the database and confirm their status with the driver.

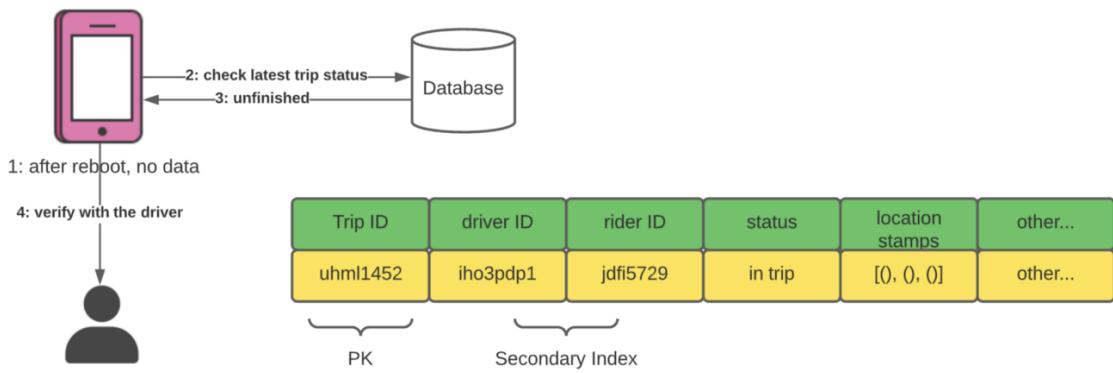


Figure 11. failure recovery, figure by author

Solution 5:

Let's design a ride-sharing service like Uber, connecting a passenger who needs a ride with a driver who has a car. Uber enables its customers to book drivers for taxi rides. Uber drivers use their cars to drive customers around. Both customers and drivers communicate with each other through their smartphones using the Uber app.

Similar Services: Lyft

1. Requirements and Goals of the System

There are two types of users in our system: Drivers and Customers.

- Drivers need to regularly notify the service about their current location and their availability to pick passengers
- Passengers get to see all the nearby available drivers
- Customers can request a ride; this notifies nearby drivers that a customer is ready to be picked up
- Once a driver and a customer accept a ride, they can constantly see each other's current location until the trip finishes.
- Upon reaching the destination, the driver marks the journey complete to be available for the next ride.

2. Capacity Estimation and Constraints

- Assume we have 300 million customers and 1 million daily active customers, and 500K daily active drivers.
- Assume 1 million daily rides
- Let's assume that all active drivers notify their current location every 3 seconds.
- Once a customer puts in a request for a ride, the system should be able to contact drivers in real-time.

3. Basic System Design and Algorithm

a. Grids

We can divide the whole city map into smaller grids to group driver locations into smaller sets. Each grid will store all the drivers locations under a specific range of longitude and latitude. This will enable us to query only a few grids to find nearby drivers. Based on a customer's location, we can find all neighbouring grids and then query these grids to find nearby drivers.

What could be a reasonable grid size?

Let's assume that GridID (say, a 4 byte number) would uniquely identify grids in our system.

Grid size could be equal to the distance we want to query since we also want to reduce the number of grids. We can search within the customer's grid which contains their location and the neighbouring eight grids. Since our grids will be statically defined, (from the fixed grid size), we can easily find the grid number of any driver (lat, long) and its neighbouring grids.

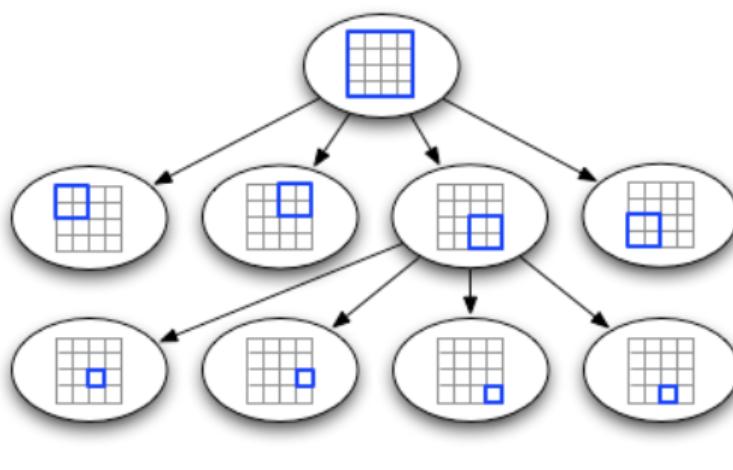
In the DB, we can store the GridID with each location and have an index on it, too, for faster searching.

b. Dynamic size Grids

Let's assume we don't want to have more than 100 drivers locations in a grid so that we can have faster searching. So, whenever a grid reaches this limit, we break it down into four grids of equal size, and distribute drivers among them (of course according to the driver's current location). This means that the city center will have a lot of grids, whereas the outskirts of the city will have large grids with few drivers.

What data structure can hold this information?

A tree in which each node has 4 children. Each node will represent a grid and will contain info about all the drivers' locations in that grid. If a node reaches our limit of 500 places, we will break it down to create 4 child nodes under it and distribute driver locations among them. In this way, all the leaf nodes will represent grids that can't be broken further down. So leaf nodes will keep a list of places with them. This tree structure is called a [QuadTree](#).



How do we build a quad tree?

We'll start with one node that represents the whole city in one grid. Each Uber-available city will have its own quad tree. Since it will have more than 100 locations, we will break it down into 4 nodes and distribute locations among them. We will keep repeating the process with each child node until there are no nodes left with more than 100 driver locations.

How will we find the grid for a given location?

We start with the root node(city) and search downward to find one required node/grid. At each step we will see if the current node we are visiting has children. If it has, we move to the child node that contains our desired location and repeat the process. We stop only if the node does not have any children, meaning that's our desired node(grid).

How will we find neighboring grids of a given grid?

Approach 1: Since our leaf nodes contain a list of locations, we can connect all leaf nodes with a doubly linked list. This way we can iterate forward or backwards among neighbouring leaf nodes to find out our desired driver locations.

Approach 2: Find it through parent nodes. We can keep a pointer in each node to access its parent, and since each parent node has pointers to all its children, we can easily find siblings of a node. We can keep expanding our search for neighboring grids by going up through parent pointers.

Issues with our Dynamic Grid solution:

- Since all active drivers are reporting their locations every 3 seconds, we need to update the QuadTree to reflect that. It will take a lot of time and resources if we have to update it for every change in the driver's coordinates.
- If the new position does not belong in the current grid, we have to remove the driver from the current grid and remove/reinsert the user to the correct grid. After this move, if the new grid reaches the maximum limit of drivers, we have to repartition it.
- We need to have a quick mechanism to propagate the current location of all nearby drivers to any active customer in that area. Also, when a ride is in progress, our system needs to notify both the driver and passenger about the current location of the car.

Although our QuadTree helps us find nearby drivers quickly, a fast update in the tree is not guaranteed.

Do we need to modify our QuadTree every time a driver reports their location?

If we don't, it will have some old data and this won't reflect the current location of drivers correctly. Since all active drivers report their location every 3 seconds, there will be a lot more updates happening to our tree than querying for nearby drivers.

Enter hash table!

We can keep the latest position reported by all drivers in a hash table and update our QuadTree a little less frequently.

Let's assume we guarantee that a driver's current location will be reflected in the QuadTree within 15 seconds. Meanwhile we will maintain a hash table that will store the current location reported by drivers; let's call this **DriverLocationHT**.

How much memory do we need for DriverLocationHT?

We need to store DriverID, their present and old location, in the hash table. So we need a total of 35 bytes to store one record.

1. DriverID (3 bytes - 1 million drivers)
2. Old latitude (8 bytes)
3. Old longitude (8 bytes)
4. New latitude (8 bytes) 5 New longitude (8 bytes) Total ==> 35 bytes

If we have one million drivers, we need:

1 million * 35 bytes ==> 35MB (ignoring hash table overhead)

How much Bandwidth?

To receive location updates from all active drivers, we get DriverID and their location (3 + 16 bytes => 19 bytes). We do this every 3 seconds from 500 active drivers:

19 bytes * 500K drivers ==> 9.5MB per 3 sec.

Do we need to distribute DriverLocationHT Hash Table onto multiple servers?

The memory and bandwidth requirements can be easily handled by one server, but for scalability, performance, and fault tolerance, we should distribute DriverLocationHT onto multiple servers. We can distribute based on the DriverID to make the distribution completely random. Let's call the machines holding DriverlocationHt the Driver location servers.

The servers will:

1. As soon as they receive driver location update, broadcast that information to all interested customers.
2. Notify the respective QuadTree server to refresh the driver's location. This happens every 15 seconds.

Broadcasting driver's location to customers

We can have a **Push Model** where the server pushes all the positions to all the relevant customers.

- A dedicated Notification Service that can broadcast current locations of drivers to all the interested customers.
- Build our Notification service on a publisher/subscriber model. When a customer opens Uber app, they query the server to find nearby drivers. On the back-end, before returning the list of drivers to the customer, we subscribe the customer for all the updates from those nearby drivers.
- We can maintain a list of customers interested in knowing the driver location and whenever we have an update in DriverLocationHT for that driver, we can broadcast the current location of the driver to all subscribed customers. This way, our system ensures that we always show the driver's current position to the customer.

Memory needed to store customer subscriptions

Assume 5 customers subscribe to 1 driver. Let's assume we store this information in a hash table to update it efficiently. We need to store driver and customer IDs to maintain subscriptions.

Assume we need 3 bytes for DriverID, 8 bytes for CustomerID:

$(500K \text{ drivers} * 3 \text{ bytes}) + (500K * 5 \text{ customers} * 8 \text{ bytes}) \approx 21\text{MB}$

How much bandwidth will we need for the broadcast?

For every active driver, we have 5 subscribing customers, so the total subscribers are:

$5 * 500K \Rightarrow 2.5M$

To all these customers, we need to send DriverID(3 bytes) + Location(16 bytes) every second, so we need the following bandwidth:

$(3 + 16) \text{ bytes} * 2.5M \Rightarrow 47.5 \text{ MB/s}$

How can we efficiently implement Notification Service?

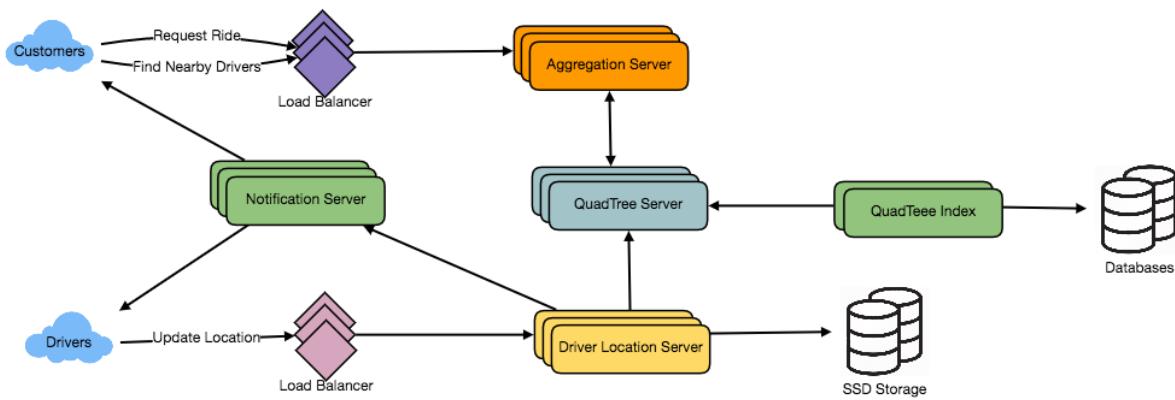
We can use either HTTP long polling or push notifications.

How about Clients pull nearby driver information from server?

- Clients can send their current location, and the server will find all the nearby drivers from the QuadTree to return them to the client.
- Upon receiving this information, the client can update their screen to reflect current positions of drivers.
- Clients will query every five seconds to limit the number of round trips to the server.
- This solution looks simpler compared to the push model described above.

Do we need to repartition a grid as soon as it reaches maximum limit?

We can have a grid shrink/grow an extra 10% before we partition/merge them. This will decrease the load for a grid partition or merge on high traffic grids.



"Requesting a Ride" use case

1. The customer will put a request for a ride
2. One of the Aggregator servers will take the request and asks QuadTree servers to return nearby drivers.
3. The Aggregator server collects all the results and sorts them by ratings.
4. The Aggregator server will send a notification to the top (say three) drivers simultaneously, whichever driver accepts the request first will be assigned the ride. The other drivers will receive a cancellation request.
5. If none of the three drivers respond, the Aggregator will request a ride from the next three drivers from the list.
6. The customer is notified once the driver accepts a request.

Solution 6:

Requirements

In Scope

- Riders should be able to view the available drivers in nearby locations as shown in the image below.

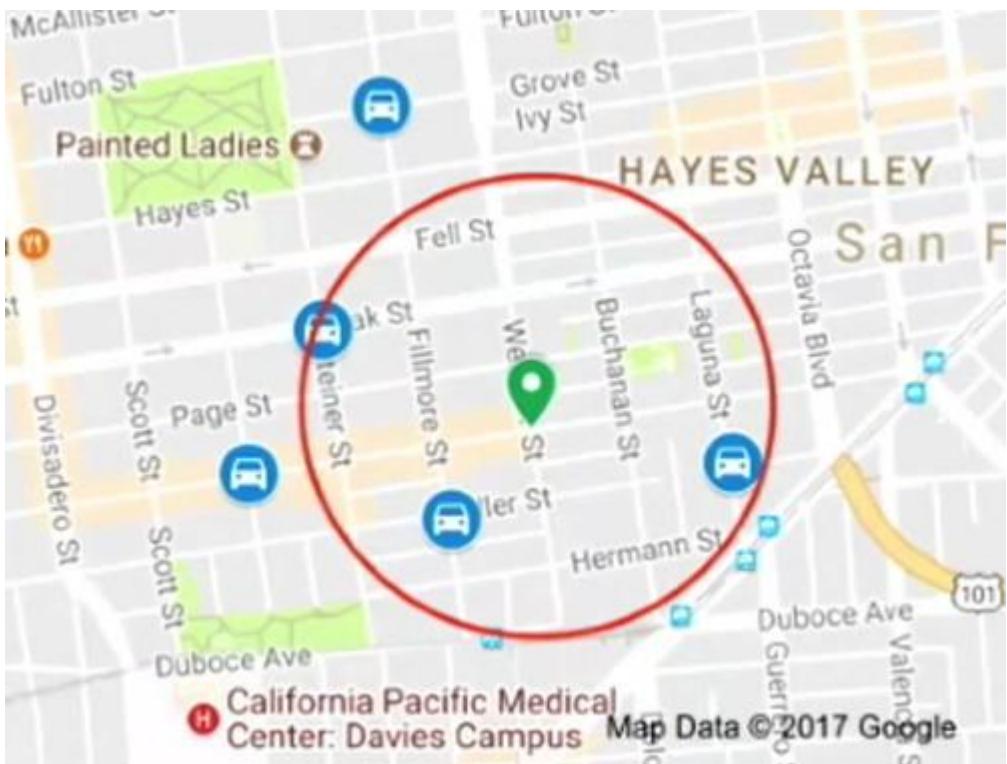


Fig: Showing nearby car locations

- Riders should be able to request rides from source to destination.
- Nearby drivers will be notified of the ride, and one of them will confirm the ride
- Pickup ETA will be shown to customers when a ride is dispatched to pick up the rider.
- Once the trip gets completed, the rider is shown the trip details such as the trip map, price, and so forth in the ride receipt.

Out of Scope

- Automated way of matching drivers to riders taking factors such as new drivers and riders into account when a ride is being dispatched

Architecture

In order to design the system, it's imperative to define the lifecycle of a trip, shown in the image below.

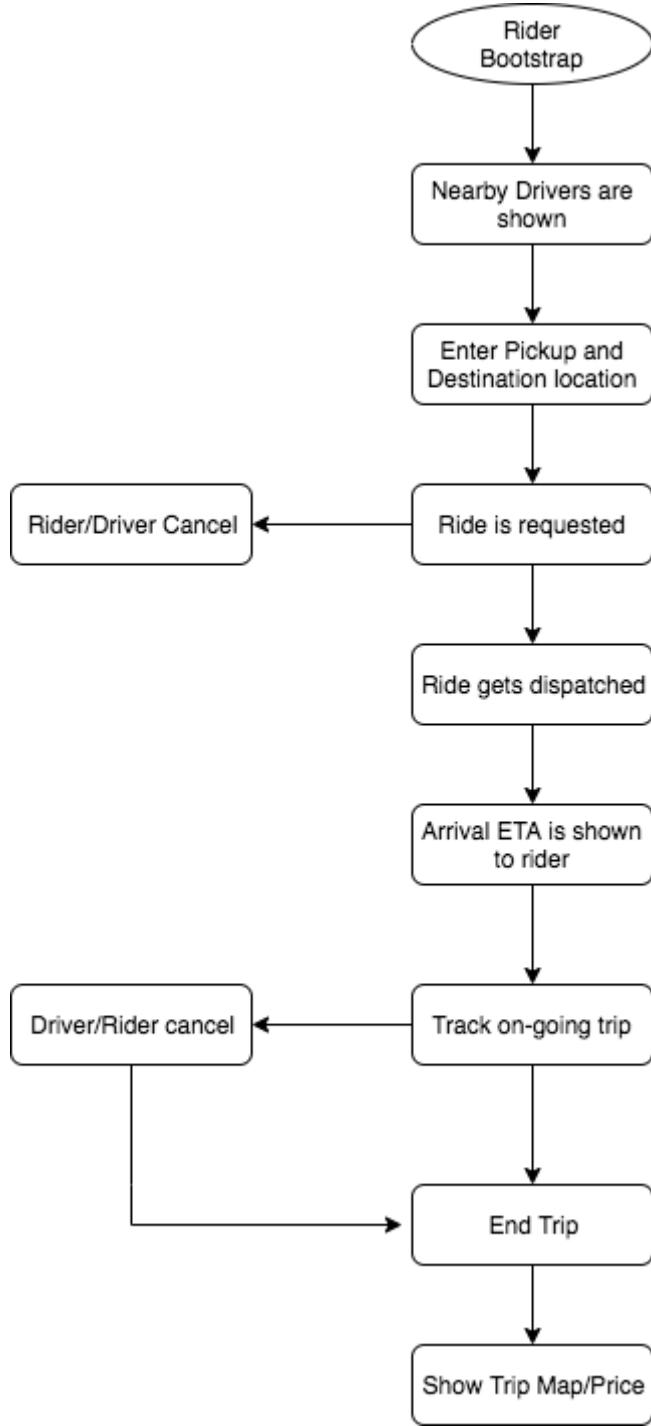


Fig: Lifecycle of a trip

We can build the following components to support this trip lifecycle. These components will work together to meet the requirements of building a ride-hailing system.

- **Driver Location Manager:** This component will be responsible for maintaining the changing driver locations. It will ingest the messages emitted by the drivers' uber app containing their current locations and update the car location index.
- **Trip Dispatcher:** It will be responsible for handling the trip requests from users and dispatch a driver in response to those requests.

- **Arrival ETA Calculator:** This component will be responsible for calculating the ETA for the driver to reach the rider once the driver has accepted the ride.
- **Trip Recorder:** It will record the GPS signals transmitted from the ride when the trip is in progress. These GPS signals will then be recorded in a data store which will be used by subsequent systems such as Map Matcher and Pricing Calculator.
- **Map Matcher:** This component will be responsible for matching the trip on the actual map using algorithms specialized for this purpose.
- **Price Calculator:** It will use the recorded trip information for computing the price which users have to pay for the trip.

The components mentioned above can be grouped into three major categories: Pre-Trip Components, On-Trip Components, and Data Storage. We have shown each of the components in the image below.

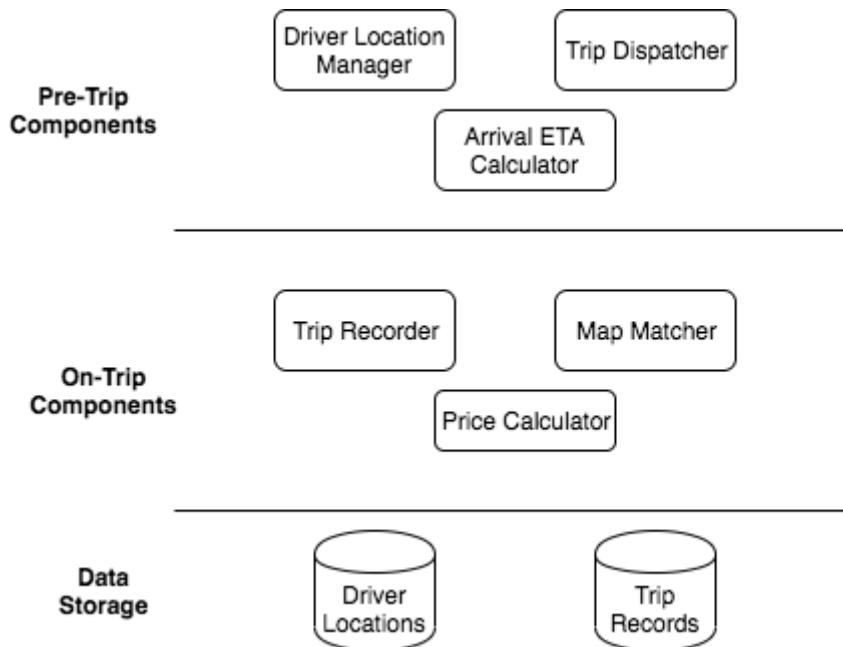


Fig: High-level components in a ride hailing system

High Level Design

Pre-Trip Component

This component will be supporting the functionality to see the cars in the nearby locations and dispatching the ride on users' requests. The location of the cars can be stored in an in-memory car location index (explained in a later section), which can support high read and write requests. The driver app will keep on sending the updated car locations, which will be updated in this index. The index will be queried when rider app requests for nearby car locations or places a ride request. The sequence of operations leading to the ride dispatch is shown in the image below.

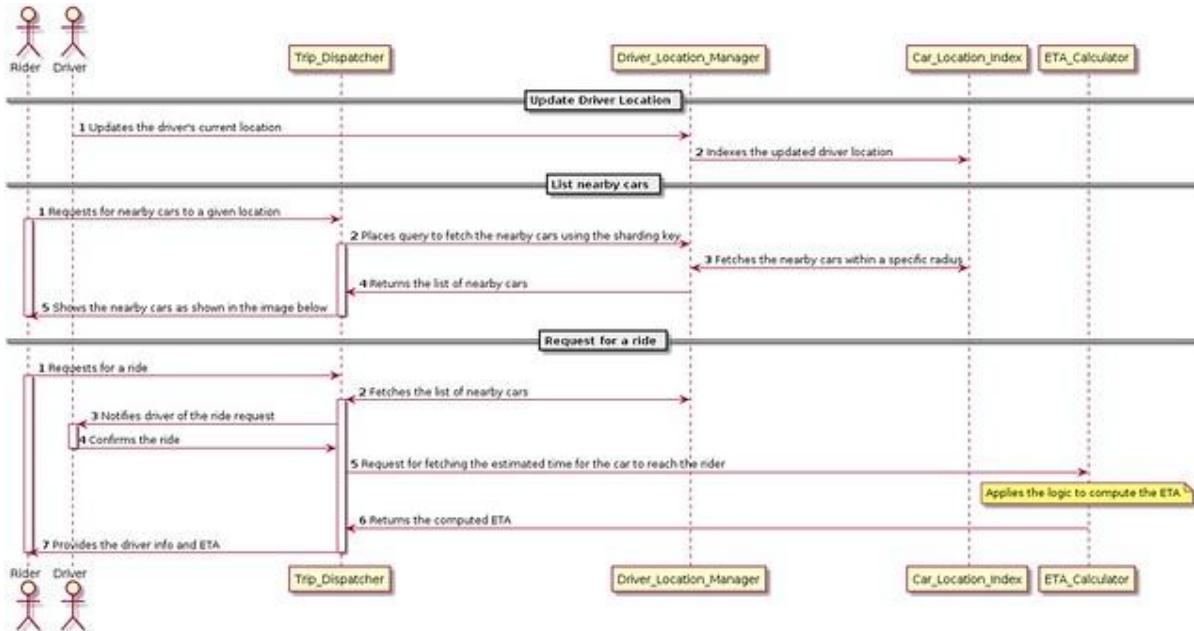


Fig: Sequence of Operations in the Pre-Trip Component

There are three sequences of operations before a trip starts, as shown in the image above: i) Update Driver Location ii) List nearby drivers iii) Requesting for a ride. The first sequence of operations involves updating the driver locations as drivers keep changing their locations. The second sequence comprises of the steps involved in fetching the list of nearby drivers. The third sequence includes a series of steps involved in requesting and dispatching a ride.

Car Location Index

We will maintain the driver locations in a distributed index, which is spread across multiple nodes. The driver information containing their locations, the number of people in the car, and if they are on a trip is aggregated as objects. These driver objects are distributed across the nodes using some sharding mechanism (e.g., city, product, and so forth) along-with replication.

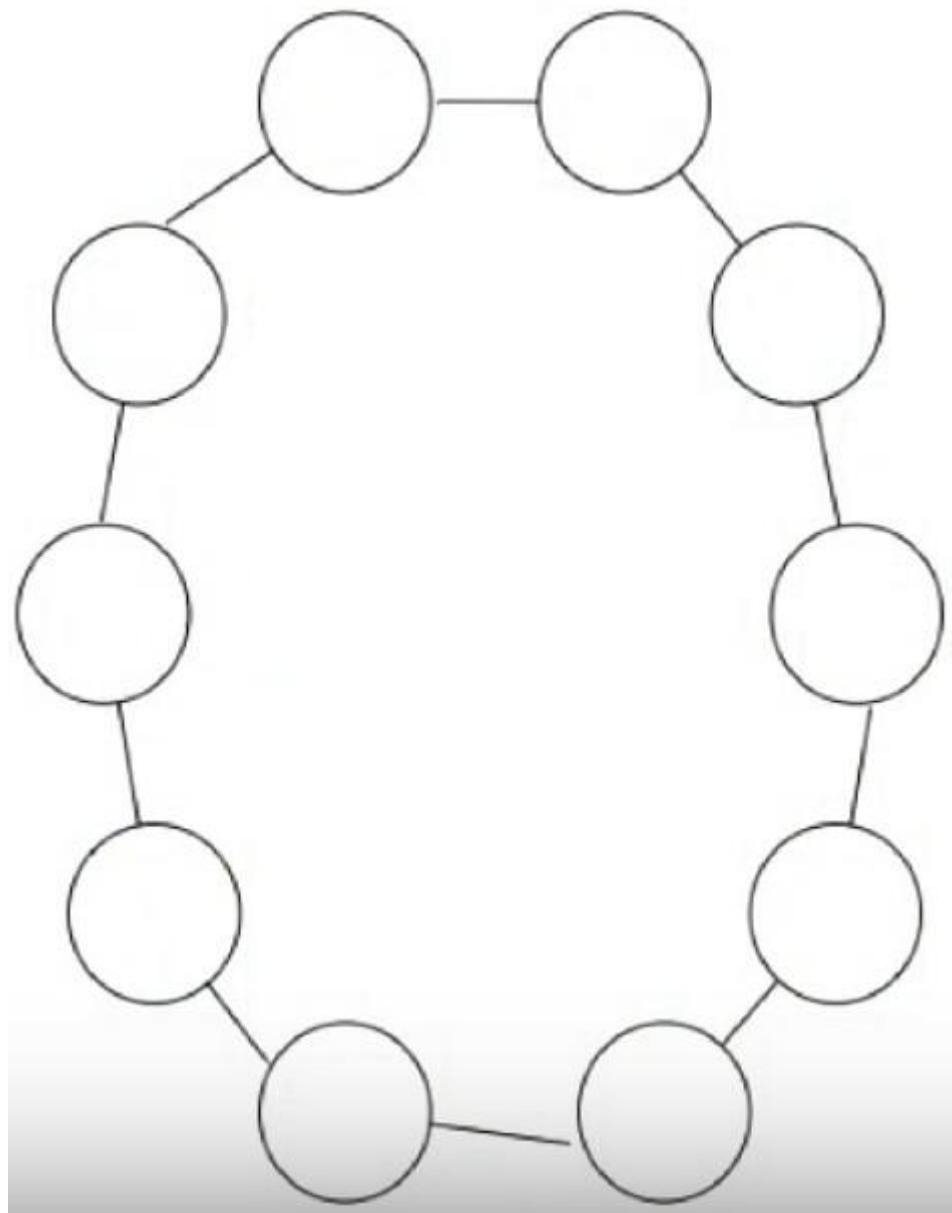


Fig: Distributed Car Location Index

Each node maintains a list of drivers objects which are queried in real-time to fetch the list of nearby drivers. On the other hand, these objects are updated when the driver app sends updates about their location or any other related information. Some interesting characteristics of this index are listed below.

- It should store the current locations of all drivers and should support fast queries by location and properties such as driver's trip status.
- It should support a high volume of reads and writes.
- The data stored in these indexes will be ephemeral, and we don't need long-term durable storage.

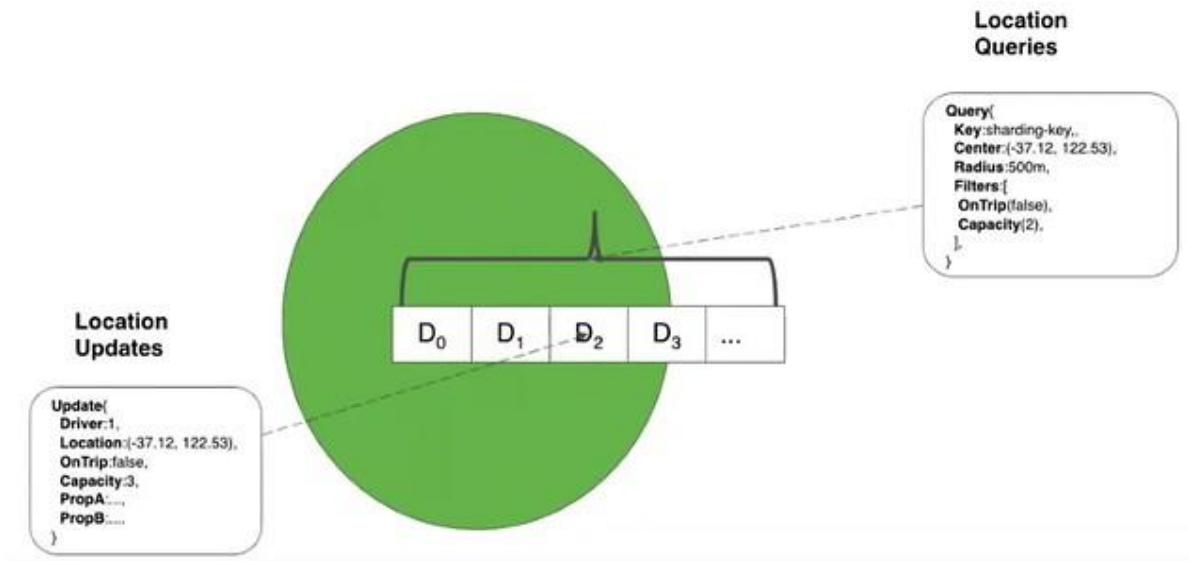


Fig: Updating and Querying Car Location Index

ETA Calculator

We need to show the pickup ETA to users once their ride has been dispatched by taking factors such as route, traffic, and weather into account. The two primary components of estimating an ETA given an origin and destination on a road network include i) computing the route from origin to destination with the least cost (which is a function of time & distance) ii) estimate the time taken to traverse the route.

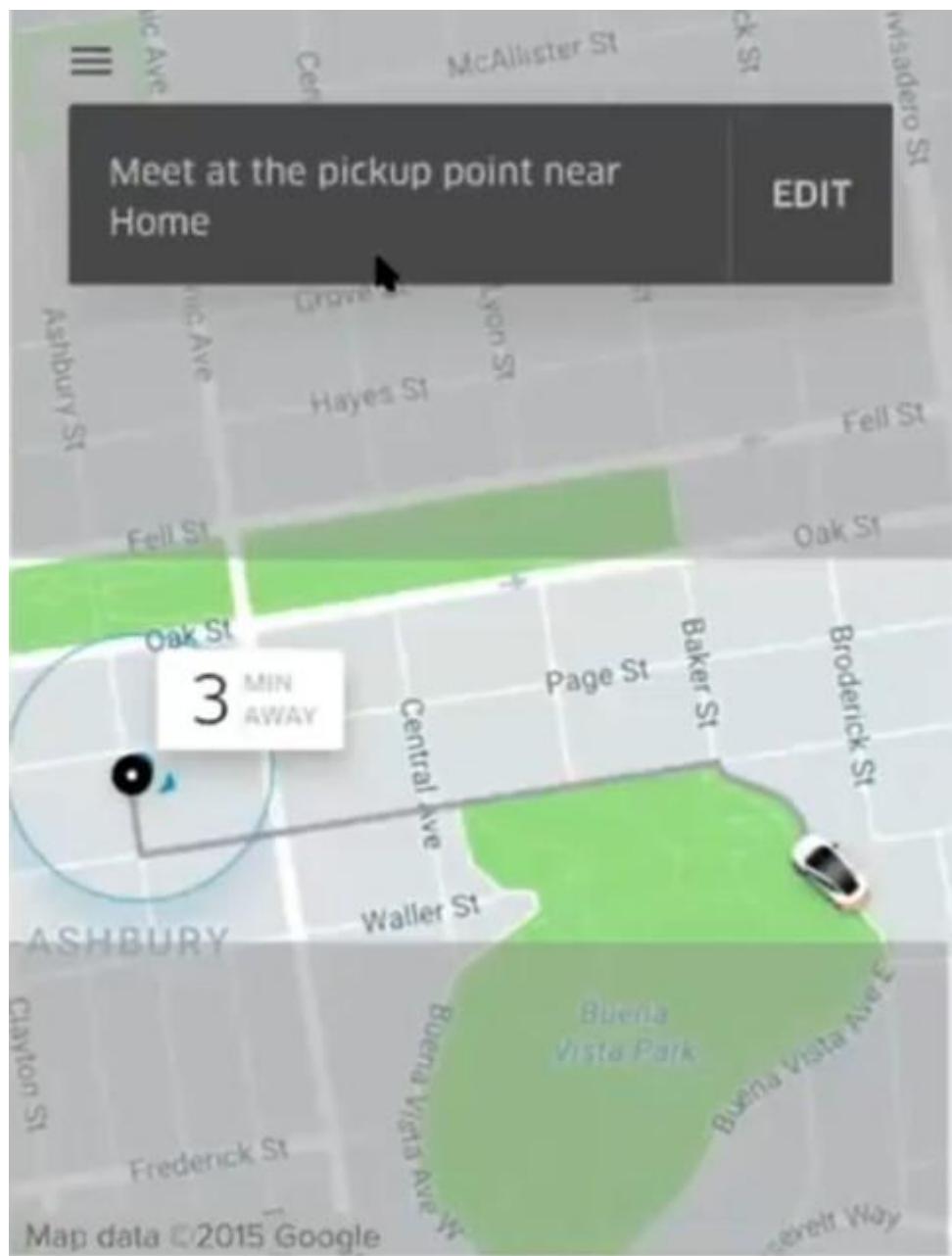


Fig: Sample ETA shown to customer

We can start by representing the physical map by a graphical representation to facilitate route computation. We do this by modeling every intersection by node and each road segment by a directed edge. We have shown the graphical representation of the physical map in the image below.

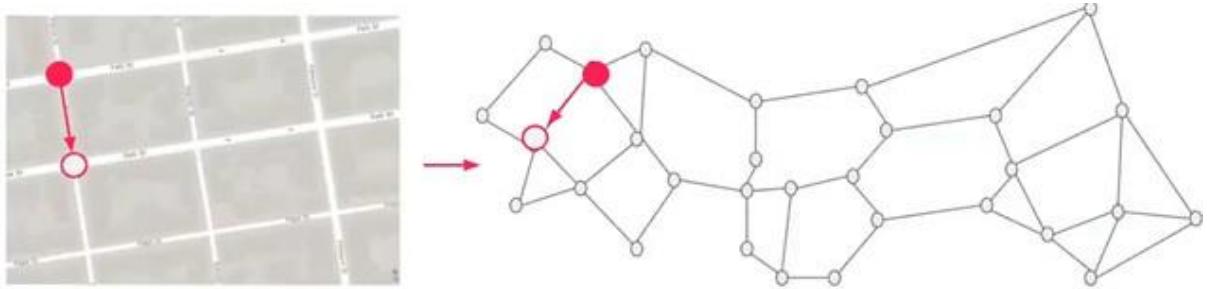


Fig: Graph Representation of physical map

We can use routing algorithms such as Dijkstra's algorithm to find the shortest path between source and destination. However, the caveat associated with using Dijkstra's algorithm is that the time to find the shortest path for "N" nodes is $O(N \lg N)$ which renders this approach infeasible for the scale at which these ride hailing platforms function. We can solve this problem by partitioning the entire graph, pre-computing the best path within partitions, and interacting with just the boundaries of the partitions. In the image below, we have shown the manner in which this approach can help in significantly reducing the time complexity from $O(N \lg N)$ to $O(N' \lg N')$, where N' is square root of N .

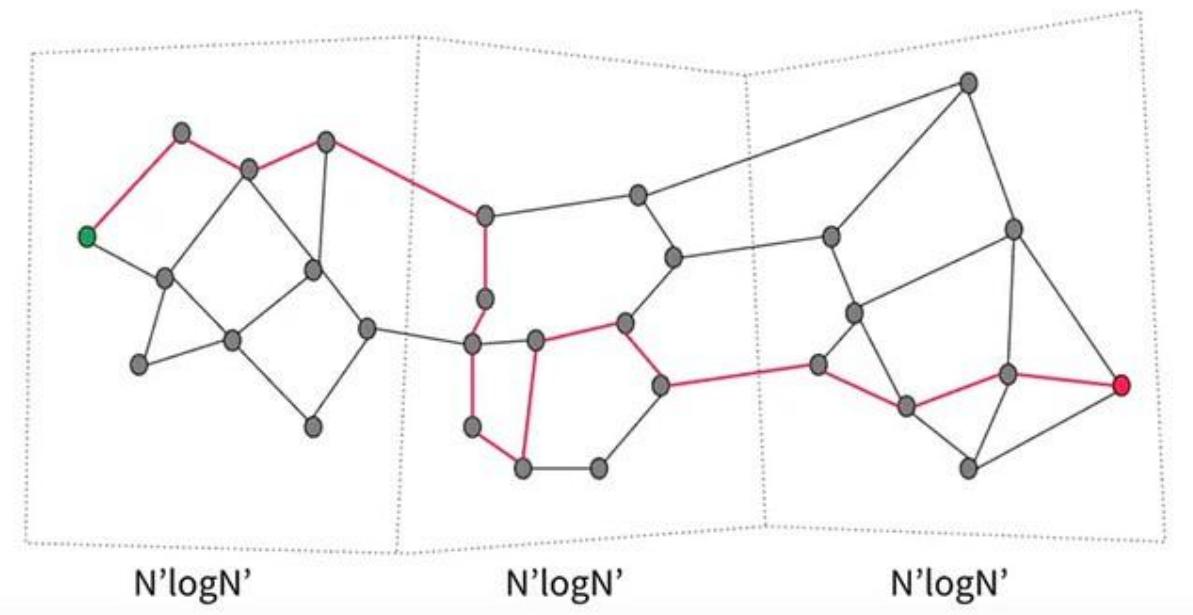


Fig: Partitioning Algorithms to find the optimal route

Once we are aware of the optimal route we find the estimated time to traverse the road segment by taking traffic into account which is a function of time, weather and real-life events. We use this traffic information to populate the edge weights on the graph, as shown in the image below.

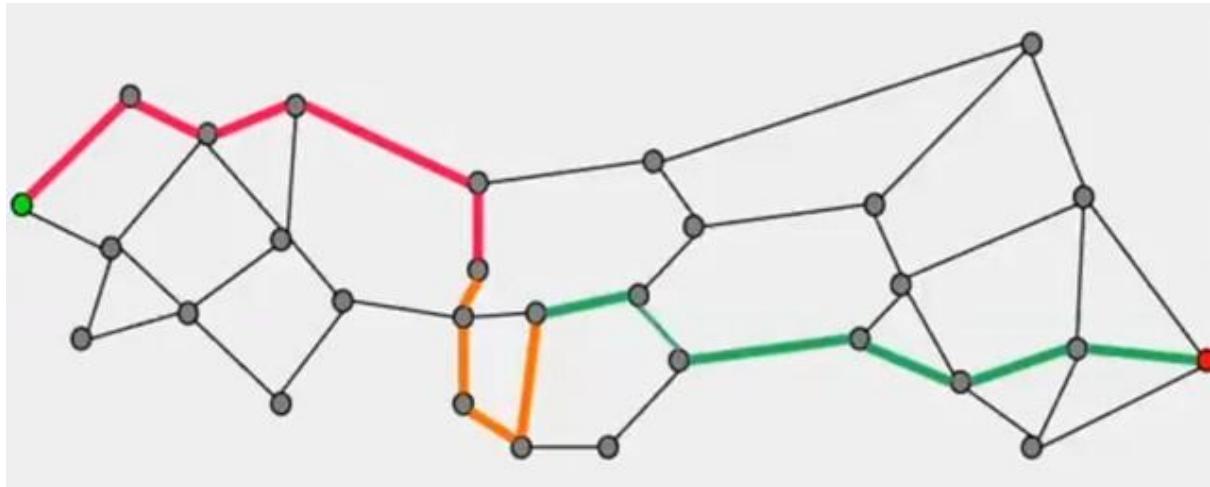


Fig: Using traffic information to determine ETA

On-Trip Component

This component handles the processes which are involved from start to completion of the trip. It keeps track of the locations of the ride when the trip is in progress. It also is responsible for generating the map route (shown in image below) and computing the trip cost when it gets completed.

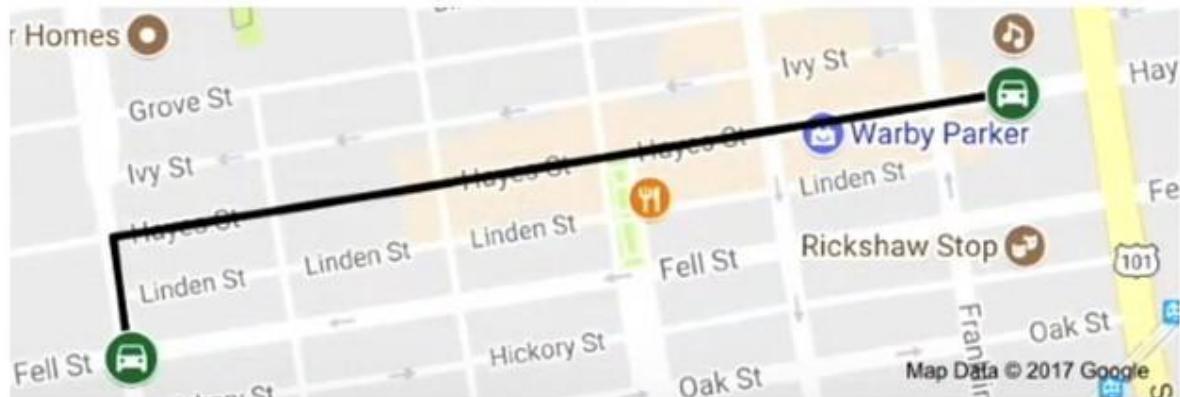


Fig: Sample result of the map route

In the image below, we have shown the sequence of operations which are involved while the trip is in progress and also when it has completed. The driver app publishes GPS locations when the ride is in progress which is consumed by the Trip Recorder through Kafka streams and persisted in the Location Store. Upon trip completion, Map Matcher generates the map route using the raw GPS locations.

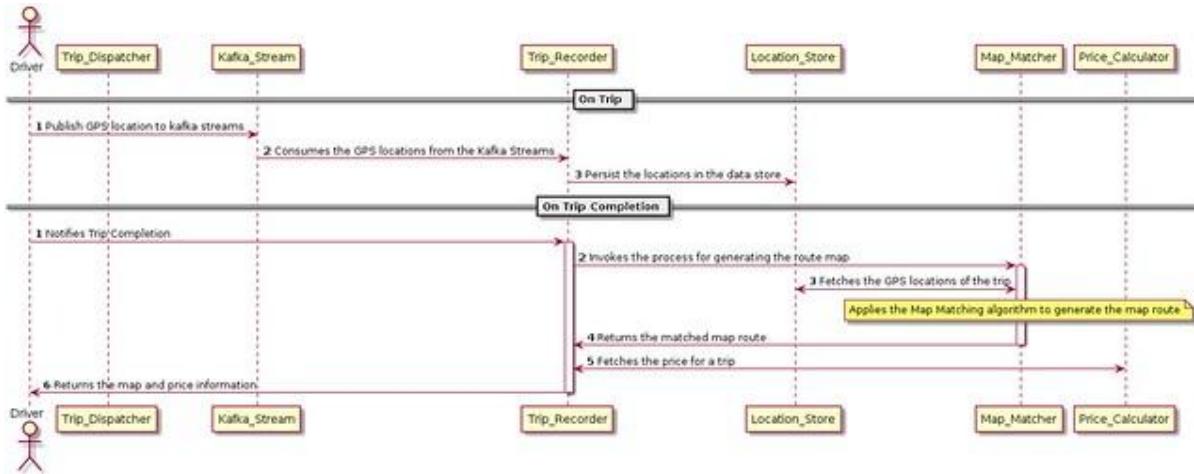


Fig: Sequence Diagram for On Trip Components

Location Store/Data Model

This storage component will be responsible for storing the stream of GPS locations sent by the rider/driver app while the trip is in progress. Some of the characteristics of this store are listed below.

- It should support high volume of writes.
- The storage should be durable.
- It should support timeseries based queries such as the location of driver in a given time range.

We can use a datastore such as Cassandra for persisting the driver locations, thereby, ensuring long-term durability. We can use a combination of as the partition key for storing the driver locations in the data store. This ensures that the driver locations are ordered by timestamp and can support time-based queries. We can store the location point as serialized message in the data store.

Map Matcher

This module takes in raw GPS signals as input and outputs the position on road network. The input GPS signals comprises of latitude, longitude, speed and course. On the other hand, the positions on a road network comprises of latitude (on an actual road), longitude (on an actual road), road segment id, road name, and direction/heading. This is an interesting challenge in itself as the raw GPS signals can be far away from the actual road network. Few such examples are shown in the image below where the box on the top-left corresponds to an urban canyon situation caused by tall buildings and the one in the bottom-right is due to sparse GPS signals. The map route generated by Map Matcher is used for two major use-cases: i) finding the actual position of the driver on the road network ii) calculating the fare prices.

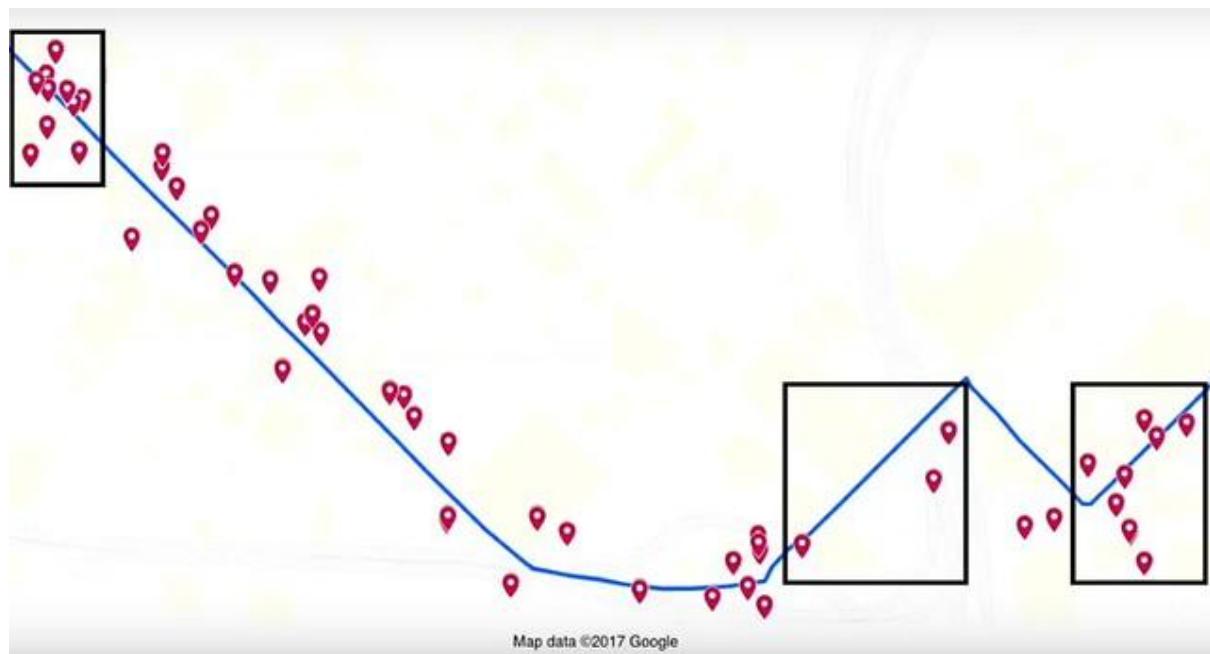


Fig: Challenges of Map Matching: Urban Canyon, Sparse GPS Signals

To solve the Map Matching problem, we need to determine the route which the car would have taken while being on the trip. We can use the approach of Hidden Markov Model (HMM) to find globally optimal route (collection of road segments; i.e. R₁, R₂, R₃, ...) which the car would have taken based on the observed GPS signals (O₁, O₂, O₃, ...). In the image below, we have shown a sample representation of road segments and GPS signals.

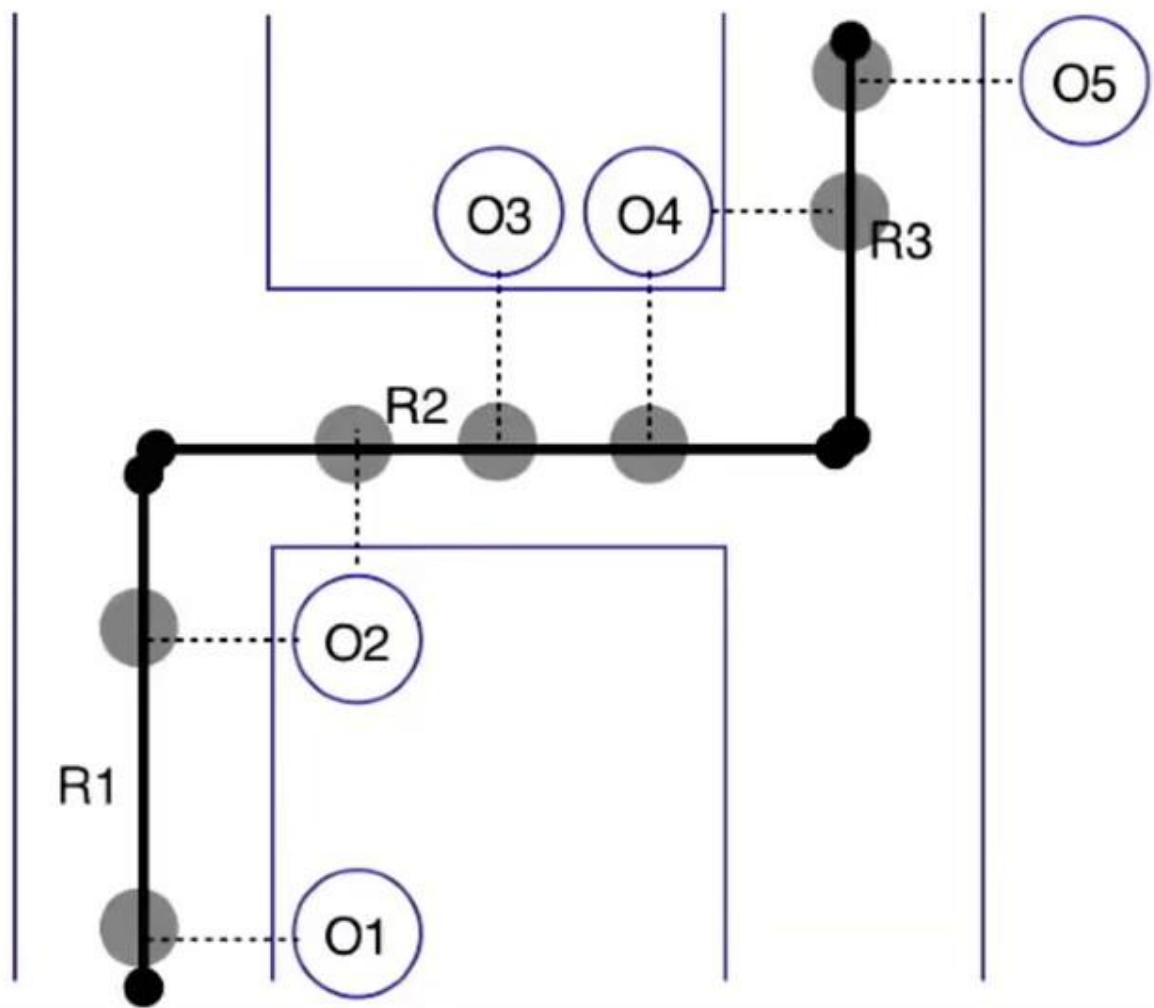


Fig: Sample of GPS signals and road segments

After that, the GPS signals and road segments are represented using Hidden Markov Model (HMM) using Emission Probability (probability of observing a GPS signal from a road segment) and Transition Probability (probability of transitioning from one road segment to another). We have explained this scenario in the image by modeling road segments as hidden states and GPS signals as observations using HMM.

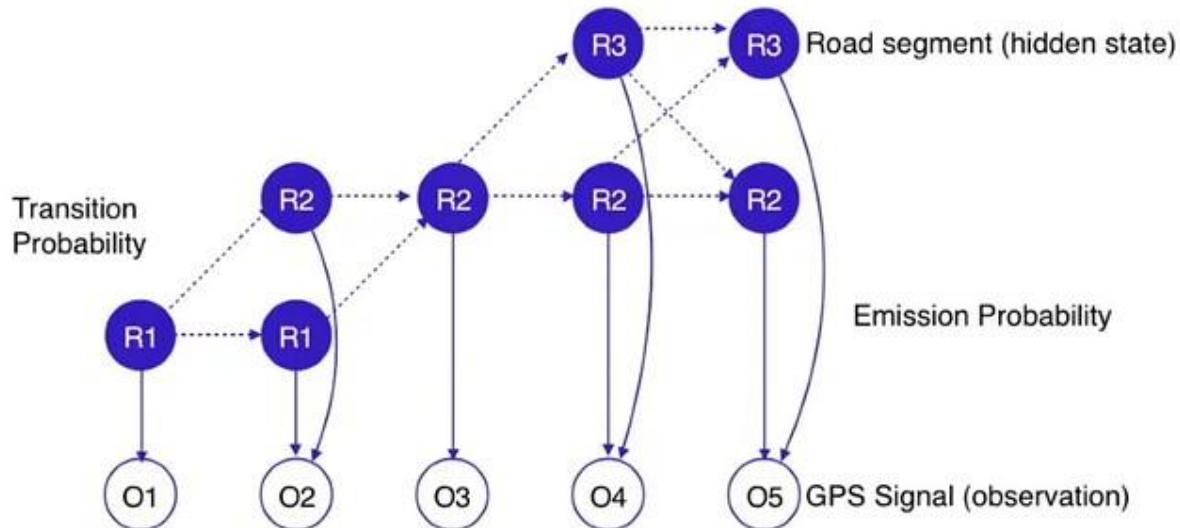


Fig: Representing GPS signals and road segments using HMM

We can then apply dynamic programming-based Viterbi algorithm on an HMM to find the hidden states given a set of observations. We have shown the output of Viterbi algorithm after running it on the HMM we have discussed in this example.

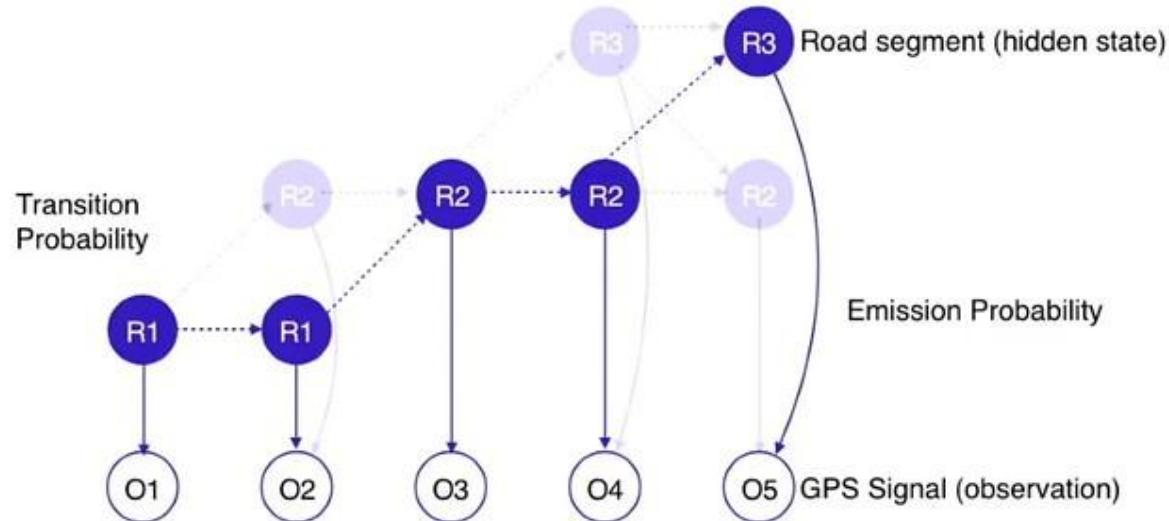


Fig: Output of Viterbi algorithm after running it on the HMM mentioned above

Optimizations

One of the problems with the proposed design is that both the heavy reads and writes are directed to the data store. We can optimize the system by separating the read traffic from the writes by caching the raw driver locations in a distributed cache (Redis). The raw driver locations from the Redis cache is leveraged by MapMatcher to create the matched map and persist it in the data-store.

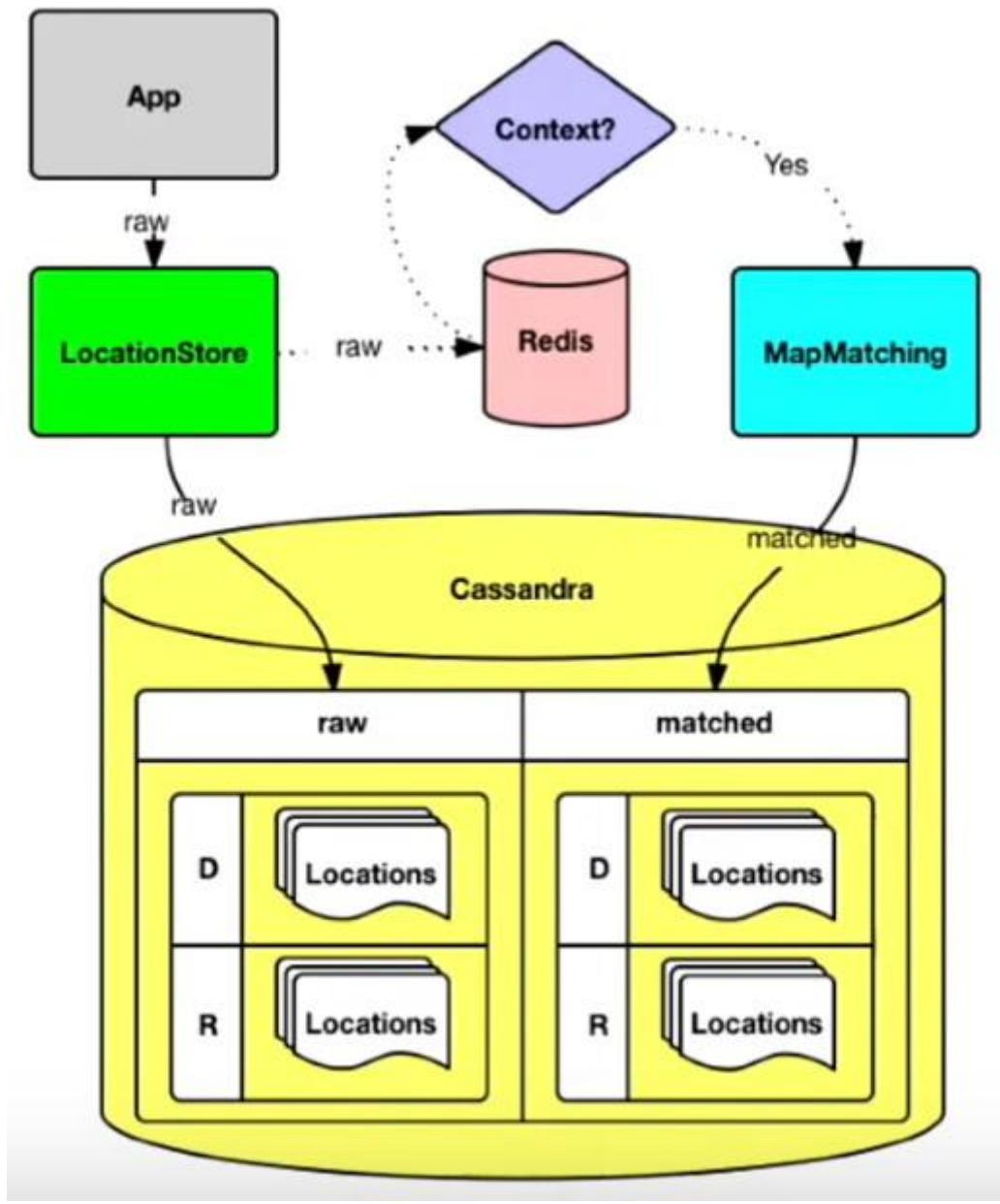
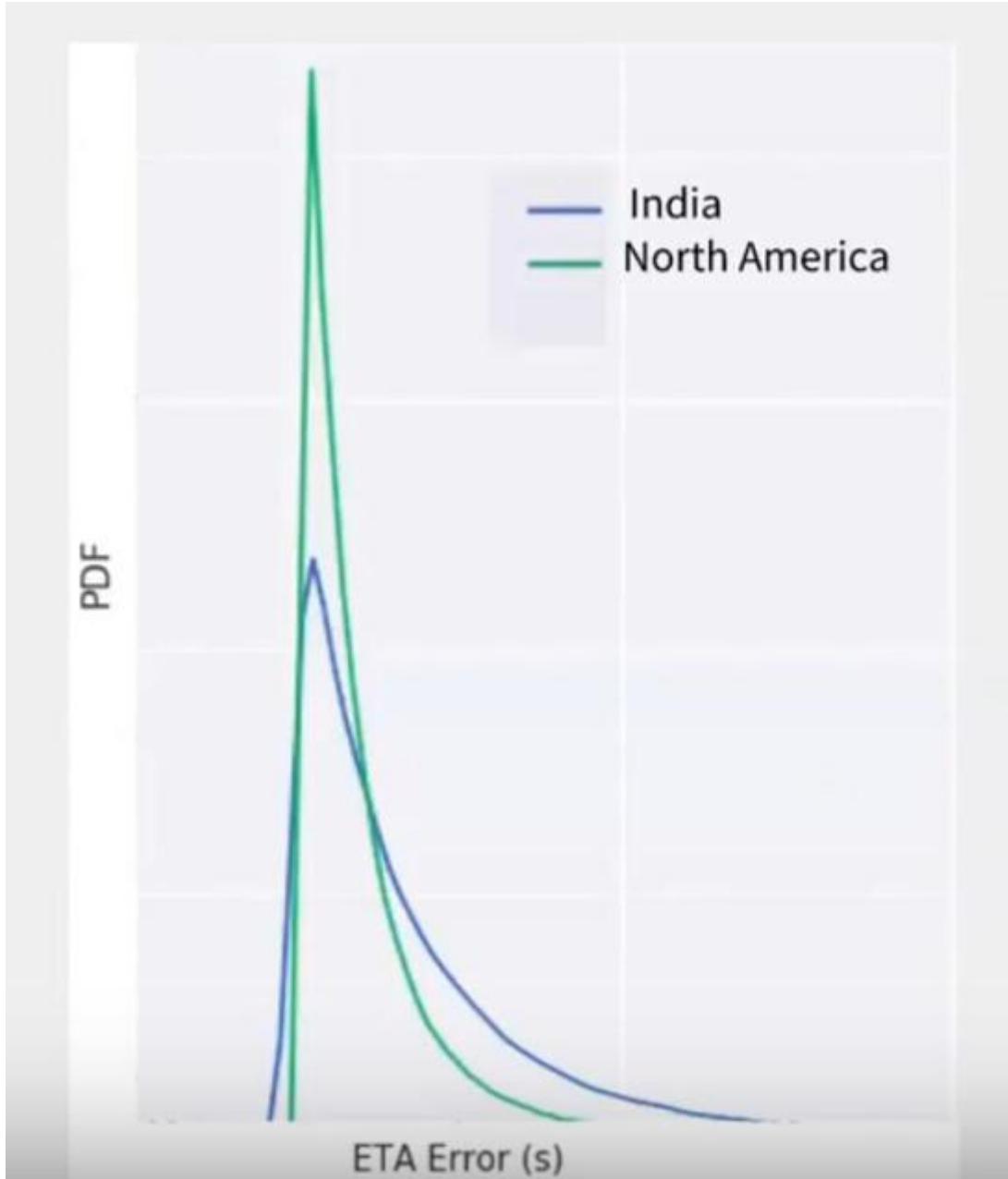


Fig: Optimization to offload the reads from Cassandra to Redis Cache

We can also improve the predicted ETAs by applying machine learning on top of the ETA calculation mechanism mentioned above. The first challenge of applying machine learning involve [feature engineering](#) to come up with features such as region, time, trip type, and driver behavior to predict realistic ETAs. Another interesting challenge would be to select the machine learning model which can be used for predicting the ETAs. For this use-case, we would rely on using a [non-linear](#) (e.g. ETA doesn't linearly relate with hour of the day) and [non-parametric](#) (there is no prior information on interaction between different features and ETA) model such as Random Forest, Neural networks and KNN learning. We can further improve the ETA prediction by monitoring customer issues when the ETA prediction is off.

FUN FACT: In this [talk](#), data scientist Sreeta Goripaty @ Uber, has discussed about the correlation between ETA error(actual ETA – predicted ETA) and region. As part of [Feature](#)

[Engineering](#), the probability density function (PDF) of the ETA error is shown in the graph below. The graph shows that the distribution has thicker tails in India compared to North America. It indicates that the ETA predictions are worse in India, making **region** an important feature for training the machine learning model which is used for ETA prediction



Addressing Bottlenecks:

In the system, each of the components comprise of different micro-services each performing a separate task. We will be having separate web-services for Driver Location Management, Map Matching, ETA calculation and so forth. We can make the system more resilient by having fallback mechanism in-place to account for service failures. Some common fallback mechanisms include using the most recently cached data or even using fallback web-services in some cases. Chaos engineering is a commonly used approach for resiliency testing where failures/disruptions are injected in the system and its performance is monitored to make improvements for making the service more resilient to failures.

In the image below, we have shown the architecture of the Chaos Platform which can be used for introducing disruptions/failures in the system and monitoring those failures on dashboards. The failures are introduced by developers through command line arguments by passing the disruption configurations in a file. The worker environments in the Chaos platform trigger the agents in the hosts to create failures using the configurations passed as input. These workers maintain the disruption logs and events through streaming RPC and persist that information in database.

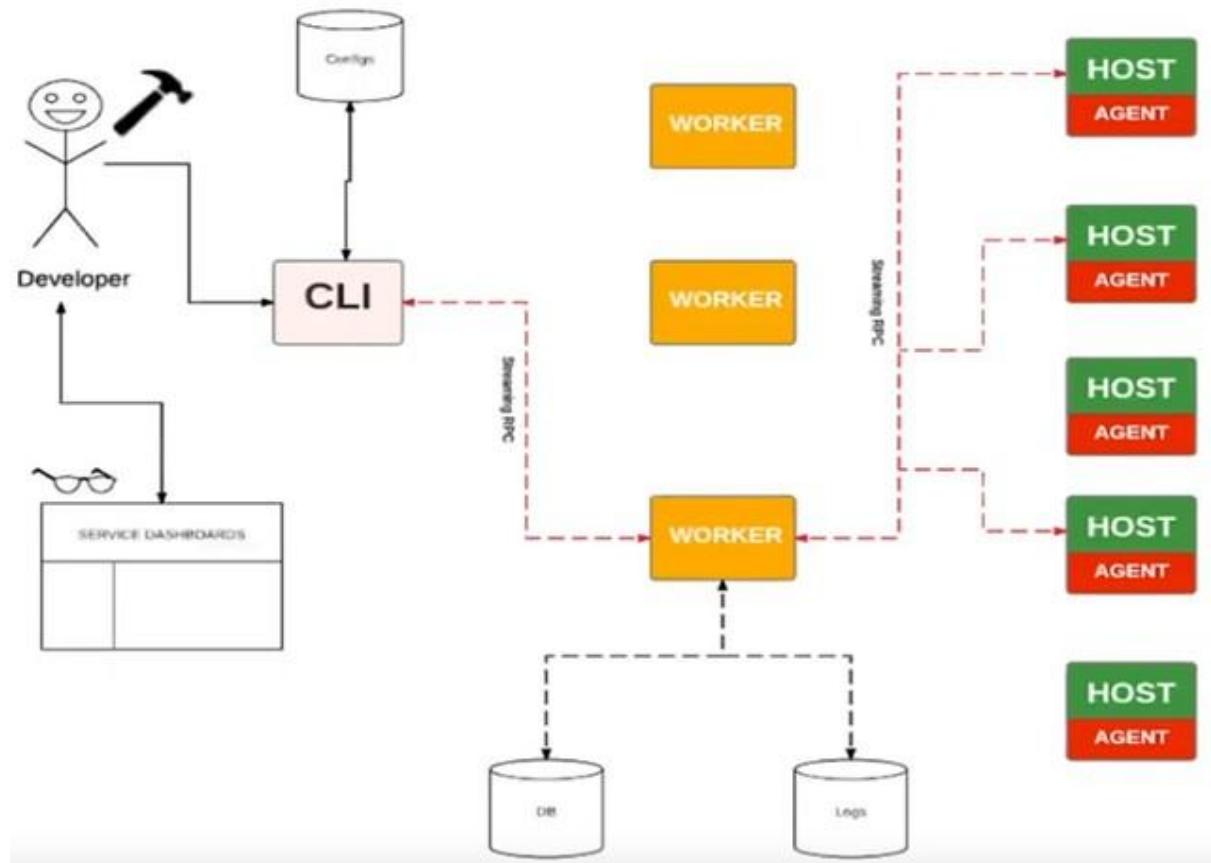
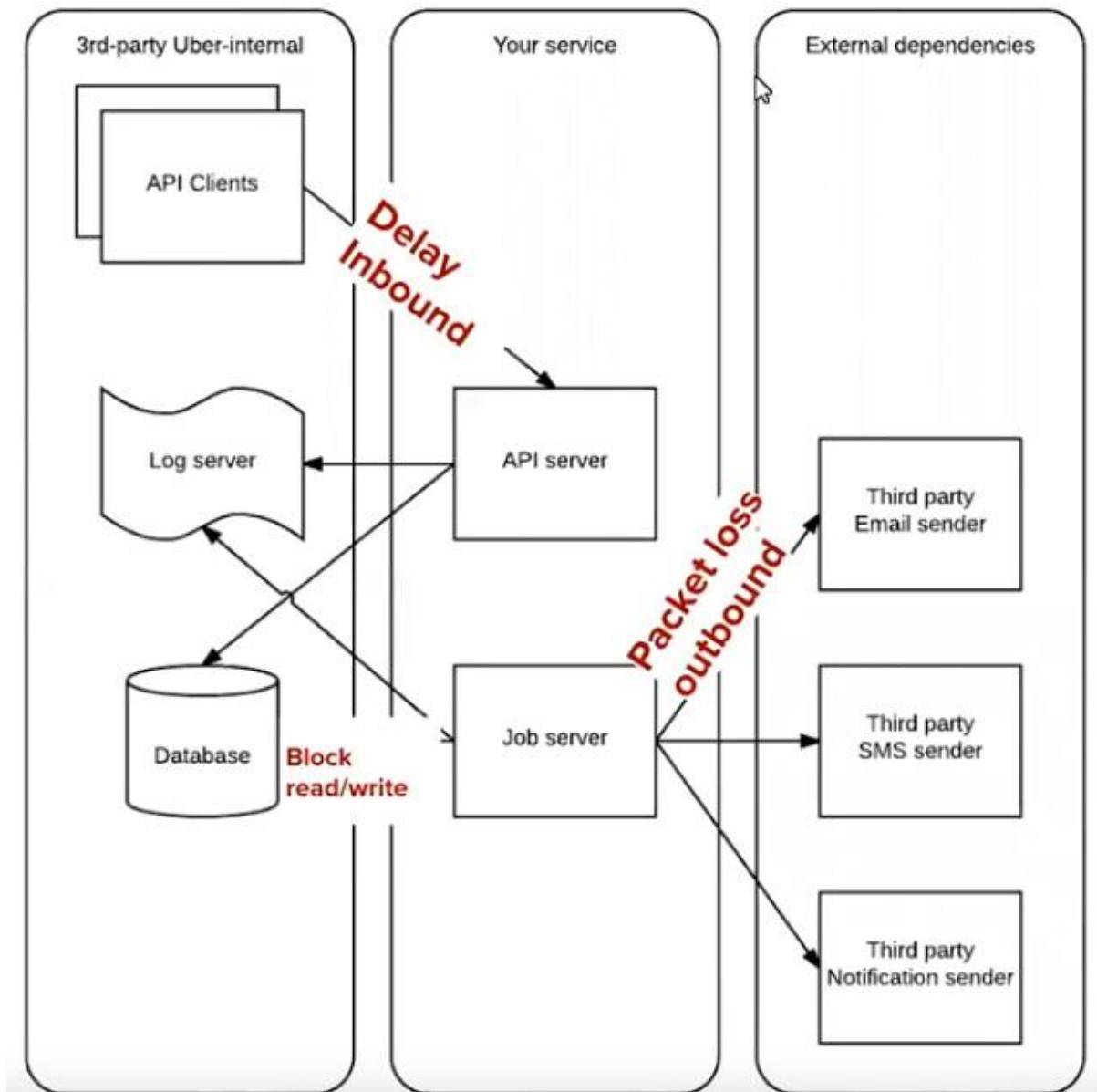


Fig: [Chaos Platform @ Uber](#)

We can introduce two types of disruptions for the purpose of resiliency testing. The first kind of failures are called the process-level failures such as SIGKILL, SIGINT, SIGTERM, CPU throttling, and Memory limiting. The second type of failures are the Network-level failures such as Delays in the inbound network calls, Packet Loss in the outbound calls to the external dependencies, and blocking read/writes to the database. We have shown all the different kind of network failures in the image below.



Extended Requirement:

In the design above, the drivers were being notified of the nearby riders and they were accepting the rides they wanted. As an extension of the existing implementation, the interviewer may ask to extend the system to enable automated matching of drivers to riders. The drivers are matched to riders based on factors such as the distance between drivers and riders, direction in which the driver is moving, traffic and so forth. We can apply the naïve approach of matching riders to the nearest driver. However, this approach doesn't tackle the problem of something called *trip upgrade* which occurs when a rider is matched to a driver and at the same time an additional driver and an additional rider enters the system. The *trip upgrade* scenario is illustrated in the images below.

Matching: Trip Upgrade

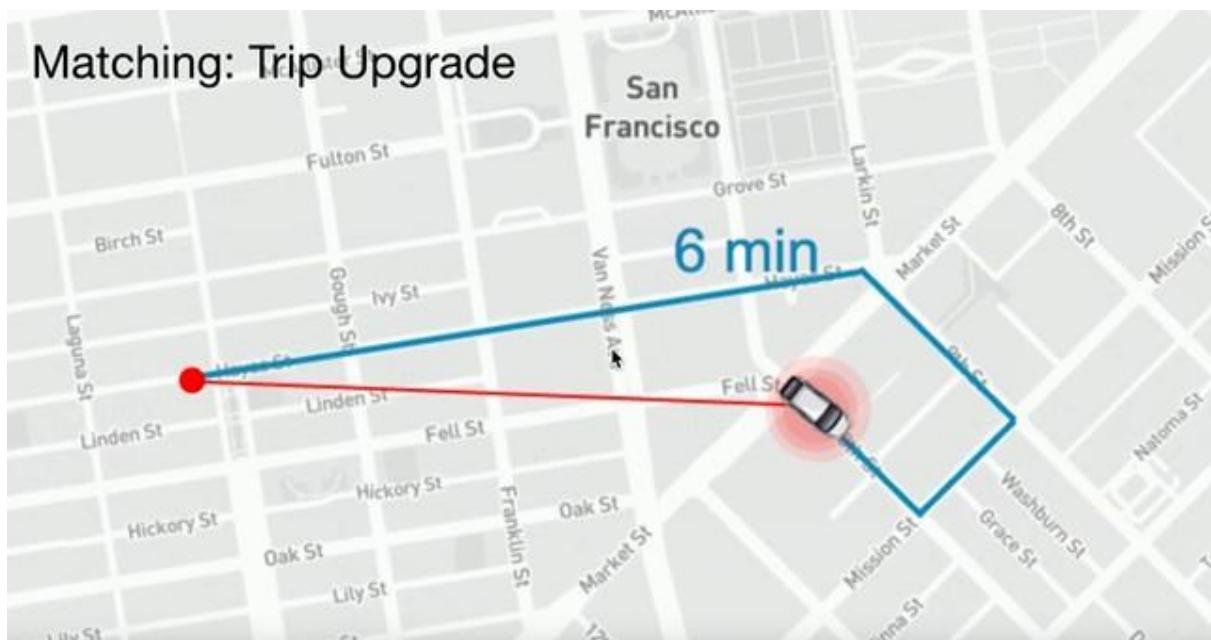


Fig: Trip Upgrade - Rider gets matched to a driver

Matching: Trip Upgrade

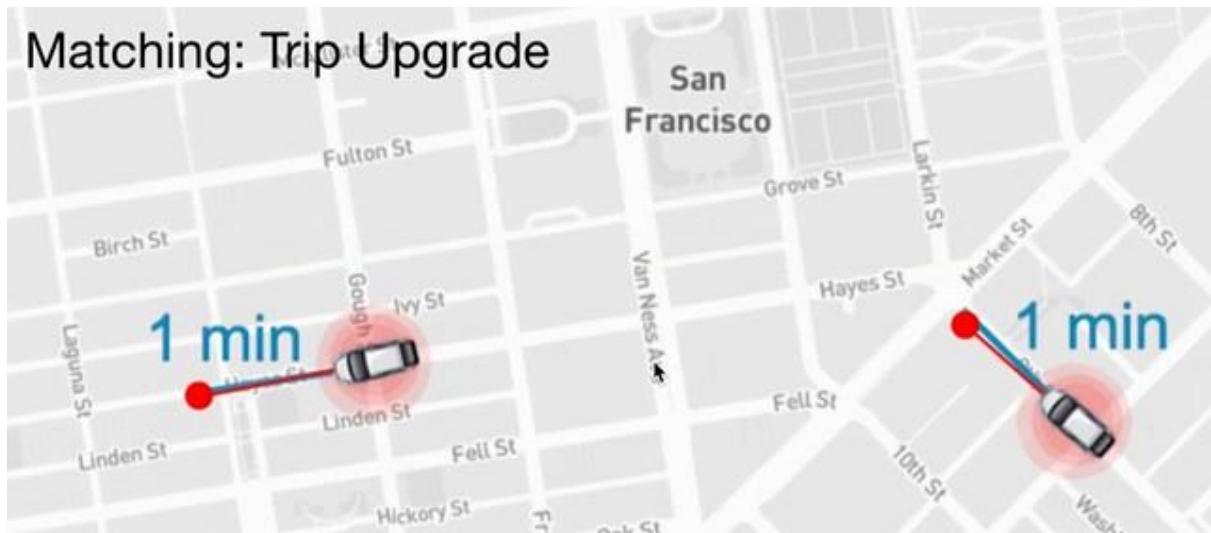


Fig: Trip Upgrade - Additional Driver and Rider enters the system

FUN FACT: Dawn Woodard, Uber's senior data science manager of maps, has talked about the science behind the Matching @Uber in her [talk](#) at Microsoft Research. Dawn has talked about different methods for matching and dynamic pricing in ride-hailing, and discussed machine learning and statistical approaches used to predict key inputs into those algorithms: demand, supply, and travel time in the road network.

4. Fault Tolerance and Replication

What if a Driver Location server or Notification server dies?

We need replicas of these servers, so that the primary can failover to the secondary server. Also we can store data in some persistent storage like SSDs that provide fast IOs; this ensures that if both primary and secondary servers die, we can recover the data from persistent storage.

5. Ranking Drivers

We can rank search results not just by proximity but also by popularity or relevance.

How can we return top rated drivers within a given radius?

Let's assume we keep track of the overall ratings in our database and QuadTree. An aggregated number can represent this popularity in our system.

For example, while searching for the top 10 drivers within a given radius, we can ask each partition of QuadTree to return the top 10 drivers with a maximum rating. The aggregator server can then determine the top 10 drivers among all drivers returned.

Solution 7:

Requirement Gathering:

Functional Requirement:

- There are two types of users for UberApplication:
 1. Drivers
 2. Passengers
- Drivers need to routinely tell the backend service about their present Location and also they have to flag their availability status to pick up travelers. Travelers/Riders get the chance to see all the close by accessible drivers. Users can request a ride; drivers are informed that the user requested a ride and ready to be picked up. When a driver and Rider acknowledge a ride, they can continually check each other's current location until the destination is reached. After arriving at the objective, the driver notifies the Journey complete to be marked as available for further riders.

Account should be there/Registration

- Either you will login to Passenger account as a Passenger
- Either you will login to Driver account as a Driver

Passenger Requirement:

- For passenger, he should be able to set the current location and search for the nearby cabs/drivers
- Requesting for a rider: Passenger should be able to request a ride by setting his start and destination location.
- The ETA for the destination.
- Passengers should be able to see his/her previous trip history.

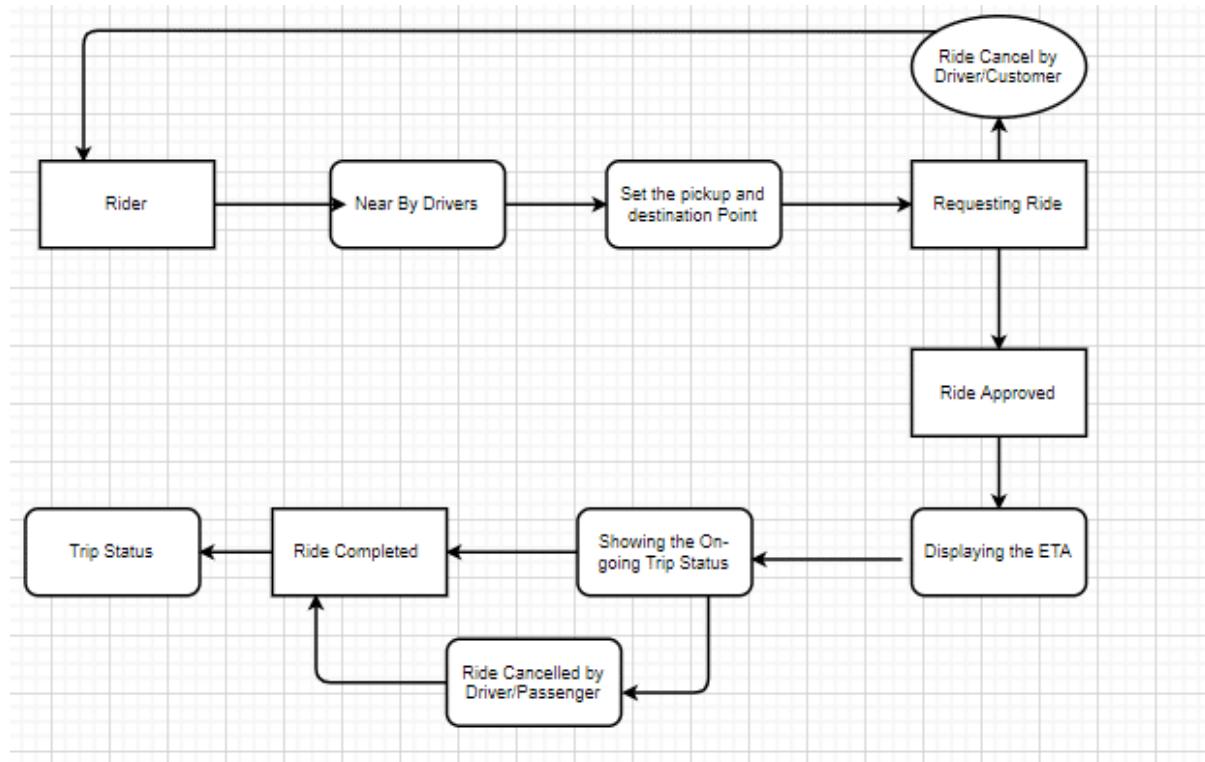
Driver Requirement:

- Driver should inform his availability to the system. The driver should receive a ride request from a nearby passenger.
- After the driver accepts the ride request, the passenger should be able to see the driver details and the ETA of the cab.
- After the driver accepts the ride request from the driver then both driver and passenger should see each other's location
- Driver should mark the trip completion.
- Rating/Feedback feature
- Analytics/Monitoring

Nonfunctional Requirement:

- The service should be highly available.
- Passenger searching time for cab/driver should be defined up to a certain threshold like 60 secs. And after that again the passenger has to again invoke the search request.
- The system should be highly scalable.

Life Cycle of Uber Ride/Trip:



API Specification:

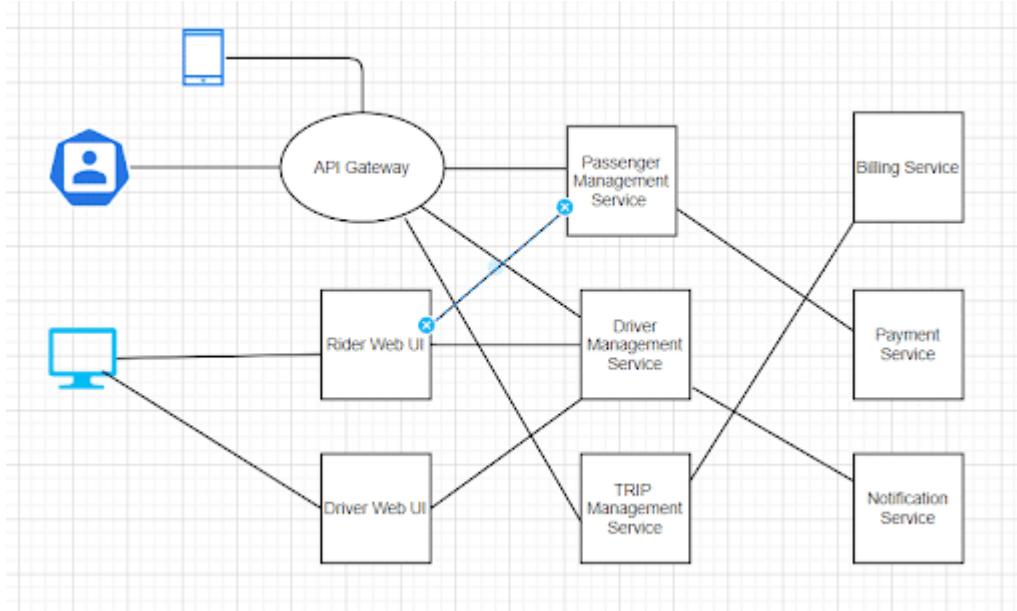
- APIs would be forgetting the Map, Getting the location, finding ETA, Requesting Trip, ConfirmingAvailabilityofDriver, acceptTrip, cancel Trip, CompleteTrip, getTripStatus, update location, rateUser, getTrip History

Capacity Estimation and Constraints:

- Let's expect we have 100M passengers/Riders and 1M drivers with 1M passengers which are active per day and 100K Cab drivers operating per day. How about we accept that all dynamic drivers pass their current location to the backend system in every 5 seconds. When a Passenger places a request for a ride, the framework/system should be capable to connect to the drivers in the real-time

Uber backend Micro Service Architecture Design:

- There are tons of services which are supporting the trip terabytes of data that have been used for this particular trip. When Uber started, they had a monolithic architecture that means they had a back-end service having front-end Java application and a database and a couple of real time [java development services](#) only. This did not work well when Uber started to roll the service into different regions. initially the design was something like they used Python for the application servers, and they use a Python based framework. After 2014 uber architecture has evolved into something like service-oriented architecture now as below:



- From the API gateway, all the internal endpoints are connected. All these are individually deployable units. We can work independently on individual applications without disturbing the other service.
- As soon as the Customer/Passenger opens up the app on his phone, it shows a list of nearby drivers that are hidden, and then it enters his pick-up and drop-off location to request a ride, and then he presses the button request.
- The request is sent to all the nearby drivers who are available at that point and then waits for the driver to accept requests at which point a ride gets dispersed and driver info.is shown on the user's uber app.
- Now two things can happen at this point either driver and user can decide to continue the trip or they can cancel. if they two forward with the ride, Uber tracks the location

of the driver in the background which will help to generate the ride receipt after the trip completion. So, at that point rider gets a charge based on the time. The driver and rider both will see a receipt on their app. Uber is completing millions of rides per day so on another day this sounds pretty amazing, but it could be also pretty embarrassing if something or some system is not efficiently designed.

- I'll try to give a very high-level system and services described here. So, first User bootstrap the Uber app, and the number app calls the trip Dispatcher Microservice which calls the car location index to show all the nearby drivers. In the area, these locations are first restored by another microservice which we call driver location manager and then we have the trip dispatcher called the ERA calculator microservice which obviously helps in calculating the ETA of the driver to appear in front of the rider. Then once this trip starts, we have a trip recorder as the on-ride component that tracks the driver's GPS signals based on the time and the distance of the ride.
- In the end, we have a Map matcher and the price calculator as the post-trip component that takes in the input from the trip recorder to actually generate the release feed and send it to both passengers and drivers.
- So, now as we have a complete picture of Uber Microservices architecture at least from the Passenger booking point. Now the question is how the Cab location index supports this high volume of reads and writes coming from both riders and drivers in the uber app so as I have mentioned that it is a distributed cache where the car locations are stored in memory. But how does it do it?
- It is done by selecting the right kind of Database Sharding Strategy to avoid problems like hot partitions so we can employ distributed cache to balance the load of reads and writes and we can store it in memory to make those queries really faster since Passenger does not have to wait five minutes to find the best cab but the main thing that we need to take care of hot partitions.

System Designing of Uber System:

- Here is the complete architecture for uber or any taxi aggregation platform and you see everything here I have written all the major components over here but instead of jumping right into explaining each and every component I am going to concentrate much on this particular component over here i.e. Dispatch optimization.

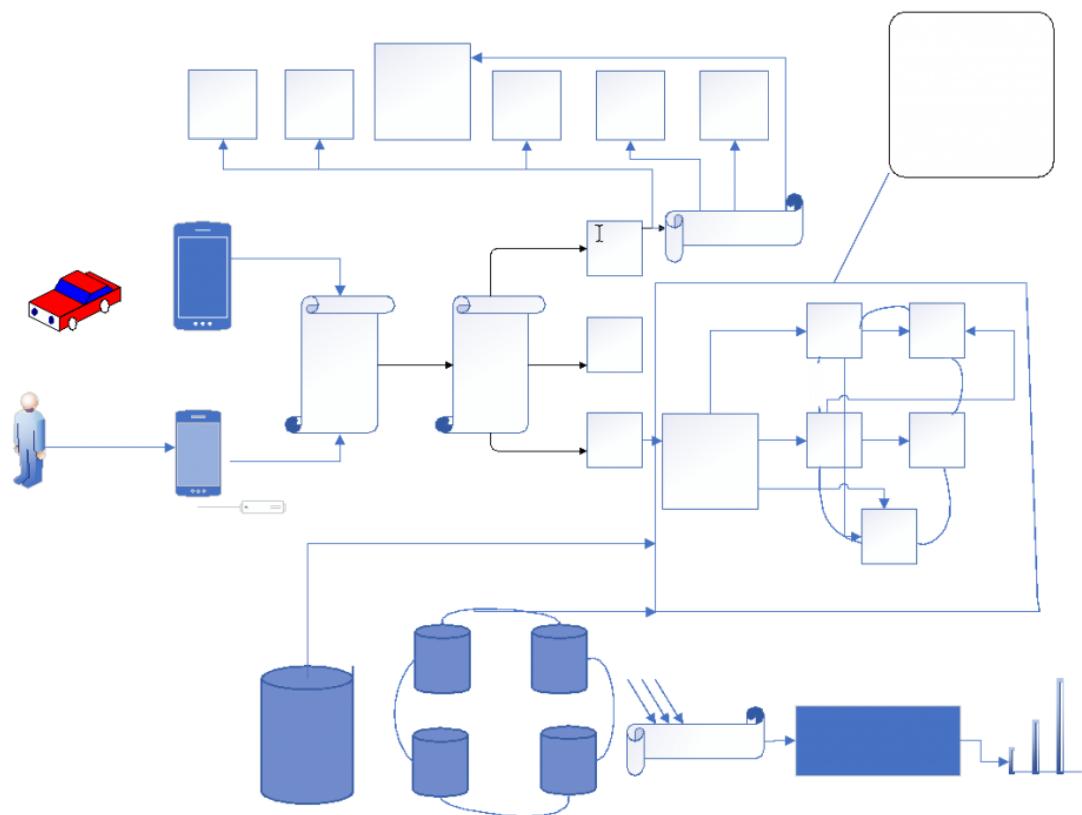
How Dispatch system works completely on map our location data

- We have to model our Maps and our location data properly. Since the earth is spherical, it is pretty hard to do summarization and approximation just by using latitude and longitude data, and to solve this particular problem what Uber uses is Google s2 library. This library takes the spherical map data and it makes and widens the data into tiny cells of 1KM* 1KM cells(For Example I have mentioned 1kn). So, when we join all these cells we get the complete map.

- So each cell is being given a unique ID and this is really helping now to spread this data in the distributed system and store it easily. So, whenever we want to access a particular cell from the Cell ID, then we can easily go to the server where that particular data is present. Here, we can use consistent hashing based on the cell ID.
- Google S2 library can easily give you the coverage for any given shape. For example, if we want to draw a circle on the map and we want to figure out all the producers/Cabs available inside that particular circle. Then in this case, what we need to do is use this S2 library and will feed the radius as an input to this, and then it will automatically filter out all the cells which contribute to that particular circle/area.
- So, in this way we get to know all the cell IDs and from this we can easily filter the data which belong to that particular cell and this will help us to tell us the list of Cabs available in that area/circle. Also, we can calculate ETA from that.
- So, when we want to match a rider/Passenger to the driver or even if you want to show a number of cars which are available in your region then in that case, what we need to do is just need to draw a circle of about two-kilometer radius and list out all the cabs available using Google S2 library. Then we need heck for the ETA.

What is the role of the ETA and how can it be checked:

- For example, within a particular circle of about a two-kilometer radius, suppose 4-5 cabs are available. Here, we have to calculate ETA or the distance from the Passenger.
- Consider the passenger is present in a particular location in the center of that circle. So, here the shortest distance obviously we can calculate using any algorithm, but this won't accurately give you the ETA because cab driver can just drive the cab from his point to the rider's place directly but instead, he has to go to the connected roads. We have to find the ETA or the distance which is connected by the road and we have to figure out all the Cabs' ETAs.
- So, considering the road path and condition then actual ETAs can be calculated. Accordingly, we can filter out which are the cabs suitable for this particular passenger and then we can send the notification to the driver.



Explanation of the Functionality of Each Component:

While jumping into the system design, first understand all the components which are needed to understand the Producer/Consumer(Supply/Demand) part and the dispatch optimization component.

WAF & Load balancer:

- You can see over here in the above image, the cabs are the producers/Supply and Users are the consumers/Demand. In every 5 seconds, the Cavs will be kept on sending the location data to the Kafka REST API and every call happens through the Web Application Firewall and then it hits the Nginx load balancer(LB) and it goes to Kafka where it keeps updating the location by pushing it to Kafka and then the data is consumed by various components of Big data.'
- A copy of the location is also sent to the database and also to the dispatch optimization to keep the updated location of the cab. We need a Web Application Firewall for security purposes. we can block the request from the blocked IPs and from the bots and also, we can block the request from the regions where Uber is still not yet launched.
- The next is the Load Balancer. We obviously need a load balancer that can have different layers within like layer 3, layer4, and layer 7. Layer 3 and 4 based on their based load balancer like all the ipv4 traffic go to this particular server or the ipv6 traffic

go to a different kind of server. In Layer 4, we can do DNS based Load balancing. And in layer 7, it is application-level Load balancing.

Kafka:

Kafka rest APIs will provide the endpoint to consume all the location data for every Cab. If for example we have thousands of Cabs running for the city and in every 4 seconds, cabs are sending their location data, that means in every four seconds, Uber backend system will be having thousand hits or thousand location point being sent to the Kafka API.

And that data will be buffered and put it to a Kafka and then they were consumed two different confidence of Big data layer and then a copy of it will be saved to NoSQL DB. When the Ride/trip is happening, and the latest location will be sent to the dispatcher component to keep the state machine updated.

Web Socket:

We also have other rest APIs and I will explain these components also. Here the important component is WebSocket and why do we need WebSocket?

Unless like normal HTTP requests, WebSocket is really helpful in these kinds of applications because we need an asynchronous way of sending messages from client to the server and server to the client at any given part of the time. That means that we should have a Connection established between the Cab Service to the server or from the user service to the server. What happens is web WebSocket keeps the connection open to all of the applications and based on changes happens in the dispatch system or any component in the server, then the data will be exchanged to and fro between the application and the server.

Since the Producer-Consumer(Supply-Demand) and the WebSocket component of are mainly written in node.js as NodeJS is really good in asynchronous messaging and small messaging and also it is an event-driven framework.

DISPATCH optimization

So, now I am explaining how the dispatch system works the uber dispatch system is built using node.js. The advantage of using node.js either asynchronously event-driven framework. So, the server can push the message or send a message to the application whenever it wants.

Now the next question is how do we scale these servers inside the DISCO component? To scale this, uber uses something called repo it has below functionalities

- it does consistent hashing to distribute the work between these servers.
- it also uses an RPC call to make calls from one server to one more server at times.
- it also uses something called “Gossip” protocol which helps every server knows the other servers’ responsibility. Every server has a specific responsibility even though they all do the same work but the responsibility to compute for a specific location is assigned to each and every server.

Why do we need in Gossip protocol?

- So, the advantage of gossip protocol is we can easily add a server and remove the server. That way when we add the server the responsibility is distributed to the new server and responsibility is reduced for other servers that way rest of the servers also knows that this new server is responsible for doing what works.
- In real-time when a passenger places a request for a cab or for a trip, then how this Uber system functions, I am going to explain that. So, you know that WebSocket has a connection to the passenger and to the Cab Service, so when the user places the request for a ride, then the request first lands to the web socket and this web socket hands over the request to the producer/Consumer(Supply/Demand) service.
- Since the Consumerserviceknows the requirement of the Cabaret offering riders. For Example, I need A uber mini car or I need a Sedan car something like that. Now the Consumer service requests the supply/producer that I need a Cab of this kind at a particular location. Here, what the Producer service does is: it knows the location of the Passenger that means the cellID of the user's location on a map as I have already mentioned that Google S2 library gives breaks the alert into tiny sizes mean it breaks location into small
- If the user is present in a particular cell, that means it knows the cell ID of that particular user. The consumer service supplies the Cell ID to the Producer/Supply Service. So based on this, what it does is it contacts one of the servers in the ring of the servers. In the consistent hashing, you know the responsibility is equally distributed that means suppose we have about 10 cells in total just for example and dividing the responsibility often cells within the available servers.
- Suppose the user is requesting from cell 9 that means the Producer knows the five and hits the server and requests the server to find a Cab for that passenger present in cell 9. Now what the server does is it figures out and draws a circle in the map within some radius and then it figures out all the cells from which the Cabs can be figured out. Then it makes the list of all the cabs and from that the Cab list, it figures out the ETA for each and every cap using the rest service called Maps ETA service and it sorts based on that.
- With all this information it gives back to the Supply/Producer service and the producer service using the WebSocket sends the request to the first few Cabs which is very near to the passenger as soon as the driver accepts whoever accept first that particular cab will be assigned to that Passenger.
- So sometimes what happens is the Producer/Supply service won't directly talk to each and every server. It just passes the requests to one of the servers and then that server internallyhands overs the requests through all the different other servers which is responsible to compute or to figure out the Cabs. The requests are passed from one server to other using RPC call.

- Once all the servers figure out the Cabs' ETA then they all respond back to the supply/Producer service and the producer service takes care of notifying the driver and matching the demand with the supply

Ring pop

- Suppose, we need to add more service to the existing dispatch optimization system. The reason is we need to handle the traffic from the newly added city. For example, we have added two different servers into the Ringpop. Now the responsibility of these servers is unknown. So, what the ring pop does is it knows all newly added cell IDs from the MAPS-ETA component and it distributes the responsibility of newly added cells to these new servers.
- Exactly the same way it works when we take down the server and then it reshuffles the cell IDs or-assigning the responsibility of the computation of a particular cell to one of the random servers which is free .

GEOSPATIAL DESIGN:

- As I have already mentioned about the Google S2 library used to break the map into different cells and that are used to easily locate the Cabs near to any particular Passenger's location so that is the use of S2 libraries.
- Now, how to build maps or to use maps in your application? EarlierUber used to use Map box because of Google's pricing strategy and etc. but now Uber is heavily using Google's map framework in which uses GoogleMaps to show on Mobile apps and also uses google's Maps API to calculate the ETA from point A to point B that is from pick-up point to the destination point. Earlier Uber used to do all this thing on its own and it also earlier used to repeatedly trace the Cab's movements, GPS points and to build the road network system on its own by Calculating the ETA on its own. But nowUber has moved on and it is currently using Google's library heavily.

Finding the Preferred Access Points:

- The preferred access point is the place where the passengers would have booked the cab from the nearby location of that place. Now the question comes to your mind that how did it learn it? It actually learns based on the repeatedly Uber drivers or the Cabs used to stop near the entry/exit gates or near any shops/malls or universities.
- Because they can't enter the campus so that is been learned by uber and it automatically shows to the customers that The Cab can only pick up from those specific points. Uber uses different algorithms and machine learning to keep automatically figure out these Preferred access points.

How ETAs are Calculated:

- Now I will explain a little bit about how ETAs are calculated and why it is a very important component of Uber or any Cab Provider service. Suppose, for example, a Passenger is requesting a cab from a particular point and the number of available

Cabs near the passengers says 4 cars. So, when a user requests a Cab, the Consumer service requests the Provider/Producer service to figure out carbs for the passenger.

- Now what the service does is it tries to figure out all the cabs which are nearby to this particular passenger, so now we trust it draws the circle and then it figures out there are 4 Cabs which are free to take the service but what happens now is it calculates ETA from the Passenger's place to those Cabs' place by the road system and then figures out the ETA for all the four different Cabs. But this approach always does network because these could lead to bigger ETAs.
- For example, if one more 5th cab which is about to finish another ride and that car is very near and the trip is about to end so this is obviously a better selection than any of those other 4 cabs as the trip is about to complete in about few minutes and this 5th Cab is much nearer to the rider. So, Uber includes all the different factors like turn cost, turn cost, the traffic condition, and everything to calculate the ETA and based on the different ETA sometimes not just the idle Cabs, but the other Cabs which are already serving the trip also included for the Cab assignment to the Passenger.

ETA Calculation(Alternate Way)

For ETA calculation, we can divide this into two major components:

- First is how to compute the route from the passenger's pickup point to the destination with the least cost
- Second is the estimated time taken to traverse the route.

The main thing to imagine here is to convert any map into a geographical representation so where each intersection is a vertex and each road segment connecting two vertices is an edge. Then we can employ any routing algorithm like Distance Vector or Link state to find the shortest path between source and destination. But as we also know that to find the shortest distance it will take us $O(n \log n)$ run time complexity so this means the larger the end the longer it will take us to find out the shortest distance between two points.

Here, what we can do is to partition the entire graph and precompute the best path within those partitions which significantly brings out the runtime complexity. Once we have the most optimal path to go from point X to point Y, we can use the current traffic information to calculate the ETA which can be influenced by the time of the day or where the conditions or any concerts of gatherings happening at that particular moment which will be influencing the time. It will take for the driver to get from point X to point Y just like by adding the weights on these edges based on that traffic condition.

Now Passenger is happy as his ride is broken down and he got an ETA for the driver to arrive so now let's dive deeper into the on-trip component. This is the part where the driver presses the "start ride" button and the back end system is tracking the trip's current location. So, we have a message broker that listens to all the events sent by the drivers through Uber app. We can use any message broker service here like Apache Kafka, EWS kinases, Microsoft event Hub, Google pub/sub etc. We can use Kafka streams here for our

design because it is open source and cost-effective as it doesn't charge on the number of messages in the channel.

Once the trip location is sent to Kafka stream, then it gets forwarded to the Trip recorder service which takes it and writes it on to a location store (Any NoSQL DB like Mongo DB). We can use Cassandra as it is highly scalable as you can keep on adding more machine nodes and keep on storing data. Since we don't want to lose trip details because of the hosts' failure.

Below are the various Machine Learning Models using which we can optimize the ETA calculation process which will give near accurate ETA.

Database Design :

Coming to the database part, actually, the earlier users used to use a DBMS that is the Postgresql database for operations. They used to save driver/passenger profile information, they used to save GPS points for everything in a DBMS. It couldn't scale as Uber rolled out service in different cities and then they talked about a new NoSQL kind of database that is built on top of MySQL and it is called schema-less.

When they are building this database below are the points they considered:

- The first one is it should be horizontally scalable i.e. you can linearly add the capacities in a different part of the cities into the network. For example, there are multiple nodes in different regions/cities that are added, and all together acts as one database that is schema-less. So, if you don't want to design then you can either use a big table or Cassandra, MongoDB or any of the NoSQL DB. since they also behave in the same way.
- The second consideration which they considered while building the scheme was a delight and read availability. As I have already mentioned that in every four seconds the Cabs will be sending the GPS location to the Kafka rest API and those points were sent to Kafka for different processing and also points are written to NoSQL DB. for record purposes. And also, points are sent to state machines. It means that it is a Write Heavy application.
- Also, when User requests a cab, all the latest Cabs information is also fetched from the DB to show to the customer on the application. That means that there are tons of Reading happening and also there are tons of Writes happening to the system. So, the system should be heavily Writable and readable, and this system should never give downtime because we haven't ever heard about Uber's downtime even for a minute. Because every minute people will be requesting Cabs and so, we can't just say that we are doing some maintenance. Hence, the system should be always available no matter what you are doing.
- What Uber does is when they launch [Limo services](#) in new cities, they try to build the data center near to the city to give a seamless service. If not, always the nearest data center is selected and the data will be served from those locations.

Use of Analytics in Uber System:

- Now, coming to the Analytics part, what is analytics? In simple words, it is just making sense of the data which we have. Uber does that a lot because they need to understand the user, and they need to understand the behaviors of the Cab Drivers. That's why you can optimize the system and eventually we can minimize the cost of operation and also make the user satisfaction better.
- So now let's discuss what are the different tools which Uber users or different frameworks that Uber Utilizes to do a lot of analytics. As I have already mentioned, there is tons of GPS data flowing into the system from drivers and also a lot of information coming in from the users/passengers. All those data are saved either in NoSQL or our DBMS system or to the HDFS directly. If we are not sending the data directly to the HDFS, then what we can do is dump all the data from the NoSQL DB and put it onto HDFS.
- And sometimes, for a different kind of analysis, we may need the data in real-time and this data can directly be consumed from Kafka. Let me brief about every component of Big Data that you see in the main architectural diagram.

Big Data module(HDFS, HIVE, MAPS)

- The Hadoop platform has a lot of analytics-related tools that we can make use of to build analytics on the existing data. So we can take a constant dump of the data which we have in the database to the HDFS and on top of that, we can use tools like HIVE and PIG to get the data that we want from the HDFS.
- The next component is Maps or ETAComponents. What we can do with this is, you consume the historical data along with real-time data, and also we can retrace the previous Maps data that we have, and then we can build the new Maps all together we can improve the Maps data that is currently there in the Map. And also, with the historical data and the real-time information coming from the Cab like traffic conditions and the speed of the Cabs, etc.
- We can use this data to compute ETA. When there is a request for Cabs, then the server contacts contact these Big Data components for ETA calculation. Now Uber is using something like simulated artificial intelligence to calculate the ETA with more accuracy.

Challenges need to be Considered:

In a Slow network or intermittent disconnecting network scenario, how will the passenger be able to use the Uber app uninterrupted?

In the middle of the trip, if the network gets disconnected then in that case how to handle the billing for that passenger?

10. Designing an API Rate Limiter

Design service or tool that monitors the number of requests per window of time a service agrees to allow. If the number of requests exceeds the rate limit blocks all the excess calls.

Things to analyze and discuss:

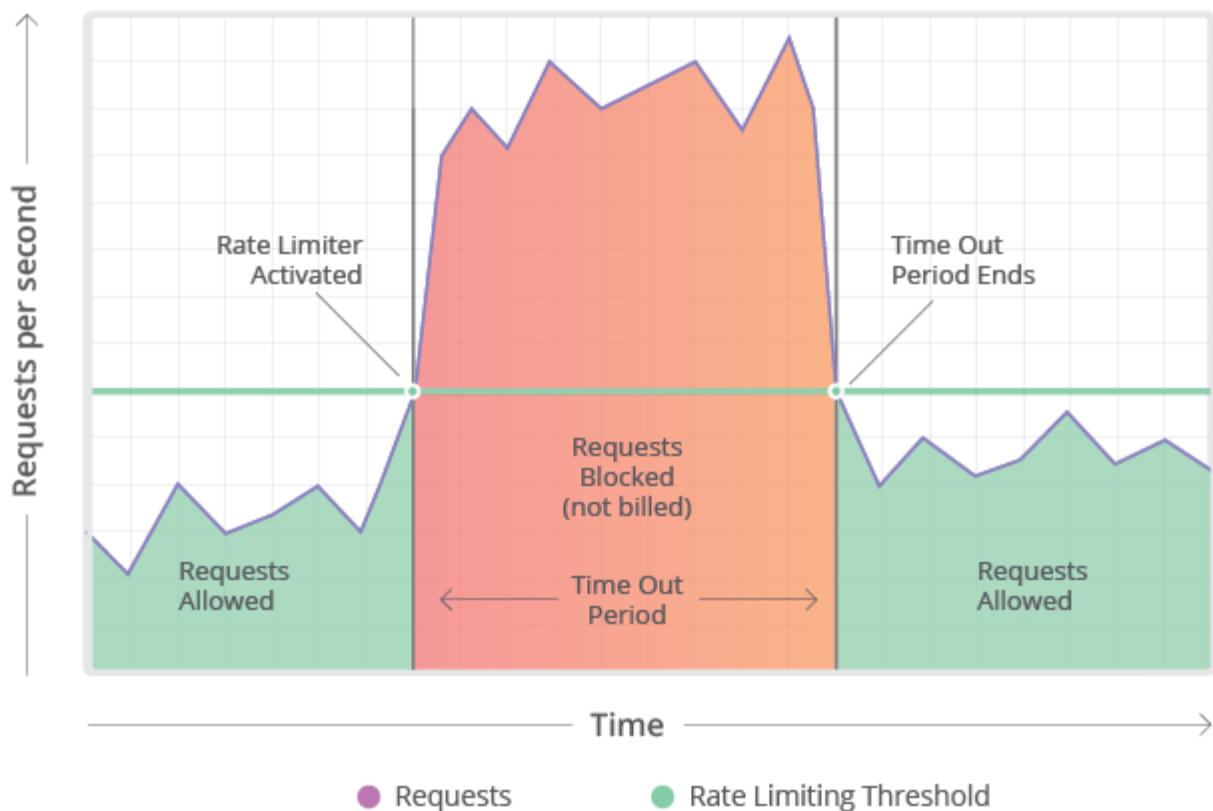
- Limiting the number of requests an entity can send to an API within a time window, for example, twenty requests per second.
- Rate limiting should work for a distributed setup, as the APIs are available through a group of servers.
- How to handle throttling (soft and hard throttling etc.).

Solution 1:

What is Rate Limiter?

Rate limiting refers to preventing the frequency of an operation from exceeding a defined limit. In large-scale systems, rate limiting is commonly used to protect underlying services and resources. Rate limiting is generally used as a defensive mechanism in distributed systems, so that shared resources can maintain availability.

Rate limiting protects your APIs from unintended or malicious overuse by limiting the number of requests that can reach your API in a given period of time. Without rate limiting any user can bombard your server with requests leading to spikes that starves other users.



Rate limiting at work

Why Rate limiting?

1. Preventing Resource Starvation: The most common reason for rate limiting is to improve the availability of API-based services by avoiding resource starvation. Load based denial of service (DoS) attacks can be prevented if rate limiting is applied. Other users are not starved even when one user bombards the API with loads of requests.
2. Security: Rate limiting prevents brute forcing of security intensive functionalities like login, promo code etc. Number of requests to these features is limited on a user level so brute force algorithms don't work in these scenarios.
3. Preventing Operational Costs: In case of auto-scaling resources on a pay per use model, Rate Limiting helps in controlling operational costs by putting a virtual cap on scaling of resources. Resources might scale out of proportion leading to exponential bills if rate limiting is not employed.

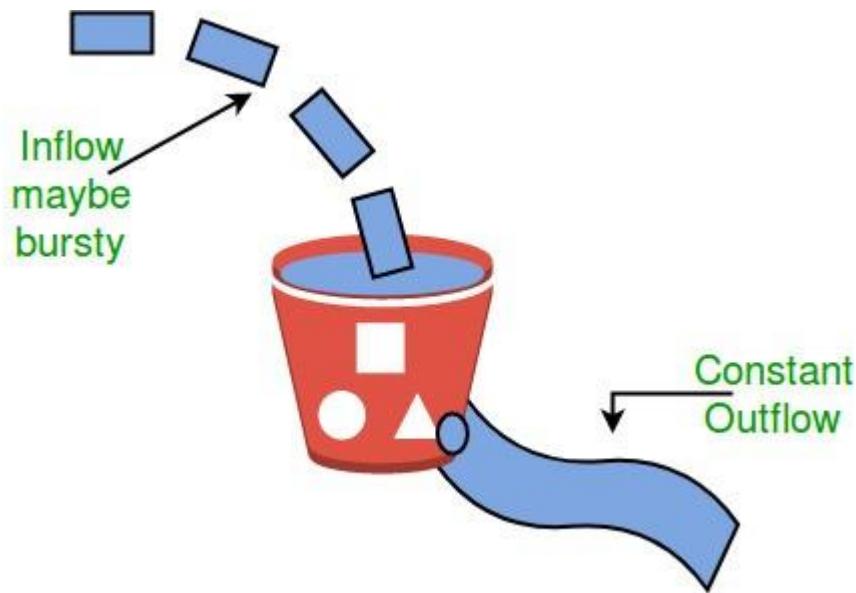
Rate Limiting Strategies

Rate limiting can be applied on the following parameters:

1. User: A limit is applied on the number of requests allowed for a user in a given period of time. User based rate limiting is one of the most common & intuitive forms of rate limiting.
2. Concurrency: Here the limit is applied on the number of parallel sessions that can be allowed for a user in a given timeframe. A limit on the number of parallel connections helps mitigate DDOS attacks as well.
3. Location/ID: This helps in running location based or demography centric campaigns. Requests not from the target demography can be rate limited so as to increase availability in the target regions
4. Server: Server based rate limiting is a niche strategy. This is employed generally when specific servers need most of the requests, i.e. servers are strongly coupled to specific functions

Rate Limiting Algorithms

1. **Leaky Bucket:** Leaky Bucket is a simple intuitive algorithm. It creates a queue with a finite capacity. All requests in a given time frame beyond the capacity of the queue are spilled off.
The advantage of this algorithm is that it smoothens out bursts of requests and processes them at a constant rate. It's also easy to implement on a load balancer and is memory efficient for each user. A constant near uniform flow is maintained to the server irrespective of the number of requests.



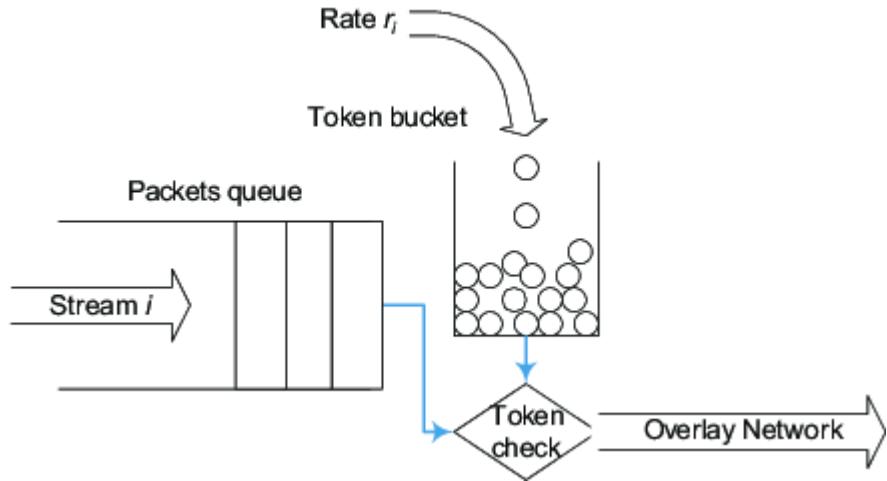
Leaky Bucket

The downside of this algorithm is that a burst of requests can fill up the bucket leading to starving of new requests. It also provides no guarantee that requests get completed in a given amount of time.

2. **Token Bucket:** Token Bucket is similar to leaky bucket. Here we assign tokens on a user level. For a given time duration d , the number of request r packets that a user can receive is defined. Every time a new request arrives at a server, there are two operations that happen:

- *Fetch token:* The current number of tokens for that user is fetched. If it is greater than the limit defined then the request is dropped.
- *Update token:* If the fetched token is less than the limit for the time duration d , then the request is accepted and the token is appended.

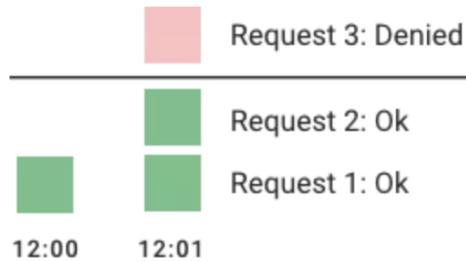
This algorithm is memory efficient as we are saving less amount of data per user for our application. The problem here is that it can cause race condition in a distributed environment. This happens when there are two requests from two different application servers trying to fetch the token at the same time.



Token Bucket Algorithm

3. **Fixed Window Counter**: Fixed window is one of the most basic rate limiting mechanisms. We keep a counter for a given duration of time, and keep incrementing it for every request we get. Once the limit is reached, we drop all further requests till the time duration is reset.

The advantage here is that it ensures that most recent requests are served without being starved by old requests. However, a single burst of traffic right at the edge of the limit might hoard all the available slots for both the current and next time slot. Consumers might bombard the server at the edge in an attempt to maximise number of requests served.



Fixed Window Counter

4. **Sliding Log** : Sliding log algorithm involves maintaining a time stamped log of requests at the user level. The system keeps these requests time sorted in a Set or a Table. It discards all requests with timestamps beyond a threshold. Every minute we look out for older requests and filter them out. Then we calculate the sum of logs to determine the request rate. If the request would exceed the threshold rate, then it is held, else it is served.

The advantage of this algorithm is that it does not suffer from the boundary conditions of fixed windows. Enforcement of the rate limit will remain precise. Since the system tracks the sliding log for each consumer, you don't have the stampede effect that challenges fixed windows.

However, it can be costly to store an unlimited number of logs for every request. It's also expensive to compute because each request requires calculating a summation over the consumer's prior requests, potentially across a cluster of servers. As a result, it does not scale well to handle large bursts of traffic or denial of service attacks.

5. **Sliding Window**: This is similar to the Sliding Log algorithm, but memory efficient. It combines the fixed window algorithm's low processing cost and the sliding log's improved boundary conditions.

We keep a list/table of time sorted entries, with each entry being a hybrid and containing the timestamp and the number of requests at that point. We keep a sliding window of our time duration and only service requests in our window for the given rate. If the sum of counters is more than the given rate of the limiter, then we take only the first sum of entries equal to the rate limit.

The **Sliding Window** approach is the best of the lot because it gives the flexibility to scale rate limiting with good performance. The rate windows are an intuitive way to present rate limit data to API consumers. It also avoids the starvation problem of the leaky bucket and the bursting problems of fixed window implementations

Rate Limiting in Distributed Systems

The above algorithms work very well for single server applications. But the problem becomes very complicated when there is a distributed system involved with multiple nodes or app servers. It becomes more complicated if there are multiple rate limited services distributed across different server regions. The two broad problems that come across in these situations are **Inconsistency** and **Race Conditions**.

Inconsistency

In case of complex systems with multiple app servers distributed across different regions and having their own rate limiters, we need to define a global rate limiter.

A consumer could surpass the global rate limiter individually if it receives a lot of requests in a small time frame. The greater the number of nodes, the more likely the user will exceed the global limit.

There are two ways to solve for these problems:

- Sticky Session: Have a sticky session in your load balancers so that each consumer gets sent to exactly one node. The downsides include lack of fault tolerance & scaling problems when nodes get overloaded. You can read more about sticky sessions [here](#)
- Centralized Data Store: Use a centralized data store like Redis or Cassandra to handle counts for each window and consumer. The added latency is a problem, but the flexibility provided makes it an elegant solution.

Race Conditions

Race conditions happen in a get-then-set approach with high concurrency. Each request gets the value of counter then tries to increment it. But by the time that write operation is completed, several other requests have read the value of the counter(which is not correct). Thus a very large number of requests are sent than what was intended. This can be mitigated using locks on the read-write operation, thus making it atomic. But this comes at a performance cost as it becomes a bottleneck causing more latency.

Throttling

Throttling is the process of controlling the usage of the APIs by customers during a given period. Throttling can be defined at the application level and/or API level. When a throttle limit is crossed, the server returns HTTP status “429 — Too many requests”.

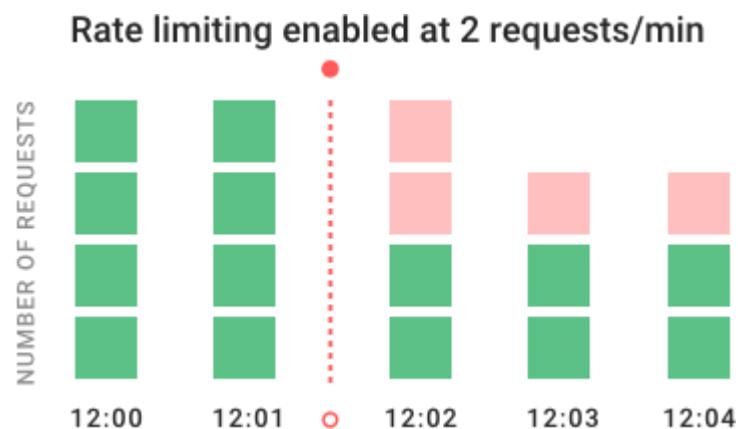
Types of Throttling:

1. **Hard Throttling**: The number of API requests cannot exceed the throttle limit.
2. **Soft Throttling**: In this type, we can set the API request limit to exceed a certain percentage. For example, if we have rate-limit of 100 messages a minute and 10% exceed-limit, our rate limiter will allow up to 110 messages per minute.
3. **Elastic or Dynamic Throttling**: Under Elastic throttling, the number of requests can go beyond the threshold if the system has some resources available. For example, if a user is allowed only 100 messages a minute, we can let the user send more than 100 messages a minute when there are free resources available in the system.

Solution 2:

What is rate limiting?

Rate limiting protects your APIs from inadvertent or malicious overuse by limiting how often each user can call the API. Without rate limiting, each user may make a request as often as they like, leading to “spikes” of requests that starve other consumers. Once enabled, rate limiting can only perform a fixed number of requests per second. A rate limiting algorithm helps automate the process.



In the example chart, you can see how rate limiting blocks requests over time. The API was initially receiving four requests per minute, shown in green. When we enabled rate limiting at 12:02, the system denied additional requests, shown in red.

Why is rate limiting used?

Rate limiting is very important for public APIs where you want to maintain a good quality of service for every consumer, even when some users take more than their fair share. Computationally-intensive endpoints are particularly in need of rate limiting — especially when served by auto-scaling, or by pay-by-the-computation services like [AWS Lambda](#). You also may want to rate limit [API technologies](#) that serve sensitive data because this could limit the data exposed if an attacker gains access in some unforeseen event.

Rate limiting can be used to accomplish the following:

- **Avoid resource starvation:** Rate limiting can improve the availability of services and help avoid friendly-fire denial-of-service (DoS) attacks.
- **Cost management:** Rate limiting can be used to enforce cost controls, (for example) preventing surprise bills from experiments or misconfigured resources .
- **Manage policies and quotas:** Rate limiting allows for the fair sharing of a service with multiple users.
- **Control flow:** For APIs that process massive amounts of data, rate limiting can be used to control the flow of that data. It can allow for the merging of multiple streams into one service or the equally distribution of a single stream to multiple workers.
- **Security:** Rate limiting can be used as defense or mitigation against some common attacks.

What kind of attacks can rate limiting prevent?

Rate limiting can be critical in protecting against a variety of bot-based attacks, including:

- DoS and DDoS (distributed denial-of-service) attacks
- Brute force and credential stuffing attacks
- Web scraping or site scraping

How do you enable rate limiting?

There are many different ways to enable rate limiting, and we will explore the pros and cons of varying rate limiting algorithms. We will also explore the issues that come up when scaling across a cluster. Lastly, we'll show you an example of how to quickly set up rate limiting using [Kong Gateway](#), which is the most popular open-source [API gateway](#).

Rate limiting algorithms

There are various algorithms for API rate limiting, each with its benefits and drawbacks. Let's review each of them so you can pick the ideal rate limiting design option for your API needs.

Leaky Bucket

[Leaky bucket](#) (closely related to [token bucket](#)) is an algorithm that provides a simple, intuitive approach to rate limiting via a queue, which you can think of as a bucket holding the requests. When registering a request, the system appends it to the end of the queue. Processing for the first item on the queue occurs at a regular interval or first in, first out (FIFO). If the queue is full, then additional requests are discarded (or leaked).

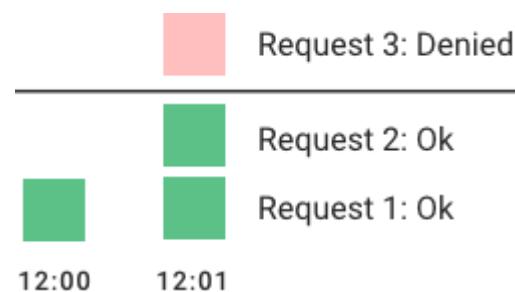


This algorithm's advantage is that it smooths out bursts of requests and processes them at an approximately average rate. It's also easy to implement on a single server or [load balancer](#) and is memory efficient for each user, given the limited queue size.

However, a burst of traffic can fill up the queue with old requests and starve more recent requests from being processed. It also provides no guarantee that requests get processed in a fixed amount of time. Additionally, if you load balance servers for fault tolerance or increased throughput, you must use a policy to coordinate and enforce the limit between them. We will come back to the challenges of distributed environments later.

Fixed Window

The system uses a window size of n seconds (typically using human-friendly values, such as 60 or 3600 seconds) to track the **fixed window** algorithm rate. Each incoming request increments the counter for the window. It discards the request if the counter exceeds a threshold. The current timestamp floor typically defines the windows, so 12:00:03, with a 60-second window length, would be in the 12:00:00 window.



This algorithm's advantage is that it ensures more recent requests get processed without being starved by old requests. However, a single burst of traffic that occurs near the

boundary of a window can result in the processing of twice the rate of requests because it will allow requests for both the current and next windows within a short time. Additionally, if many consumers wait for a reset window, they may stampede your API at the same time at the top of the hour.

Sliding Log

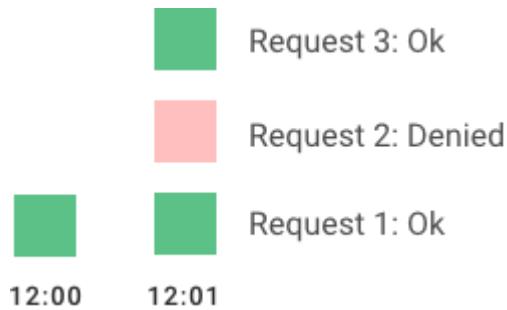
Sliding Log rate limiting involves tracking a time-stamped log for each consumer's request. The system stores these logs in a time-sorted hash set or table. It also discards logs with timestamps beyond a threshold. When a new request comes in, we calculate the sum of logs to determine the request rate. If the request would exceed the threshold rate, then it is held.

-  Request at 12:00:35 Hold
-  Request at 12:00:15 Ok
-  Request at 12:00:01 Ok

The advantage of this algorithm is that it does not suffer from the boundary conditions of fixed windows. Enforcement of the rate limit will remain precise. Since the system tracks the sliding log for each consumer, you don't have the stampede effect that challenges fixed windows. However, it can be costly to store an unlimited number of logs for every request. It's also expensive to compute because each request requires calculating a summation over the consumer's prior requests, potentially across a cluster of servers. As a result, it does not scale well to handle large bursts of traffic or denial of service attacks.

Sliding Window

Sliding Window is a hybrid approach that combines the fixed window algorithm's low processing cost and the sliding log's improved boundary conditions. Like the fixed window algorithm, we track a counter for each fixed window. Next, we account for a weighted value of the previous window's request rate based on the current timestamp to smooth out bursts of traffic. For example, if the current window is 25% through, we weigh the previous window's count by 75%. The relatively small number of data points needed to track per key allows us to scale and distribute across large clusters.



We recommend the **sliding window** approach because it gives the flexibility to scale rate limiting with good performance. The rate windows are an intuitive way to present rate limit data to API consumers. It also avoids the starvation problem of the leaky bucket and the bursting problems of fixed window implementations.

Rate limiting in distributed systems

Synchronization Policies

If you want to enforce a **global rate limit** when using a [cluster of multiple nodes](#), you must set up a policy to [enforce it](#). If each node were to track its rate limit, a consumer could exceed a global rate limit when sending requests to different nodes. The greater the number of nodes, the more likely the user will exceed the global limit.

The simplest way to enforce the limit is to set up sticky sessions in your load balancer so that each consumer gets sent to exactly one node. The disadvantages include a lack of fault tolerance and scaling problems when nodes get overloaded.

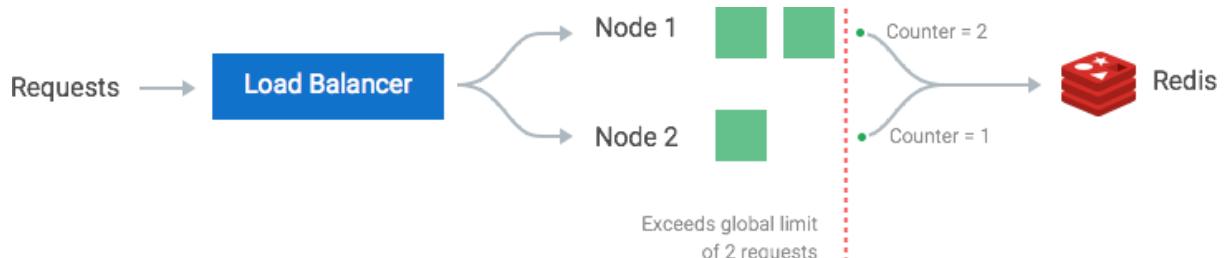
A better solution that allows more flexible load-balancing rules is to use a centralized data store such as Redis or [Cassandra](#). A centralized data store will collect the counts for each window and consumer. The two main problems with this approach are increased latency making requests to the data store and race conditions, which we will discuss next.



Race Conditions

One of the most extensive problems with a centralized data store is the potential for [race conditions](#) in [high concurrency](#) request patterns. This issue happens when you use a naïve

“get-then-set” approach, wherein you retrieve the current rate limit counter, increment it, and then push it back to the datastore. This model’s problem is that additional requests can come through in the time it takes to perform a full cycle of read-increment-store, each attempting to store the increment counter with an invalid (lower) counter value. This allows a consumer to send a very high rate of requests to bypass rate limiting controls.



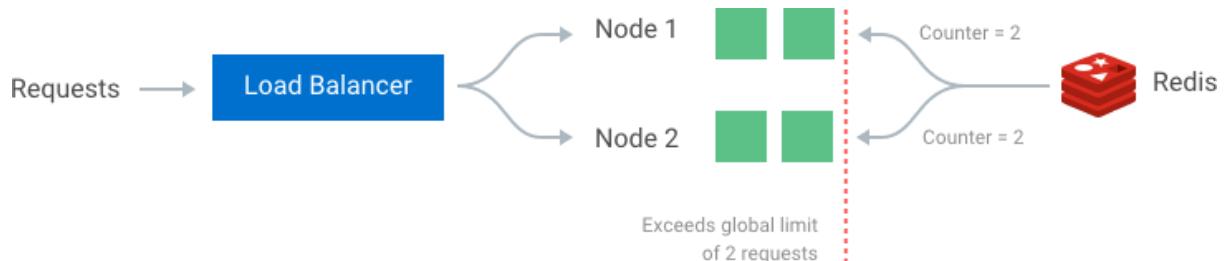
One way to avoid this problem is to put a “**lock**” around the key in question, preventing any other processes from accessing or writing to the counter. A lock would quickly become a significant performance bottleneck and does not scale well, mainly when using remote servers like Redis as the backing datastore.

A better approach is to use a “**set-then-get**” mindset, relying on atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.

Optimizing for Performance

The increased [latency](#) is another disadvantage of using a centralized data store when checking the rate limit counters. Unfortunately, even checking a fast data store like Redis would result in milliseconds of additional latency for every request.

Make checks locally **in memory** to make these rate limit determinations with minimal latency. To make local checks, relax the rate check conditions and use an eventually consistent model. For example, each node can create a data sync cycle that will synchronize with the centralized data store. Each node periodically pushes a counter increment for each consumer and window to the datastore. These pushes atomically update the values. The node can then retrieve the updated values to update its in-memory version. This cycle of converge → diverge → reconverge among nodes in the cluster is eventually consistent.

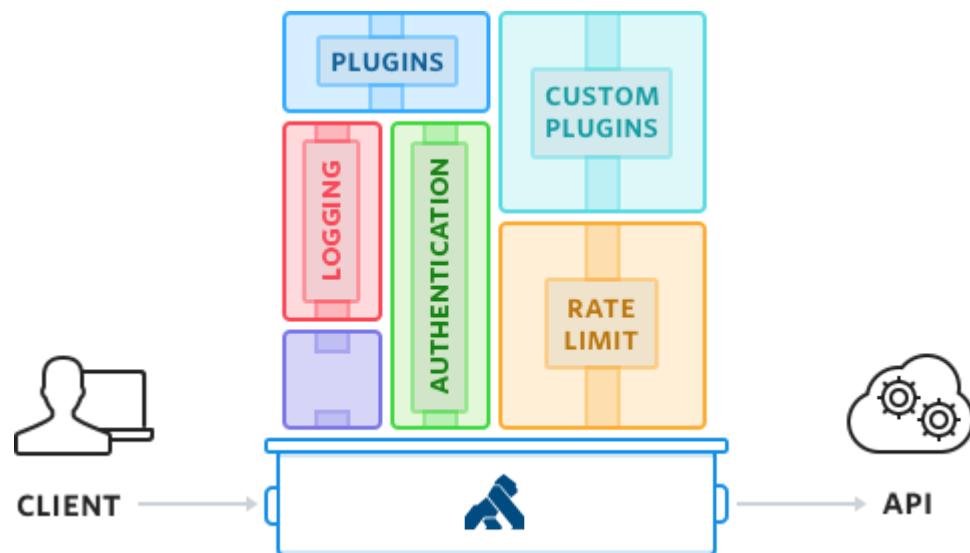


The periodic rate at which nodes converge should be configurable. Shorter sync intervals will result in less divergence of data points when spreading traffic across multiple nodes in the cluster (e.g., when sitting behind a round robin balancer). Whereas longer sync intervals put less read/write pressure on the datastore and less overhead on each node to fetch new synced values.

Quickly set up rate limiting with Kong API Gateway

Kong is an [open source API gateway](#) that makes it very easy to build scalable services with rate limiting. It scales perfectly from single Kong Gateway nodes to massive, globe-spanning Kong clusters.

Kong Gateway sits in front of your APIs and is the main entry point to your upstream APIs. While processing the request and the response, Kong Gateway will execute any plugin that you have decided to add to the API.



Kong API Gateway's rate limiting plug-in is highly configurable. It:

- Offers flexibility to define multiple rate limit windows and rates for each API and consumer
- Includes support for local memory, Redis, Postgres, and Cassandra backing datastores
- Offers a variety of data synchronization options, including synchronous and eventually consistent models

You can quickly [try Kong Gateway](#) on one of your dev machines to test it out. My favorite way to get started is to use the [AWS cloud formation template](#) since I get a pre-configured dev machine in just a few clicks. Just choose one of the HVM options, and set your instance sizes to use t2.micro as these are affordable for testing. Then ssh into a command line on your new instance for the next step.

Adding an API on Kong Gateway

The next step is [adding an API](#) on Kong Gateway using the admin API, which can be done by setting up a Route and a Service entity. We will use [httpbin](#), a free testing service for APIs, as our example upstream service,. The get URL will mirror back my request data as JSON. We also assume Kong Gateway is running on the local system at the default ports.

```
curl -i -X POST \
--url http://localhost:8001/services/ \
--data 'name=httpbin' \
--data 'url=http://httpbin.org/get'
```

```
curl -i -X POST \
--url http://localhost:8001/services/httpbin/routes \
--data 'paths[]=/test'
```

Now Kong Gateway is aware that every request sent to “/test” should be proxied to httpbin. We can make the following request to Kong Gateway on its proxy port to test it:

```
curl http://localhost:8000/test
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.51.0",
    "X-Forwarded-Host": "localhost"
  },
  "origin": "localhost, 52.89.171.202",
  "url": "http://localhost/get"
}
```

It's alive! Kong Gateway received the request and proxied it to httpbin, which has mirrored back the headers for my request and my origin IP address.

Adding Basic Rate-Limiting

Let's go ahead and protect it from an excessive number of requests by adding the rate-limiting functionality using the community edition [Rate-Limiting plugin](#), with a limit of 5 requests per minute from every consumer:

```
curl -i -X POST http://localhost:8001/services/httpbin/plugins/ \
-d "name=rate-limiting" \
-d "config.minute=5"
```

If we now make more than five requests, Kong Gateway will respond with the following error message:

```
curl http://localhost:8000/test
```

```
{
"message":"API rate limit exceeded"
}
```

Looking good! We have added an API on Kong Gateway, and we added rate-limiting in just two HTTP requests to the admin API.

It defaults to rate limiting by IP address using fixed windows and synchronizes across all nodes in your cluster using your default datastore. For other options, including rate limiting per consumer or using another datastore like Redis, please see the [documentation](#).

For a more detailed tutorial, check out [Protecting Services With Kong Gateway Rate Limiting](#)

[**>>**](#)

Better performance with advanced rate limiting and Konnect

The [Advanced Rate Limiting plugin](#) adds support for the sliding window algorithm for better control and performance. The sliding window prevents your API from being overloaded near window boundaries, as explained in the sections above. For low latency, it uses an in-memory table of the counters and can synchronize across the cluster using asynchronous or synchronous updates. This gives the latency of local thresholds and is scalable across your entire cluster.

The first step is [start a free trial of Konnect](#). You can then configure the rate limit, the window size in seconds, and how often to sync the counter values. It's straightforward to use, and you can get this robust control with a simple API call:

```
curl -i -X POST http://localhost:8001/services/httpbin/plugins \
-d "name=rate-limiting" \
-d "config.limit=5" \
-d "config.window_size=60" \
-d "config.sync_rate=10"
```

The enterprise edition also supports Redis Sentinel, which makes Redis highly available and more fault-tolerant. You can read more in the [rate limiting plugin documentation](#).

Other features include an admin GUI, more security features like role-based access control, analytics and professional support. If you're interested in learning more about the Enterprise edition, just [contact Kong's sales team to request a demo](#).

Solution 3:

Design a Rate Limiter.

Key: Italicized lines are for your understanding, they may not be part of the actual interview discussion. Also, the design is far from perfect but I have tried to capture things that you can realistically cover in a 25-35 mins interview.

Rate Limiter: A rate limiter in HTTP world is used to limit the number of client requests allowed to be processed over a specified period. Here are few examples:

1. In a banking system you may request OTP (One Time Password) once every 5 minutes.
2. In a social media platform, a user maybe allowed to post maximum 5 posts per minute
3. You can register a new user from same IP maximum 5 times per day. This is case where we want to prevent bot/sock-puppet accounts

Benefits of having a Rate Limiter:

1. Prevents DDoS/DoS (Denial of Service) attacks. DDoS attacks can swamp the system and cause resource starvation leading to massive downtimes. A proper and robust rate limiter can help reduce chances of DDoS attack being successful
2. Cost benefit: If you have a rate limiting system in place, you can control maximum requests that your system is allowed to process. This in turn can be used to allocate resources in other places rather than scaling servers to handle ever increasing request per second

In general a system design interview goes as below

1. **Gather scope requirements** - High level idea on what is the intent, scope, scale of system. Standalone vs BeSpoke service etc. Distributed vs non distributed. This is important because it helps us design keeping the scalability aspect in mind
2. **Functional and Non functional requirements:** Functional requirements pertains to the APIs of your system and Non Functional refers to the performance aspect such as availability, low latency etc
3. **Basic design** - start with a simple High level diagram (a few boxes is fine) explaining the flow and various helper services you can think of. Here it is important to ask interview as to where they want you to deep dive. For example, if you are designing a ride-share service (Uber) then it depends on what interviewer wants to focus on, maybe they are interested in driver pricing optimisation or they want to improve the map search portion. Clarify whenever you are uncertain. System deisgn interviews are DISCUSSIONS with NO PERFECT SOLUTION.
4. **Dive deeper** as per instructions and expand your design, it is a good practice to highlight possible shortcomings/bottlenecks in your design while you are making them.
5. **Validation:** validate that your design works, address possible bottlenecks and seek interviewer feedback

6. **Future scope:** such as a metric service to measure performance of your design; whether your design will work with 1 million customers, 100 million customers etc. What may work for 1 million users may not scale for 100 million.
7. **closing points :** aligns with point 6 but can be a general commentary on your design and you may also mention other solutions briefly

Now let us proceed to the actual solution portion:

We can ask a few questions such as, **If the rate limiter is to be a server side/client side?**, **Whether we should inform users/clients who are rate limited?**, **What is the expected scale of the system. How much throughput are we looking forward to handle?** and **Whether rate limiting is client specific or system specific, like if we would use IP, userId etc to enforce the rate limiting logic**

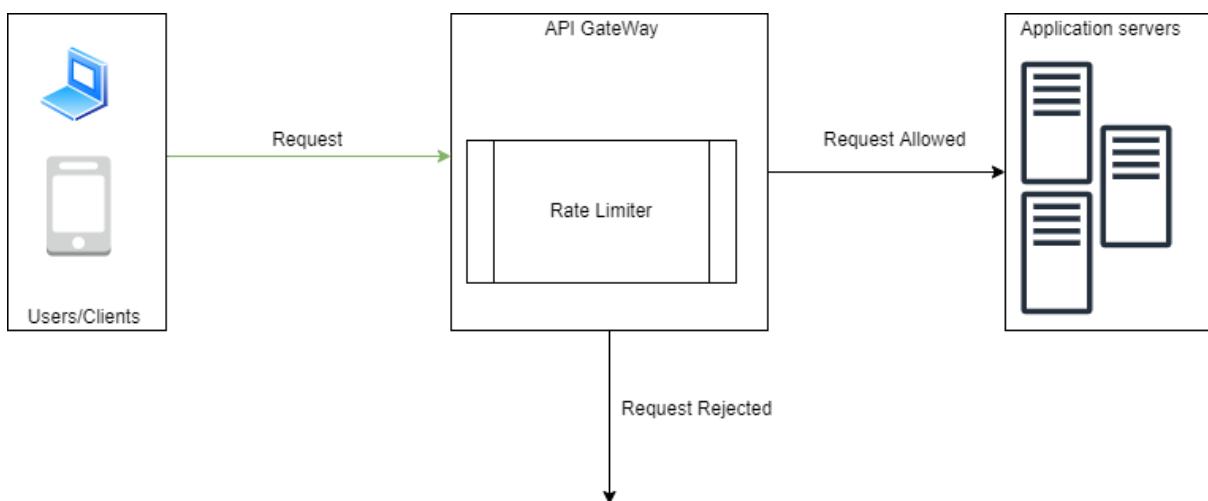
Based on the above I am assuming below as my **functional requirements**:

1. Server side rate limiting
2. Should be able to handle large number of requests
3. Clients should be informed of being rate limited (proper exception handling)
4. For now, let us assume rate limiter is a separate service
5. should have clearly defined APIs such as allowRequest() that would return a boolean signifying if request is to be processed or dropped

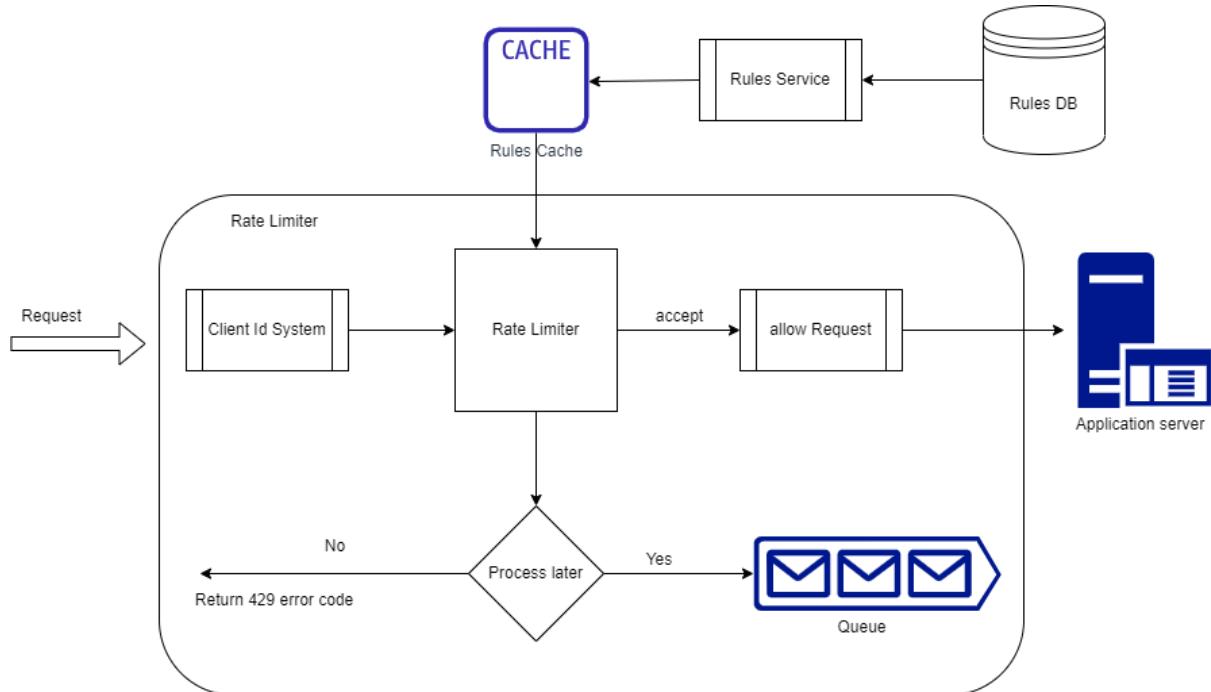
Non-functional requirements

1. Highly available service (since our system is a standalone application that will be used by various other services, it should be highly available)
2. Low latency: The rate limiting should not impact the application's response time. Our logic should be efficient and optimized
3. Fault tolerant: if rate limiter is down, it should not cause failures on the client applications. Failures needs to be handled
4. Resource optimisation: Rate limiter should not be too expensive to implement

A very simple design to start with:



Now we can proceed further, at this point we can explode the rate limiter component and go to full fledged design. In order to do that we must keep track of the **functional** requirements



Briefly let us go over the job/role of each component

1. Client Id system is for helping us identify client. As our rate limiter is a plug and play component, it must have a mechanism to id the clients and then use the id to pull the rules
2. Rules DB is to store rules for all clients onboarded onto our service.
3. Rules Service is responsible for loading Rules from DB into Rules Cache
4. Rules Cache is for fast access of rules for client using their id (coming from id system)

I am not diving into the rate limiting algorithm because it depends on what direction your interviewer wants you to take. But Do read the following important Rate Limiting algos:

- Token Bucket (quite common and very easy to implement)
- Leaking Bucket
- Fixed window
- Sliding Window
- Sliding window logged

Validation (that our design works)

Scenario 1: Request comes => client Id is say 100 => Rate Limiter pulls rules for id 100 from Rules Cache => Rate Limiter validates if enough tokens are there for request to pass => tokens are there => request allowed and sent to Application server

Scenario 2: Request comes => client Id is say 100 => Rate Limiter pulls rules for id 100 from Rules Cache => Rate Limiter validates if enough tokens are there for request to pass => not

enough tokens => request rejected and HTTP code 429 (too many requests) returned to client.

Questions for you.

1. How will you make it distributed?
2. what challenges you foresee if distributed Rate limiter is implemented.
3. Can you identify points of failures in this design.
4. Handling failure when Rate Limiter service goes down
5. Rate Limting done at other level of the OSI model

Solution 4:

What is Rate Limiting?

Imagine that a Distributed Denial of Service (DDOS) attack has been started to your server by an attacker. What can you do? Your API will not be available for users. Why? A DDOS attack can send a ton of requests to your API in a second, and the API can't handle these requests. We can overcome this issue with auto-scaling. Auto-scaling might be costly, and in this scenario, the above requests were from an attack and not the real customer requests. So auto-scaling won't be a viable solution since the newly provisioned instances will also be victims of DDOS. We can use a mechanism to limit the rate at which requests are made to avoid the server. We call this feature as rate-limiting. Rate limiting is the easiest way to prevent server overloading. More than that, rate-limiting can be used to give an excellent customer experience and commercial use of APIs.

Rate Limiting is important to preventing malicious attacks on your APIs. That's why lack of resources is one of OWASP's top API risks.

Rate limiting applies to the number of calls a user can make to an API within a set time frame. This is used to help control the load that's put on the system.

Rate limiting helps prevent a user from exhausting the system's resources. Without rate limiting, it's easier for a malicious party to overwhelm the system. This is done when the system is flooded with requests for information, thereby consuming memory, storage, and network capacity.

An API that utilizes rate limiting may throttle clients that attempt to make too many calls or temporarily block them altogether. Users who have been throttled may either have their requests denied or slowed down for a set time. This will allow legitimate requests to still be fulfilled without slowing down the entire application.

API responds with HTTP Status Code **429 Too Many Requests** when a request is rate limited or throttled.

Using Rate Limiter we can enforce policies as below:

- A maximum of 20 accounts are allowed to be created per day from a unique source IP address. This helps minimizing spams or malicious accounts creation, and Denial of Service (DoS) and Distributed Denial of Service (DDoS).
- Devices are allowed 5 failed credit card transactions per day.
- Users may only send 1 message per day with risky keywords.

Use Cases: Why do we need Rate Limiter?

Prevent Brute Force Attacks:

One of the most common use cases for rate limiting is to block brute force attacks. In a brute force attack, a hacker uses automation to send an endless stream of requests to an API, hoping that eventually one may be accepted. Limiting client access will slow down this attack. At the same time, the receiving system should notice the unexpectedly large number of failed requests and generate alerts so that further action may be taken.

In some cases, a user may accidentally cause a brute attack—a bug may cause repeated failed requests, causing the client to keep trying. Rate limiting would help to force a temporary stop and allow for follow-up action.

Prevent Denial of Service Attacks:

Another use case is to prevent a Denial of Service (DoS) attack. DoS attacks occur when a user attempts to slow or shut down an application completely. The user may do this by flooding the API with requests. Multiple users may attempt to overwhelm the system in this way as well; this is known as a Distributed Denial of Service (DDoS) attack.

Preventing resource starvation:

The most common reason for rate limiting is to improve the availability of API-based services by avoiding resource starvation. Many load-based denial-of-service incidents in large systems are unintentional—caused by errors in software or configurations in some other part of the system—not malicious attacks (such as network-based distributed denial of service attacks). Resource starvation that isn't caused by a malicious attack is sometimes referred to as friendly-fire denial of service (DoS).

Generally, a service applies rate limiting at a step before the constrained resource, with some advanced-warning safety margin. Margin is required because there can be some lag in loads, and the protection of rate limiting needs to be in place before critical contention for a resource happens. For example, a RESTful API might apply rate limiting to protect an underlying database; without rate limiting, a scalable API service could make large numbers of calls to the database concurrently, and the database might not be able to send clear rate-limiting signals.

Prevent Cascading Failures:

Rate limiting is generally put in place as a defensive measure for services. Shared services need to protect themselves from excessive use—whether intended or unintended—to

maintain service availability. Even highly scalable systems should have limits on consumption at some level. For the system to perform well, clients must also be designed with rate limiting in mind to reduce the chances of **cascading failure**. Rate limiting on both the client side and the server side is crucial for maximizing throughput and minimizing end-to-end latency across large distributed systems.

What is Throttling?

Rate Limiting is a process that is used to define the rate and speed at which consumers can access APIs. **Throttling** is the process of controlling the usage of the APIs by customers during a given period. Throttling can be defined at the application level and/or API level. When a throttle limit is crossed, the server returns HTTP status “429 - Too many requests”.

What Are the Major Types of Rate Limiting?

There are several major types of rate limiting models that a business can choose between depending on which one offers the best fit for a business based on the nature of the web services that they offer, as we will explore in greater detail below.

- **User-Level Rate Limiting**

In cases where a system can uniquely identify a user, it can restrict the number of API requests that a user makes in a time period. For example, if the user is only allowed to make two requests per second, the system denies the user's third request made in the same second. User-level rate limiting ensures fair usage. However, maintaining the usage statistics of each user can create an overhead to the system that if not required for other reasons, could be a drain on resources.

- **Server-Level Rate Limiting**

Most API-based services are distributed in nature. That means when a user sends a request, it might be serviced by any one of the many servers. In distributed systems, rate limiting can be used for load-sharing among servers. For example, if one server receives a large chunk of requests out of ten servers in a distributed system and others are mostly idle, the system is not fully utilized. There will be a restriction on the number of service requests that a particular server can handle in server-level rate limiting. If a server receives requests that are over this set limit, they are either dropped or routed to another server. Server-level rate limiting ensures the system's availability and prevents denial of service attacks targeted at a particular server.

- **Geography-Based Rate Limiting**

Most API-based services have servers spread across the globe. When a user issues an API request, a server close to the user's geographic location fulfills it.

Organizations implement geography-based rate limiting to restrict the number of service requests from a particular geographic area. This can also be done based on timing. For example, if the number of requests coming from a particular geographic location is small from 1:00 am to 6:00 am, then a web server can have a rate limiting rule for this particular period. If there is an attack on the server during these hours,

the number of requests will spike. In the event of a spike, the rate limiting mechanism will then trigger an alert and the organization can quickly respond to such an attack.

Client-side strategies

The strategies described so far apply to rate limiting on the server side. However, these strategies can inform the design of clients, especially when you consider that many components in a distributed system are both client and server.

Just as a service's primary purpose in using rate limiting is to protect itself and maintain availability, a client's primary purpose is to fulfill the request it is making to a service. A service might be unable to fulfill a request from a client for a variety of reasons, including the following:

- The service is unreachable because of network conditions.
- The service returned a non-specific error.
- The service denies the request because of an authentication or authorization failure.
- The client request is invalid or malformed.
- The service rate-limits the caller and sends a backpressure signal (commonly a 429 response).

In response to rate-limiting, intermittent, or non-specific errors, a client should generally retry the request after a delay. It is a best practice for this delay to increase exponentially after each failed request, which is referred to as exponential backoff. When many clients might be making schedule-based requests (such as fetching results every hour), additional random time (jitter) should be applied to the request timing, the backoff period, or both to ensure that these multiple client instances don't become periodic thundering herd, and themselves cause a form of DDoS.

Imagine a mobile app with many users that checks in with an API at exactly noon every day, and applies the same deterministic back-off logic. At noon, many clients call the service, which starts rate limiting and returning responses with a **429 Too Many Requests** status code. The clients then dutifully back off and wait a set amount of time (deterministic delay) of exactly 60 seconds, and then at 12:01 the service receives another large set of requests. By adding a random offset (jitter) to the time of the initial request or to the delay time, the requests and retries can be more evenly distributed, giving the service a better chance of fulfilling the requests.

Ideally, non-idempotent requests can be made in the context of a strongly consistent transaction, but not all service requests can offer such guarantees, and so retries that mutate data need to consider the consequences of duplicate action.

For situations in which the client developer knows that the system that they are calling is not resilient to stressful loads and does not support rate-limiting signals (back-pressure), the client library developer or client application developer can choose to apply self-imposed throttling, using the same techniques for enforcing rate limits that can be used on the server side. These techniques are discussed below in the algorithms section.

For clients of APIs that defer the response with an asynchronous long-running operation ID, the client can choose to enter a blocking loop polling the status of the deferred response, removing this complexity from the user of the client library.

What Are the Algorithms Used for Rate Limiting?

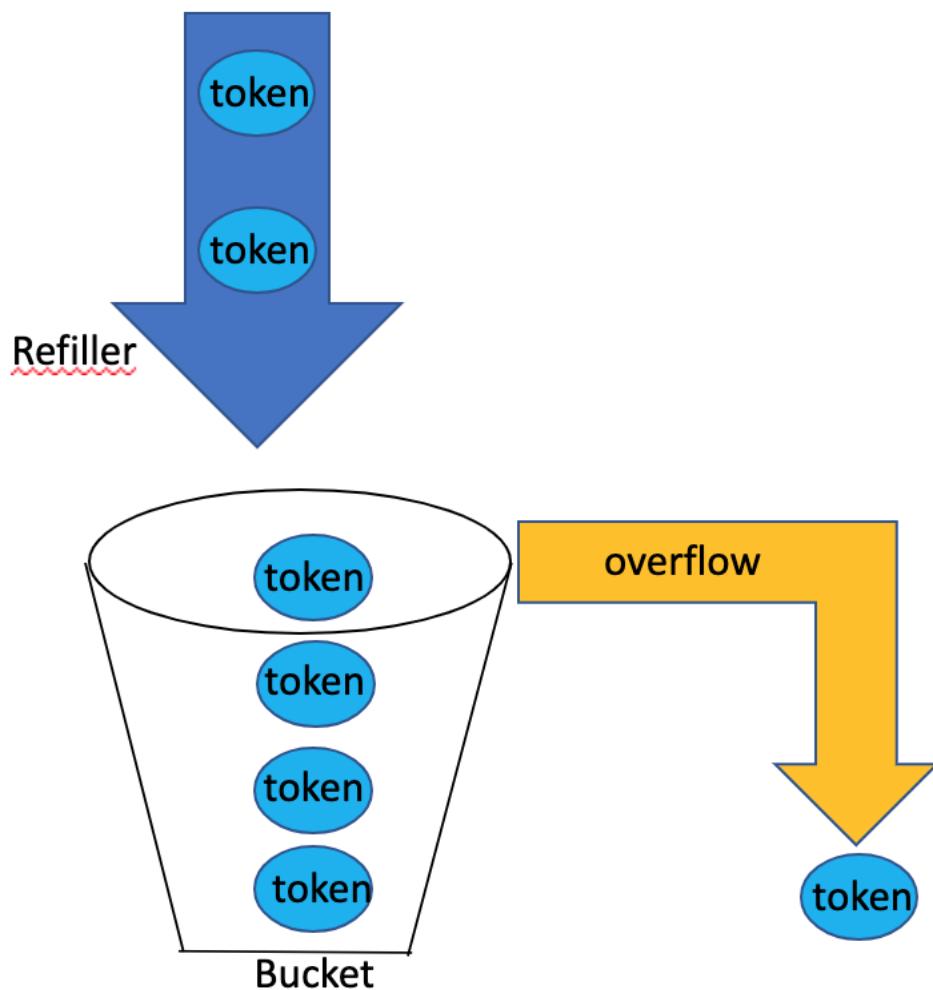
In general, a rate is a simple count of occurrences over time. However, there are several different techniques for measuring and limiting rates, each with their own uses and implications.

Below are the common rate limiting algorithms. We will now discuss each of these algorithms in great details.

1. Token Bucket algorithm

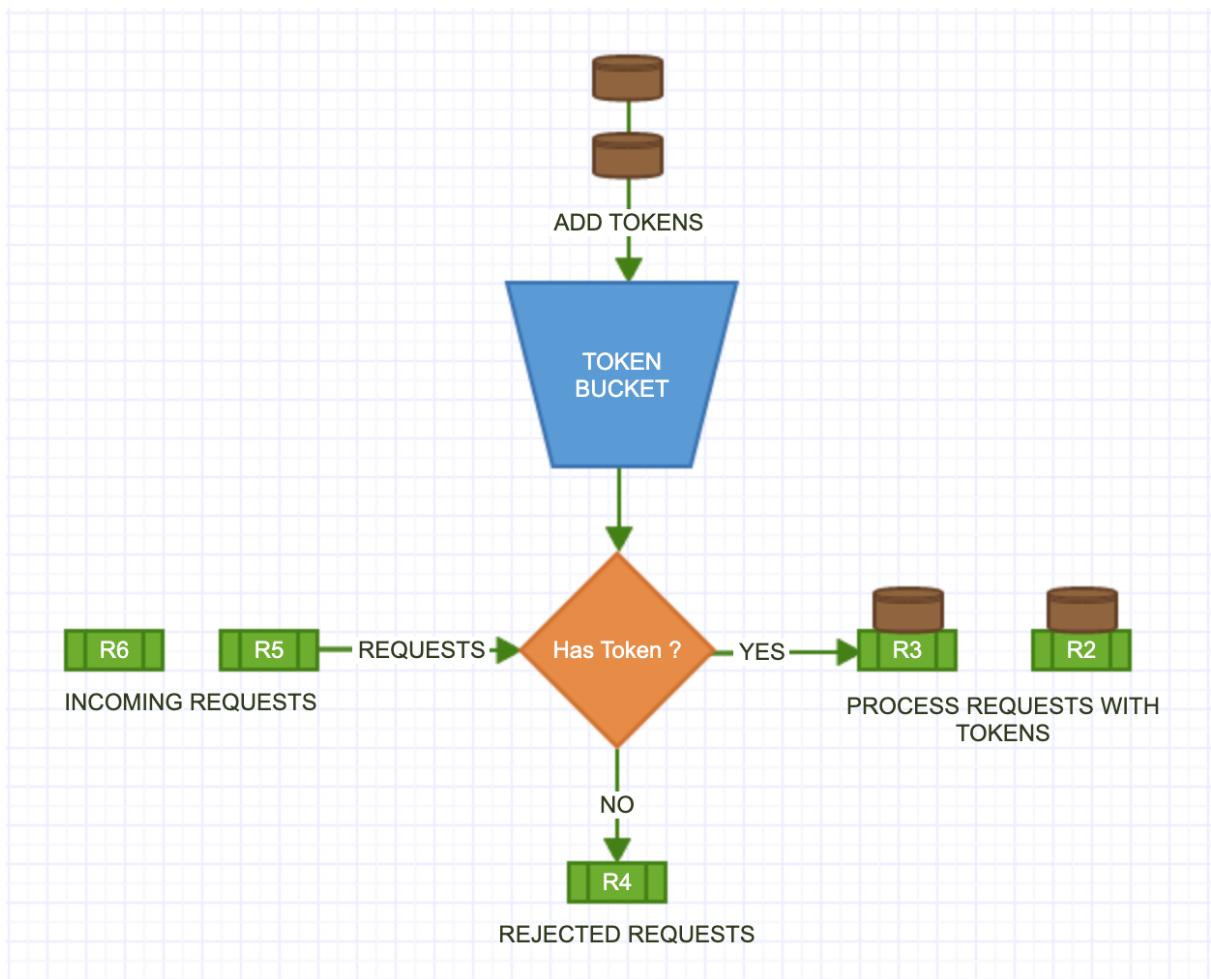
The *Token Bucket throttling algorithm* works as follows: A token bucket is a container that has pre-defined capacity. Tokens are put in the bucket at preset rates periodically. Once the bucket is full, no more tokens are added.

In the example below, the bucket has **maximum capacity** of 4. So the bucket will hold a maximum of 4 tokens at any given time. The **Refiller** puts **2 tokens per second** in the container. At any point of time if the bucket becomes full, i.e. reaches its maximum capacity, then extra tokens will **overflow**.



As you can already understand, that we are implementing **rate limiting or throttling mechanism for an API** using a **bucket or container** and **tokens**.

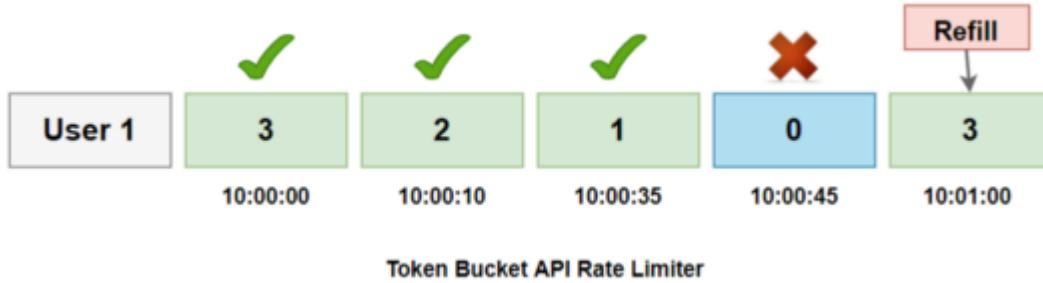
Each API request **consumes one token**. When a request arrives, we check if there is at least one token left in the bucket. If there is, we take one token out of the bucket, and the request goes through. If the the bucket is empty, the request is dropped. This is illustrated in below doagram.



Example:

- In the below example, there is a limit of 3 requests per minute per user.
- User 1 makes the first request at **10:00:00**; the available token is 3, therefore the request is approved, and the available token count is reduced to 2.
- At **10:00:10**, the user's second request, the available token is 2, the request is permitted, and the token count is decremented to 1.
- The third request arrives at **10:00:35**, with token count 1 available, therefore the request is granted and decremented.
- The 4th request will arrive at **10:00:45**, and the available count is already zero, hence API is denied.
- Now, at **10:01:00**, the count will be refreshed with the available token 3 for the third time.

2.



Token buckets with customizable refill amounts let you express more complex rate limits like:

- Each user account is allowed a maximum of 10 failed logins per hour, which refill at a rate of 1 login per hour.
- Each geocoded shipping address is allowed a maximum of \$200 worth of purchases per day from mail.ru email addresses, which refills at a rate of \$50 per day.

3.

As you can see, these rate limits allow a lot of freedom for legitimate user behavior, but quickly clamp down on repeated violations.

Implementation of Token Bucket algorithm

Each bucket has a few properties:

- **key**
a unique byte string that identifies the bucket
- **maxAmount**
the maximum number of tokens the bucket can hold
- **refillTime**
the amount of time between refills
- **refillAmount**
the number of tokens that are added to the bucket during a refill
- **value**
the current number of tokens in the bucket
- **lastUpdate**
the last time the bucket was updated

First, if the bucket doesn't exist, we create it:

```
bucket.value = bucket.maxAmount  
bucket.lastUpdate = now()
```

Then we refill the bucket if there are any pending refills:

```
refillCount = floor((now() - bucket.lastUpdate) / bucket.refillTime)  
bucket.value = min(  
    bucket.maxAmount,  
    bucket.value + refillCount * bucket.refillAmount  
)  
bucket.lastUpdate = min(  
    now(),  
    bucket.lastUpdate + refillCount * bucket.refillTime  
)
```

Next, we check if tokens >= bucket.value and bail out if it is true.

Finally, we take the tokens out of the bucket:

```
bucket.value -= tokens
```

4.

Use of In-Memory Cache:

Our algorithm needs to persist below data to operate correctly:

- **user id or client id or tenant id or app key or source ip address or precise geographic location** to represent who is sending the API request or making the API call.
- **number of token(s) left for that user/client/tenant in the bucket**
- **timestamp** when the request was sent

5.

Keep in mind the rate at which the traffic comes is not in our control, at any point of

time we should be prepared to see a burst or spike in traffic. **Using a database is not a good idea due to slowness of disk access. In memory cache is the way to go. Because it is quick and supports a time-based expiration technique, an in-memory cache is chosen.** [Redis, for example, is a popular choice for rate-limiting. “INCR” and “EXPIRE” are two commands that can be used to access in-memory storage.](#)

- INCR: It is used to increase the stored counter by 1.
 - EXPIRE: It is used to set a timeout for the counter. The counter is automatically deleted when the timeout expires.
6. For each unique user, we would record their last request's Unix timestamp and available token count within a hash in Redis.

key	value
“user_1”:	{“tokens”: 2, “ts”: 1490868000}

User_1 has two tokens left in their token bucket and made their last request on Thursday, March 30, 2017 at 10:00 GMT.

Whenever a new request arrives from a user, the rate limiter would have to do a number of things to track usage. It would fetch the entry from Redis and refill the available tokens based on a chosen refill rate and the time of the user's last request. Then, it would update the entry with the current request's timestamp and the new available token count. When the available token count drops to zero, the rate limiter knows the user has exceeded the limit.

Token bucket with 1 token/min refill

```
"user_1": {"tokens": 2, "ts": 1490868000}
```

Request at 10:00:07 AM

- Remove token
- ✓ Allow request



```
"user_1" : {"tokens": 1, "ts": 1490868007}
```

Request at 10:02:15 AM

- ⌚ Refill 2 tokens for time passed
- Remove token
- ✓ Allow request



```
"user_1" : {"tokens": 2, "ts": 1490868135}
```

Request at 10:02:24 AM

- Remove token
- ✓ Allow request



```
"user_1" : {"tokens": 1, "ts": 1490868144}
```

Request at 10:02:36 AM

- Remove token
- ✓ Allow request



```
"user_1" : {"tokens": 0, "ts": 1490868156}
```

User_1 has two tokens left in their token bucket and made their last request on Thursday, March 30, 2017 at 10:00 GMT.

We can break that algorithm into three steps

- Fetch the token
- Access the token
- Update the token

7. Consider we have planned to limit our API 10 requests/minute. For every unique user, we can track last time they have accessed the API and available token(number) of that user. These things can be stored in the Redis server with the key as user id.
 - **Fetch the token:** So when a request comes from the user, then our rate-limiting algorithm first fetch the user's token using the user id.
 - **Access the token:** When we have enough tokens for the particular user, then it will allow processing otherwise it blocks the request to hit the API. The last access time and remaining tokens of that specific user will be changed.
 - **Update Token:** Rate limiting algorithm will update the token in the Redis as well. One more thing, the tokens will be restored after the time interval. In our scenario, ten will be updated after the time period as a token value.
8. In this algorithm, we maintain a counter which shows how many requests a user has left and time when the counter was reset. For each request a user makes,
 - Check if window time has been elapsed since the last time counter was reset. For rate 100 requests/min, current time 10:00:27, last reset time 9:59:00 is not elapsed and $9:59:25 - 9:59:00 > 60$ sec is elapsed.
 - If window time is not elapsed, check if the user has sufficient requests left to handle the incoming request.
 - If the user has none left, the request is dropped.
 - Otherwise, we decrement the counter by 1 and handle the incoming request.
 - If the window time has been elapsed, i.e., the difference between last time counter was reset and the current time is greater than the allowed window(60s), we reset the number of requests allowed to allowed limit(100)

```

// RateLimitUsingTokenBucket .
func RateLimitUsingTokenBucket(
    userID string,
    intervalInSeconds int64,
    maximumRequests int64) bool {

    // userID can be apikey, location, ip
    value, _ := redisClient.Get(ctx, userID+"_last_reset_time").Result()

    lastResetTime, _ := strconv.ParseInt(value, 10, 64)

    // if the key is not available,
    // i.e., this is the first request, lastResetTime will be set to 0
    // and counter be set to max requests allowed
    // check if time window since last counter reset has elapsed
    if time.Now().Unix() - lastResetTime >= intervalInSeconds {
        // if elapsed, reset the counter
        redisClient.Set(ctx, userID+"_counter", strconv.FormatInt(maximumRequests,
10), 0)
    }

    else {
        value, _ := redisClient.Get(ctx, userID+"_counter").Result()

        requestLeft, _ := strconv.ParseInt(value, 10, 64)

        if requestLeft <= 0 { // request left is 0 or < 0
            // drop request
            return false
        }
    }

    // decrement request count by 1
    redisClient.Decr(ctx, userID+"_counter")

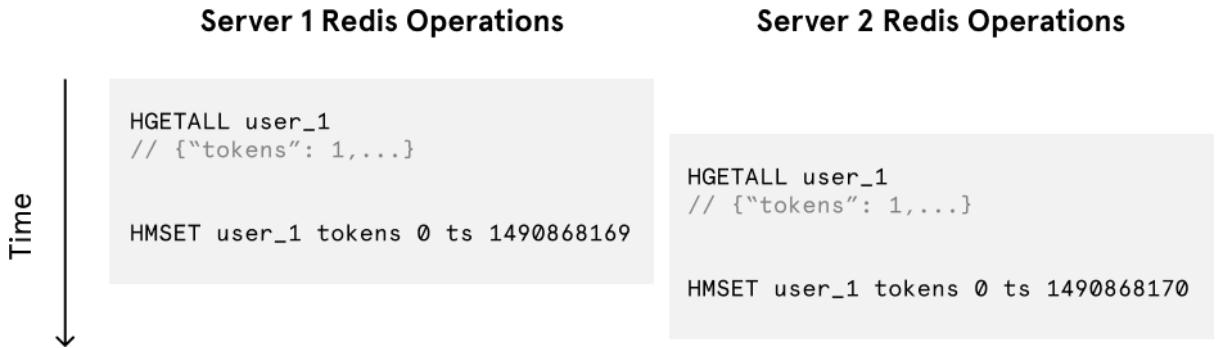
    // handle request
    return true
}

```

9.

Distributed Rate Limiting, Race Condition and Concurrency:

Despite the token bucket algorithm's elegance and tiny memory footprint, its **Redis operations aren't atomic**. In a **distributed environment**, the “read-and-then-write” behavior creates a race condition, which means the rate limiter can at times be too lenient.



If only a single token remains and two servers' Redis operations interleave, both requests would be let through.

Imagine if there was only one available token for a user and that user issued multiple requests. If two separate processes served each of these requests and concurrently read the available token count before either of them updated it, each process would think the user had a single token left and that they had not hit the rate limit.

Our token bucket implementation could achieve atomicity if each process were to fetch a **Redis lock** for the duration of its Redis operations. This, however, would come at the expense of slowing down concurrent requests from the same user and introducing another layer of complexity. Alternatively, we could make the token bucket's Redis operations atomic via [Lua scripting](#).

Pros:

- Token Bucket algorithm is very simple and easy to implement.
- Token Bucket algorithm is very memory efficient.
- Token Bucket technique allows spike in traffic or burst of traffic. A request goes through as long as there are tokens left. This is super important since traffic burst is not uncommon. One example is events like [Amazon Prime Day](#) when traffic spikes for a certain time period.

10.

Cons:

- A race condition, as described above, may cause an issue in a distributed system due to concurrent requests from the same user. We have discussed above how to avoid this situation.

11.

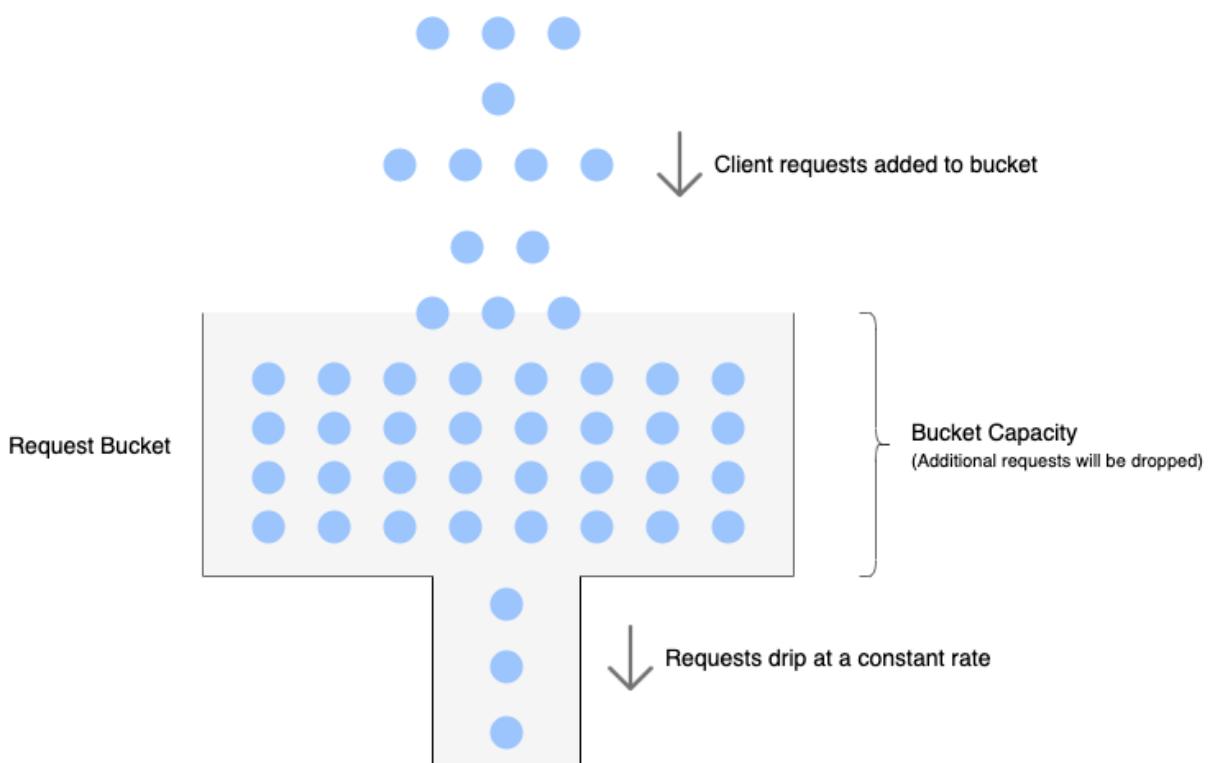
Which internet companies use Token Bucket for their rate limiting?

- [Stripe](#)
- [Amazon Elastic Compute Cloud or Amazon EC2](#)

12.

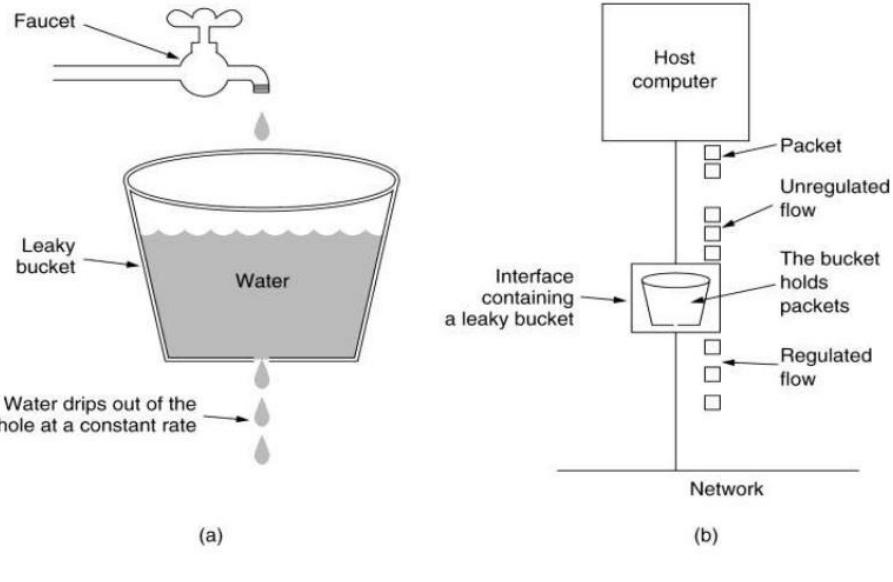
13. Leaky Bucket algorithm

Similar to the token bucket, leaky bucket also has a bucket with a finite capacity for each client. However, instead of tokens, it is filled with requests from that client. Requests are taken out of the bucket and processed at a constant rate. If the rate at which requests arrive is greater than the rate at which requests are processed, the bucket will fill up and further requests will be dropped until there is space in the bucket.



The following image perfectly illustrates the leaky bucket algorithm.

The Leaky Bucket Algorithm

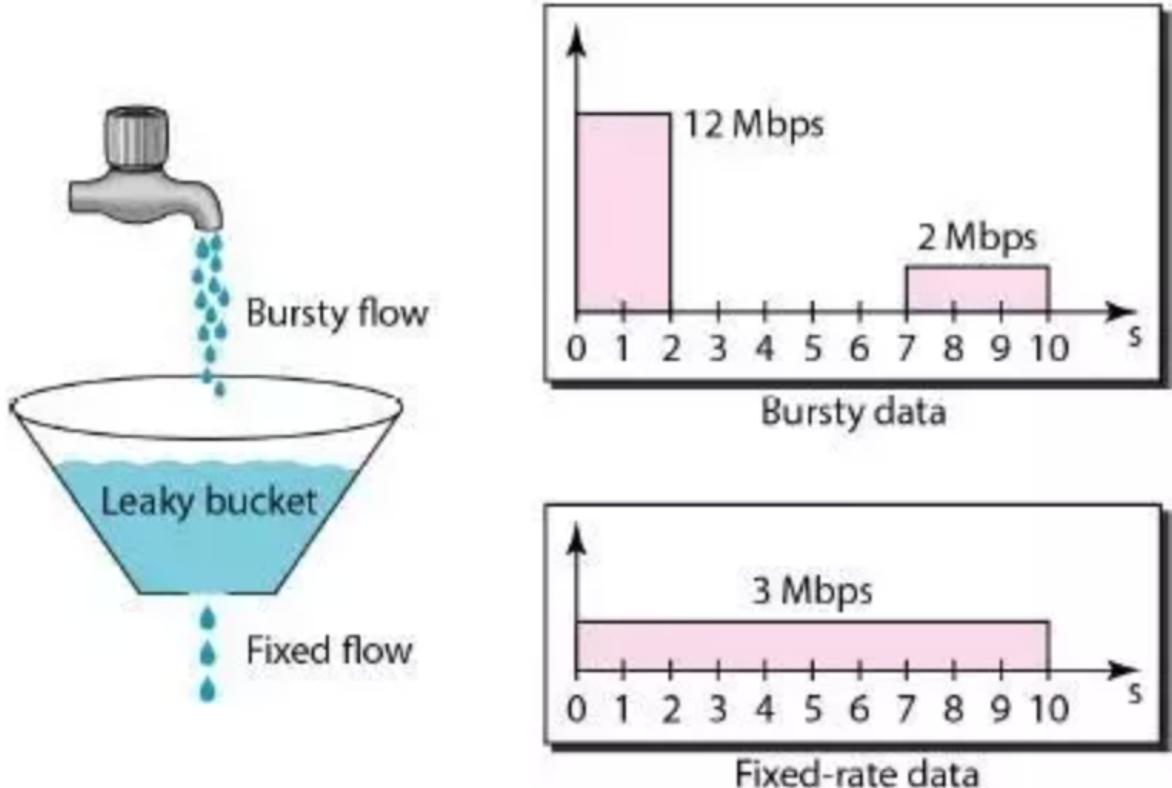


(a) A leaky bucket with water.

(b) a leaky bucket with packets.

The image shows the usage of leaky bucket algorithm in traffic shaping. If we map it our limiting requests to server use case, water drops from the faucets are the requests, the bucket is the **request queue** and the water drops leaked from the bucket are the responses. Just as the water dropping to a full bucket will overflow, the requests arrive after the queue becomes full will be rejected.

As we can see, in the leaky bucket algorithm, the requests are processed at an approximately constant rate, which **smooths out bursts of requests**. Even though the incoming requests can be bursty, the outgoing responses are always at a same rate as shown below.

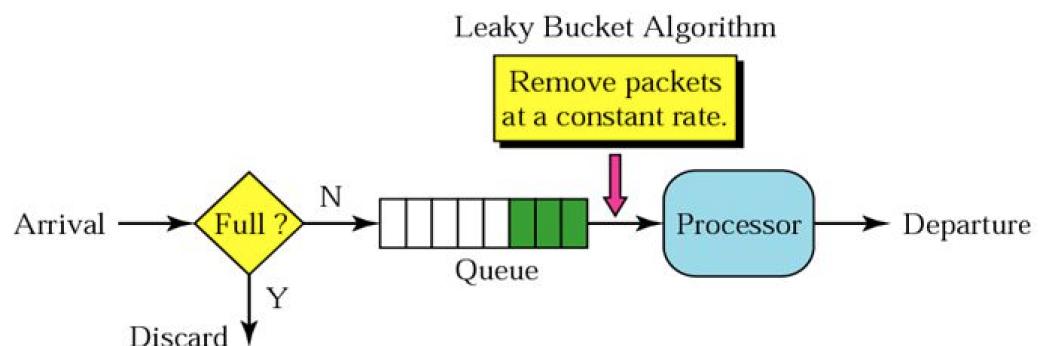


In this algorithm, we use limited sized queue and process requests at a constant rate from queue in First-In-First-Out(FIFO) manner.

For each request a user makes,

- Check if the queue limit is exceeded.
- If the queue limit has exceeded, the request is dropped.
- Otherwise, we add requests to queue end and handle the incoming request.

The above described process is shown in diagram below.



Just like in **Token Bucket**, we will use cache, for example Redis, in **Leaky Bucket** as well.

Code:

```
// RateLimitUsingLeakyBucket .
func RateLimitUsingLeakyBucket(
    userID string,
    uniqueRequestID string,
    intervalInSeconds int64,
    maximumRequests int64) bool {

    // userID can be apikey, location, ip
    requestCount := redisClient.LLen(ctx, userID).Val()

    if requestCount >= maximumRequests {
        // drop request
        return false
    }

    // add request id to the end of request queue
    redisClient.RPush(ctx, userID, uniqueRequestID)

    return true
}
```

14.

In **Leaky Bucket algorithm** bucket size is equal to the queue size. The queue holds the requests to be processed at a fixed rate.

Pros:

- Memory efficient given the limited queue size.
- Requests are processed at a fixed rate, therefore it is suitable for use cases where a stable outflow rate is required.

15.

Cons:

- A burst of traffic fills up the queue with old requests, and if they are not processed in time, recent requests will be rate limited.

16.

Which internet companies use Token Bucket for their rate limiting?

- [Shopify](#)

17.

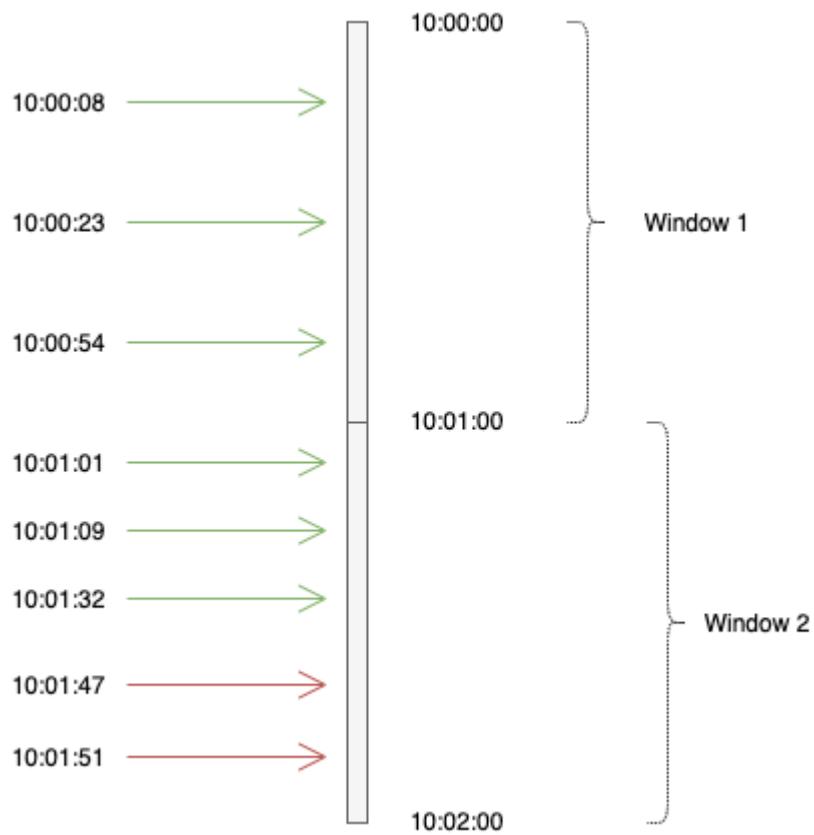
18. Fixed Window Counter algorithm

Fixed window is one of the most basic rate limiting mechanisms and probably the simplest technique for implementing rate limiting.

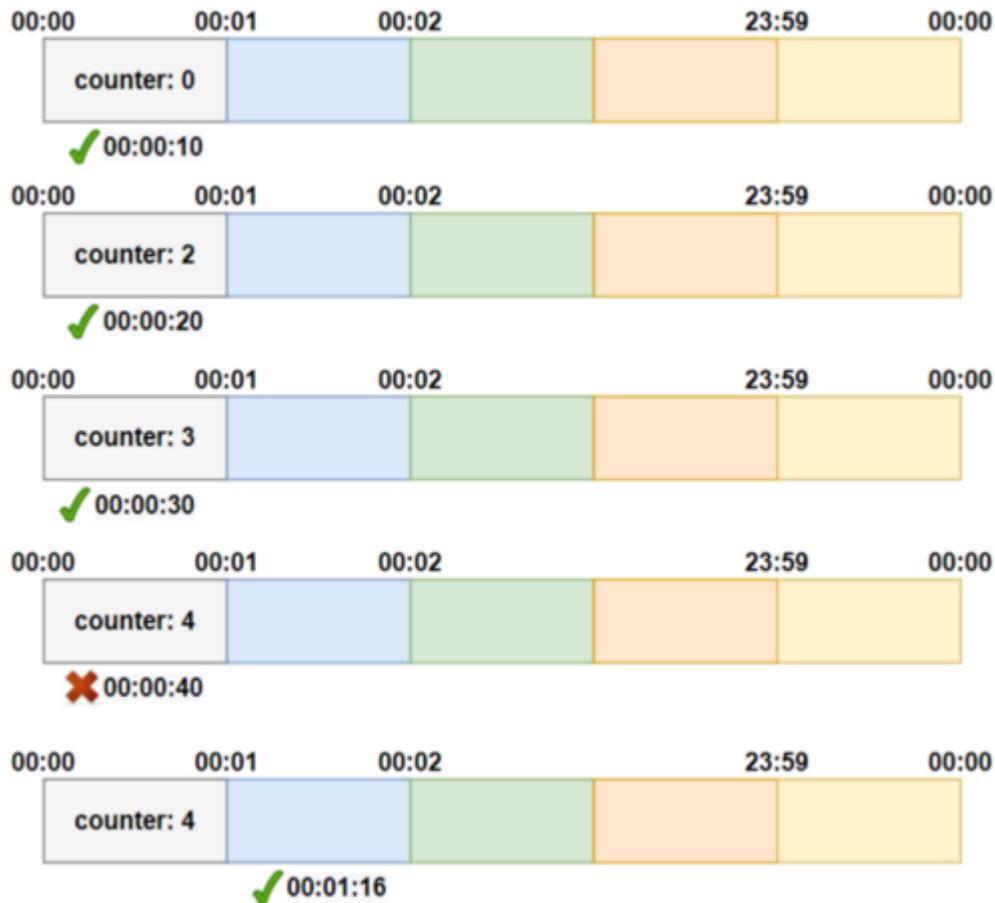
We keep a counter for a given duration of time, and keep incrementing it for every request we get. Once the limit is reached, we drop all further requests till the time duration is reset.

Below two images illustrates how it works when the limit is 3 requests per minute:

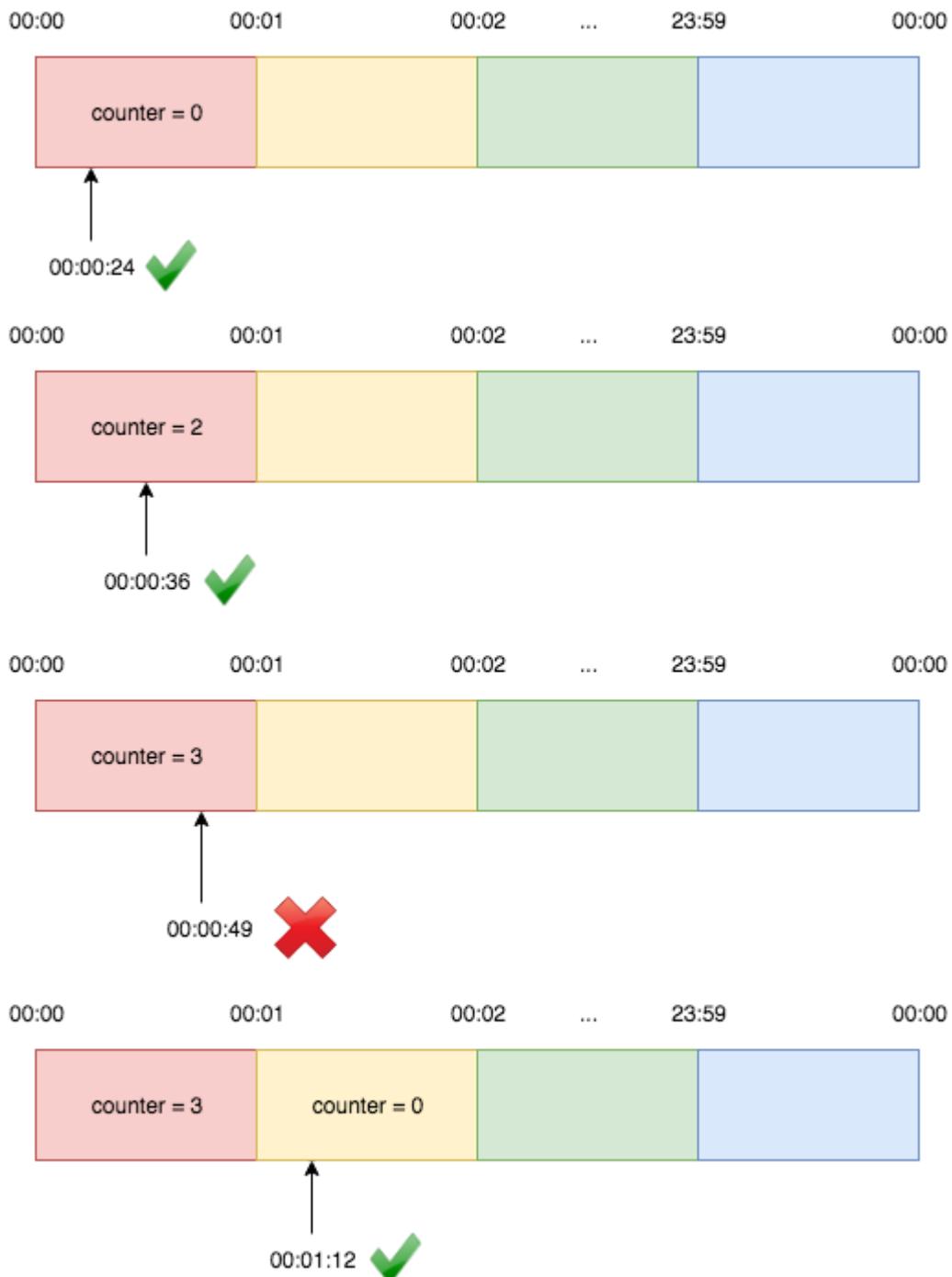
1sr representation:



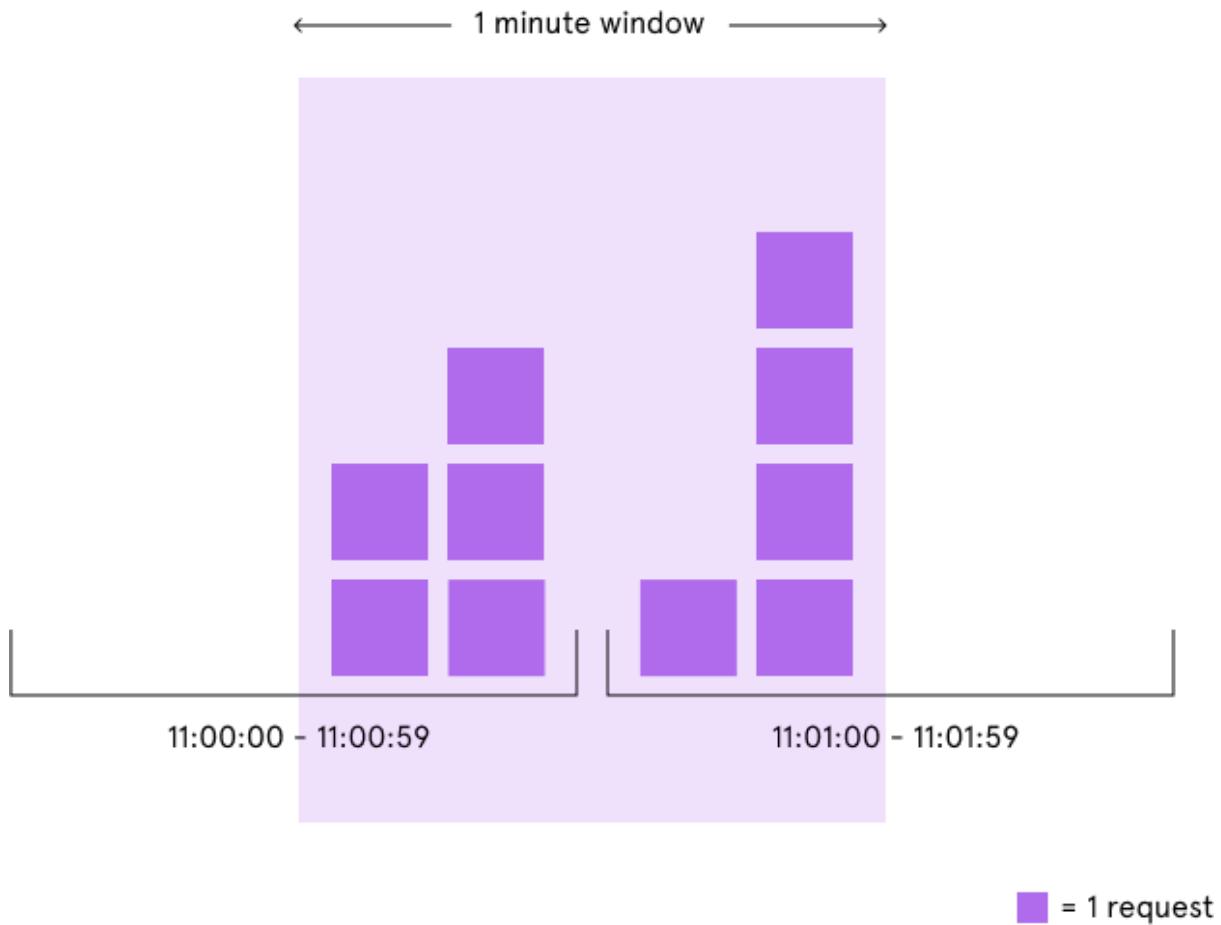
2nd representation:



Fixed window counter algorithm divides the timeline into fixed-size windows and assign a counter to each window. Each request, based on its **arriving time**, is mapped to a window. If the counter in the window has reached the limit, requests falling in this window should be rejected. For example, if we set the window size to 1 minute. Then the windows are [00:00, 00:01), [00:01, 00:02), ...[23:59, 00:00). Suppose the limit is 2 requests per minute:



The advantage here is that it ensures that most recent requests are served without being starved by old requests. However, there is a major problem with this algorithm. Although the fixed window approach offers a straightforward mental model, it can sometimes let through twice the number of allowed requests per minute. For example, if our rate limit were 5 requests per minute and a user made 5 requests at 11:00:59, they could make 5 more requests at 11:01:00 because a new counter begins at the start of each minute. Despite a rate limit of 5 requests per minute, we've now allowed 10 requests in less than one minute!



We could avoid this issue by adding another rate limit with a smaller threshold and shorter enforcement window—e.g. 2 requests per second in addition to 5 requests per minute—but this would overcomplicate the rate limit. Arguably, it would also impose too severe of a restriction on how often the user could make requests.

Implementation

Unlike the token bucket algorithm, this approach's Redis operations are atomic. Each request would increment a Redis key that included the request's timestamp. A given Redis key might look like this:

key	value
"user_1_1490868000":	1

The above indicates that User 1 has made 1 request between 10:00:00 AM GMT and 10:00:59 GMT on Thursday, March 30, 2017. **Notice that in Redis key contains timestamp for 10:00:00 AM GMT on Thursday, March 30, 2017 for all**

entries in between 10:00:00 AM GMT and 10:00:59 GMT on Thursday, March 30, 2017. For example, Redis key for requests made at 10:00:10 AM GMT, March 30, 2017 and 10:00:20 AM GMT, March 30, 2017 will both timestamp for 10:00:00 AM GMT, March 30, 2017. This is what makes the Redis operation atomic.

When incrementing the request count for a given timestamp, we would compare its value to our rate limit to decide whether to reject the request. We would also tell Redis to expire the key when the current minute passed to ensure that stale counters didn't stick around forever.

Fixed window counters with 2 requests/min rate limit

Request at 10:00:07 AM

- ⊕ Increment key with 10:00 AM timestamp
- ✓ Allow request

```
"user_1_1490868000": 1
```

Request at 10:00:38 AM

- ⊕ Increment key with 10:00 AM timestamp
- ✓ Allow request

```
"user_1_1490868000": 2
```

Request at 10:02:15 AM

- ✗ Key with 10:00 AM timestamp expires
- ⊕ Increment key with 10:02 AM timestamp
- ✓ Allow request

```
"user_1_1490868000": 2
```

```
"user_1_1490868120": 1
```

Request at 10:02:24 AM

- ⊕ Increment key with 10:02 AM timestamp
- ✓ Allow request

```
"user_1_1490868120": 2
```

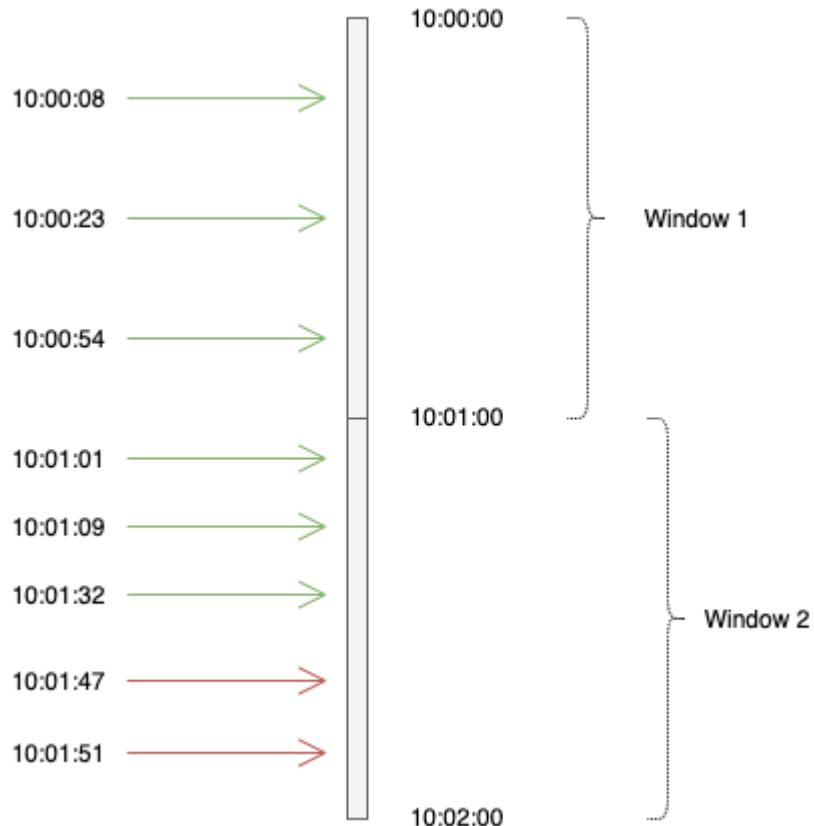
Request at 10:02:36 AM

- ⊕ Increment key with 10:02 AM timestamp
- ✗ Reject request

```
"user_1_1490868120": 3
```

In the below example, from the top, requests 1–3 belong to window 1, hence tracked against the same key in Redis (e.g. 12345_21042020_10:00:00 or 12345_1587463200 using epoch). Since there are only 3 requests in this window, no

request is blocked.



Algorithm:

For each request a user makes,

- Check if the user has exceeded the limit in the current window.
- If the user has exceeded the limit, the request is dropped
- Otherwise, we increment the counter

```
// RateLimitUsingFixedWindow .
func RateLimitUsingFixedWindow(userID string, intervalInSeconds int64, maximumRequests int64) bool {
    // userID can be apikey, location, ip
    currentWindow := strconv.FormatInt(time.Now().Unix()/intervalInSeconds, 10)
    key := userID + ":" + currentWindow // user userID + current time window
    // get current window count
    value, _ := redisClient.Get(ctx, key).Result()
    requestCount, _ := strconv.ParseInt(value, 10, 64)
    if requestCount >= maximumRequests {
        // drop request
    }
}
```

```

        return false
    }

    // increment request count by 1
    redisClient.Incr(ctx, key) // if the key is not available, value is initialised to 0 and
incremented to 1

    // handle request
    return true
    // delete all expired keys at regular intervals
}

```

19.

At regular intervals, say every 10 min or every hour, delete all the expired window keys (instead of setting the current window to expire at the start next window using EXPIRE command in Redis, which is another command/load on Redis for every request).

Pros:

- Memory efficient.
- Easy to understand.
- Best for use cases where quota is reset only at the end of the unit time window. For example: if the limit is 10 requests / min, it allows 10 requests in every unit minute window say from 10:00:00 AM to 10:00:59 AM, and the quota resets at 10:01:00 AM. It does not matter if 20 requests were allowed in between 10:00:30 AM and 10:01:29 AM, since 10:00:00 AM to 10:00:59 AM is one slot and 10:01:00 AM to 10:01:59 AM is another slot, even though 20 requests were allowed in last one minute at 10:01:30 AM. **This is why this algorithm is called Fixed Window and not Sliding Window.**

20. Cons:

- Spike in traffic at the edge of a window makes this algorithm unsuitable for use cases where time window needs to be tracked real-time at all given time. Example 1: if we set a maximum of 10 message per minute, we don't want a user to be able to receive 10 messages at 0:59 and 10 more messages at 1:01. Example 2: if the limit is 10 requests / min and 10 requests were sent starting from 10:00:30 AM to 10:00:59 AM, then no requests will be allowed till 10:01:29 AM and quota will be reset only at 10:01:29 AM since 10 requests

were already sent in last 1 min starting at 10:00:30 AM.

21.

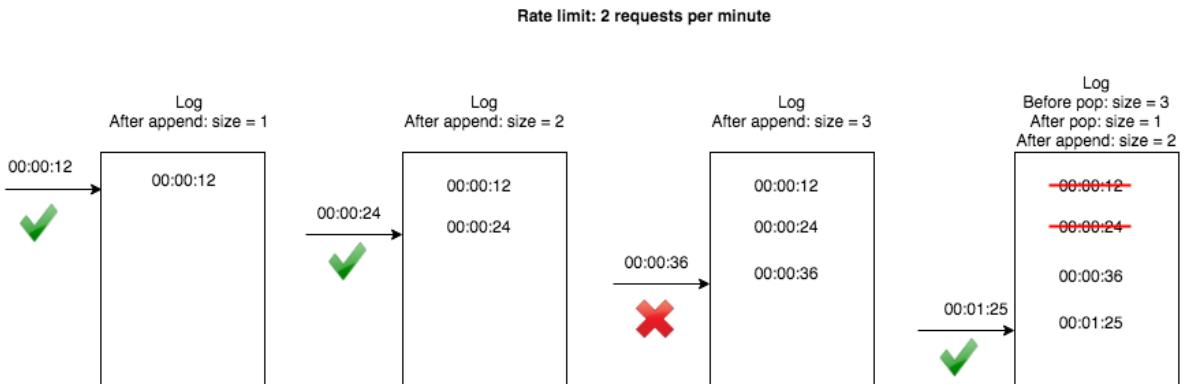
22. Sliding Window Logs algorithm

As discussed above, the **fixed** window counter algorithm has a major drawback: it allows more requests to go through at the edge of a window. The **sliding window logs algorithms** fixes this issue. It works as follows:

- The algorithm keeps track of request timestamps. Timestamp data is usually kept in cache, such as [sorted sets of Redis](#).
- When a new request comes in, remove all the outdated timestamps. Outdated timestamps are defined **as those older than the start of the current time window**.
Storing the timestamps in sorted order in sorted set enables us to efficiently find the outdated timestamps.
- Add timestamp of the new request to the log.
- If the log size is the same or lower than the allowed count, a request is accepted. Otherwise, it is rejected.

23.

Let's see this algorithm in action below:



Implementation:

Whenever request arrives, Sliding Window Logs algorithm would insert a new member into the sorted set with a sort value of the Unix microsecond timestamp. This would allow us to **efficiently remove all of the set's members with outdated timestamps** and count the size of the set afterward. The sorted set's size would then be equal to the number of requests in the most recent sliding window of time.

Fixed window counters with 2 requests/min rate limit

Request at 10:00:07 AM

- ⊕ Increment key with 10:00 AM timestamp
- ✓ Allow request

"user_1_1490868000": 1

Request at 10:00:38 AM

- ⊕ Increment key with 10:00 AM timestamp
- ✓ Allow request

"user_1_1490868000": 2

Request at 10:02:15 AM

- ✗ Key with 10:00 AM timestamp expires
- ⊕ Increment key with 10:02 AM timestamp
- ✓ Allow request

"user_1_1490868000": 2

"user_1_1490868120": 1

Request at 10:02:24 AM

- ⊕ Increment key with 10:02 AM timestamp
- ✓ Allow request

"user_1_1490868120": 2

Request at 10:02:36 AM

- ⊕ Increment key with 10:02 AM timestamp
- ✗ Reject request

"user_1_1490868120": 3

Algorithm:

For each request a user makes,

- Check if the user has exceeded the limit in the current window by getting the count of all the logs in the last window in the sorted set. If current time is 10:00:27 and rate is 100 req/min, get logs from previous window(10:00:27–60= 09:59:28) to current time(10:00:27).
 - If the user has exceeded the limit, the request is dropped.
 - Otherwise, we add the unique request ID (you can get from the request or you can generate a unique hash with userID and time) to the sorted sets(sorted by time).

24.

For people familiar with Redis, below is how we can implement the above algorithm:

Below is what the above code does.

For each request (assuming 1 min rate-limiting window):

- Remove elements from the set which are older than 1 min using ZRemRangeByScore. This leaves only elements (requests) that occurred in the last 1 minute.
- Add current request timestamp using ZAdd.
- Fetch all elements of the set using ZRange(0, -1) and check the count against rate-limit value and accept/reject the request accordingly.
- Update sorted set TTL with each update so that it's cleaned up after being idle for some time.

All of these operations can be performed atomically using MULTI command, so this rate limiter can be shared by multiple processes.

We need to be careful while choosing the timestamp granularity (second, millisecond, microsecond). Sorted set, being a set, doesn't allow duplicates. So if we get multiple requests at the same timestamp, they will be counted as one.

```
// RateLimitUsingSlidingLogs .
func RateLimitUsingSlidingLogs(userID string, uniqueRequestID string, intervalInSeconds int64, maximumRequests int64) bool {
    // userID can be apikey, location, ip
    currentTime := strconv.FormatInt(time.Now().Unix(), 10)
```

```

lastWindowTime := strconv.FormatInt(time.Now().Unix()-intervalInSeconds, 10)
// get current window count
requestCount := redisClient.ZCount(ctx, userID, lastWindowTime, currentTime).Val()
if requestCount >= maximumRequests {
    // drop request
    return false
}

// add request id to last window
redisClient.ZAdd(ctx, userID, &redis.Z{Score: float64(time.Now().Unix()), Member: uniqueRequestID}) // add unique request id to userID set with score as current time

// handle request
return true
// remove all expired request ids at regular intervals using ZRemRangeByScore from -inf to last window time
}

```

25.

Pros:

Sliding Window Logs algorithm works flawlessly. Rate limiting implemented by this algorithm is very accurate. In rolling window, requests will not exceed the rate limit.

Cons:

Sliding Window Logs algorithm consumes a lot of memory because even if a request is rejected , it's timestamp will still be stored in memory.

26. Sliding Window Counter algorithm

The Sliding Window Counter algorithm is a hybrid approach that combines the **Fixed Window Counter algorithm** and **Sliding Window Logs algorithm**.

This algorithm is often used to stop the spammers very efficiently.

Sliding Window Counter algorithm can be implemented by two different approaches as described below:

Approach #1:

Let's take an example: Say we are building a rate limiter of 100 requests/hour. Say a bucket size of 20 mins is chosen, then there are 3 buckets in the unit time.

For a window time of 2AM to 3AM, the buckets are

```
{  
  "2AM-2:20AM": 10,  
  "2:20AM-2:40AM": 20,  
  "2:40AM-3:00AM": 30  
}
```

If a request is received at 2:50AM, we find out the total requests in last 3 buckets including the current and add them, in this case they sum upto 60 (<100), so a new request is added to the bucket of 2:40AM – 3:00AM giving below:

```
{  
  "2AM-2:20AM": 10,  
  "2:20AM-2:40AM": 20,  
  "2:40AM-3:00AM": 31  
}
```

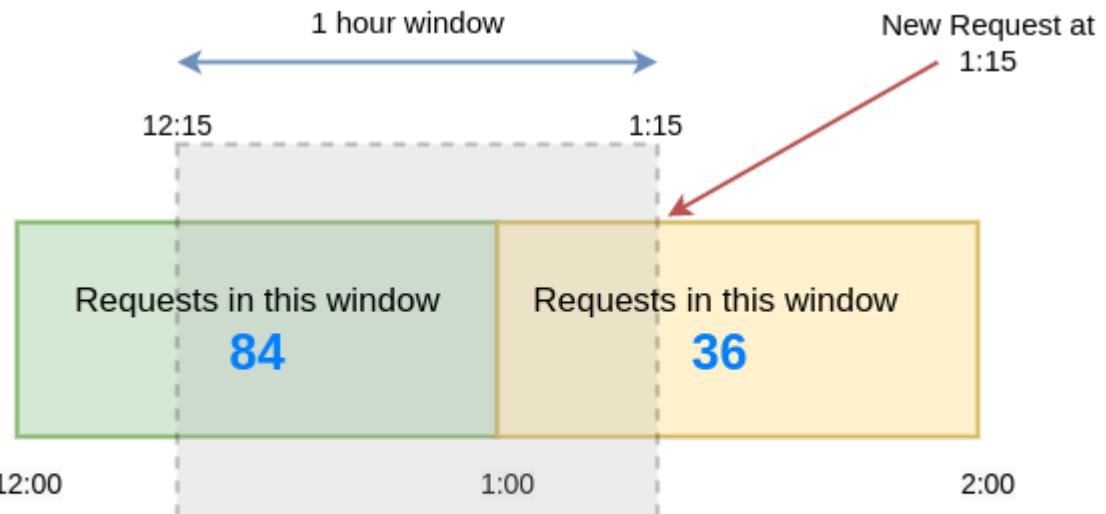
○

Approach #2

In the **sliding** window counter algorithm, instead of **fixed** window size, we have a **rolling** window of time to smooth bursts.

The windows are typically defined by the floor of the current timestamp, so 12:03:15 with a 60-second window length would be in the 12:03:00 window.

Let's say we want to limit 100 requests per hour on an API and assume there are 84 requests in the time window [12:00–1:00) and 36 requests current window [1:00 to 2:00) which started 15 minutes ago.



nlogn.in
Limit = 100 requests/hour

Now imagine a new request arrives at 1:15. To decide, whether we should accept this request or deny it will be based on the approximation.

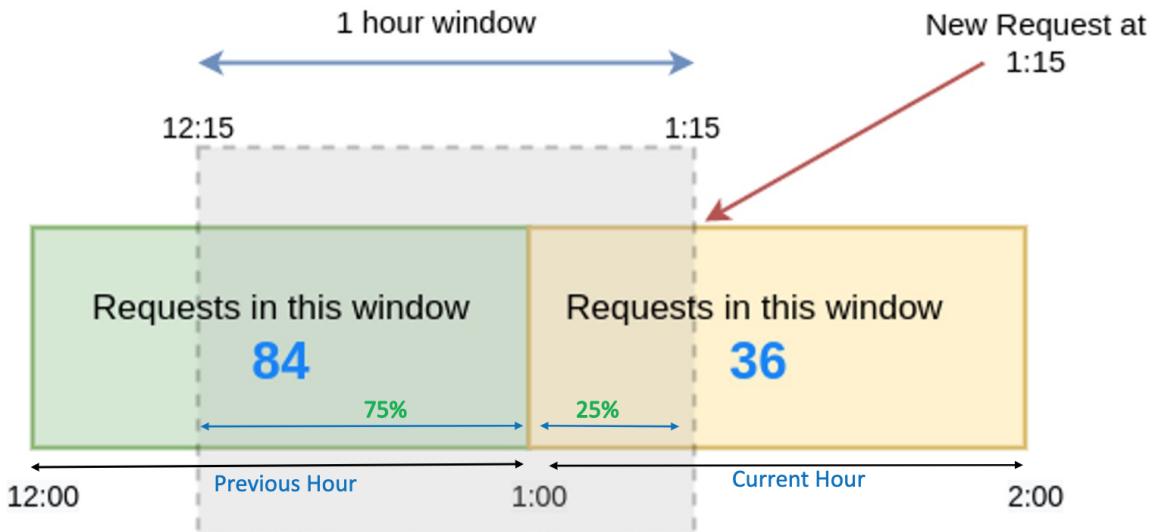
The approximation rate will be calculated like this:

limit = 100 requests/hour

$$\begin{aligned}
 \text{rate} &= (84 * ((\text{time interval between 1:00 and 12:15}) / \text{rolling window size})) + 36 \\
 &= 84 * ((60 - 15)/60) + 36 \\
 &= 84 * 0.75 + 36 \\
 &= 99
 \end{aligned}$$

rate < 100
hence, we will accept this request.

Since the requests in the current window [12:15 – 1:15] are 99 which is less than our limit of 100 requests/hour, hence this request will be accepted. But any new request during the next second will not be accepted.



nlogn.in

Limit = 100 requests/hour

We can explain the above calculation in a more lucid way: Assume the rate limiter allows a maximum of 100 requests per hour, and there are 84 requests in the previous hour and 36 requests in the current hour. For a new request that arrives at a 25% position in the current hour, the number of requests in the **rolling window** is calculated using the following formula:

Requests in current window + (Requests in the previous window * overlap percentage of the rolling window and previous window)

Using the above formula, we get $(36 + (84 * 75\%)) = 99$

Since the rate limiter allows 100 requests per hour, the current request will go through.

This algorithm assumes a constant request rate in the (any) previous window, which is not true as there can be request spikes too during a minute and no request during another hour. Hence the result is only an approximated value.

```
// RateLimitUsingSlidingWindow .
func RateLimitUsingSlidingWindow(userID string, uniqueRequestID string, intervalInSeconds int64, maximumRequests int64) bool {
    // userID can be apikey, location, ip
    now := time.Now().Unix()

    currentWindow := strconv.FormatInt(now/intervalInSeconds, 10)
    key := userID + ":" + currentWindow // user userID + current time window
    // get current window count
```

```

value, _ := redisClient.Get(ctx, key).Result()
requestCountCurrentWindow, _ := strconv.ParseInt(value, 10, 64)
if requestCountCurrentWindow >= maximumRequests {
    // drop request
    return false
}

lastWindow := strconv.FormatInt(((now - intervalInSeconds) / intervalInSeconds), 10)
key = userID + ":" + lastWindow // user userID + last time window
// get last window count
value, _ = redisClient.Get(ctx, key).Result()
requestCountlastWindow, _ := strconv.ParseInt(value, 10, 64)

elapsedTimePercentage := float64(now%intervalInSeconds) /
float64(intervalInSeconds)

// last window weighted count + current window count
if
(float64(requestCountlastWindow)*(1-elapsedTimePercentage))+float64(requestCountCurre
ntWindow) >= float64(maximumRequests) {
    // drop request
    return false
}

// increment request count by 1 in current window
redisClient.Incr(ctx, userID+":"+currentWindow)

// handle request
return true
}

```

○

Pros:

- Memory efficient.
- It smoothes out spikes in the traffic because the rate is based on the average rate of the previous window.

○

Cons:

- It only works for not-so-strict look back window. It is an approximation of the actual rate because it assumes requests in the previous window are evenly distributed.

However, this problem may not be as bad as it seems. [According to experiments done by Cloudflare, only 0.003% of requests are wrongly allowed or rate limited among 400 million requests.](#)

Space Optimization

If you are using Redis, use [Redis hash](#) to optimize space. This space optimization applies to all the algorithms described above.

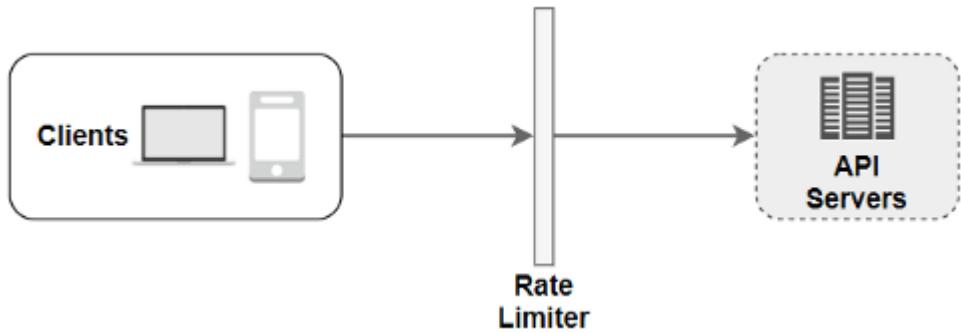
Where to put the Rate Limiter ?

A rate limiter can be implemented on either the client or server-side.

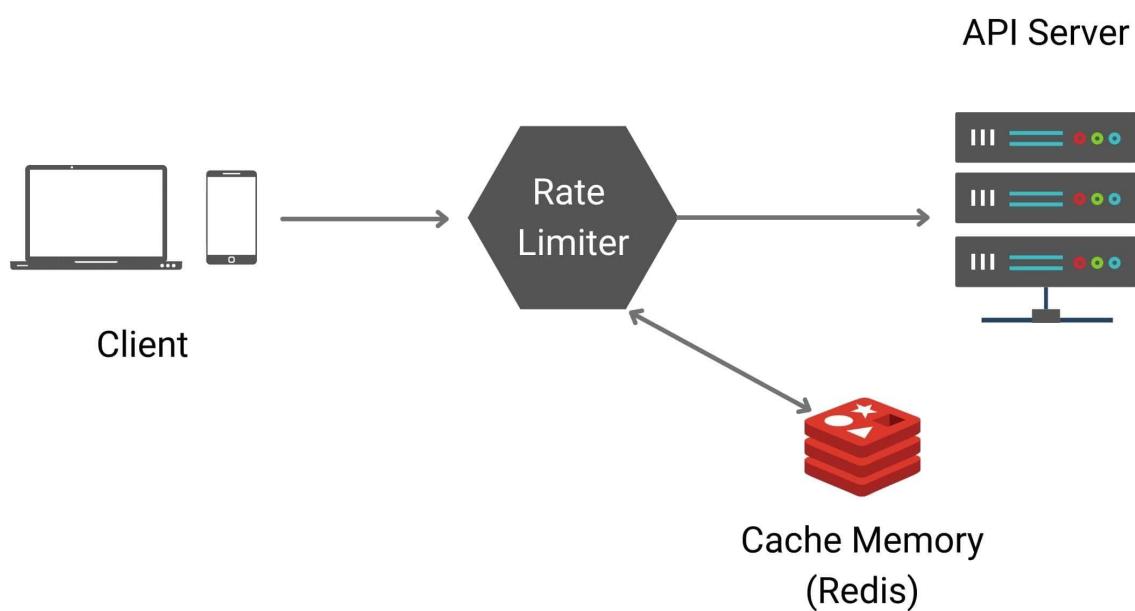
- Client-side implementation: In general, client side is not the ideal place to implement rate limiter for most use cases, since a malicious actors can manipulate the client-side code.
- Server-side implementation: On the server-side, a rate limiter is shown in the diagram below.



There is an alternative and much better implementation. We can construct a rate limiter **middleware**, which throttles requests, whenever appropriate, before the request can even hit your APIs, rather than establishing a rate limiter on the API servers, as indicated in the diagram below.



Now since we already know by now that we will be needing cache, below is a more complete architecture diagram.



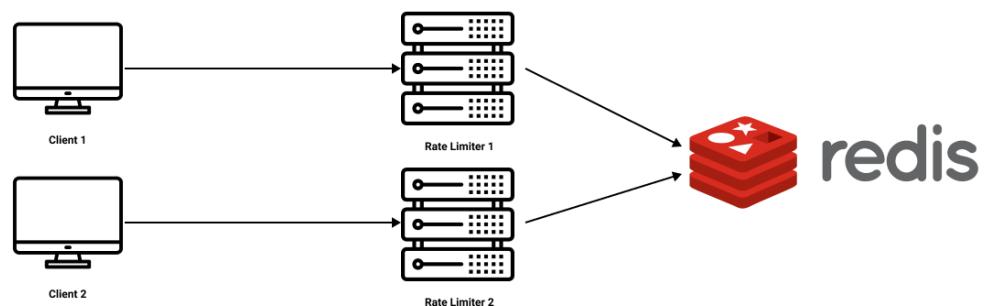
In general, having a separate rate limiter middleware is a better architecture because:

1. that way it is decoupled from the other components and microservice, for example your APIs. This helps us to scale our rate limiter middleware independently of all other components.
2. We might have than one microservices exposing more than one APIs that we want to rate limit. In that case we do not need to implement rate limiter for each of these microservices. This one rate limiter middleware will be able to rate limit all the APIs. This also avoids code duplication.

Distributed Rate Limiter

To support millions of users, one rate limiter server might not be enough to handle the traffic. When multiple rate limiter servers are used, synchronization of data becomes an overhead. How will one rate limiter server know about how much quota is left for a customer that just got served by the other rate limiter server(s) ? There are two approaches to handle this:

1. One possible solution is to use **Sticky Session** that allows a client to send traffic to the same rate limiter. This solution is not advisable because this design option is neither scalable nor flexible.
2. A better approach is to use [centralized data stores like Redis](#).



Monitoring:

After the rate limiter is put in place, it is important to gather analytics data to check whether the implemented rate limiter is effective. Primarily, we are interested in measuring or assessing that the rate limiting algorithm and rate limiting rules are effective and working as expected.

How are most companies implement rate limiting in real-world projects?

While there are companies like [Lyft](#), [Shopify](#), [Stripe](#), [Amazon](#) which implement their own rate limiter, most companies leverage rate limiting services that are already available out there, like below:

- API Gateway ([Kong](#), [Azure API Management](#), [AWS API Gateway](#), etc.)
- Reverse Proxy like [Ngin](#)

Types of Throttling

1. **Hard Throttling:** The number of API requests cannot exceed the throttle limit.
2. **Soft Throttling:** In this type, we can set the API request limit to exceed a certain percentage. For example, if we have rate-limit of 100 messages a minute and 10% exceed-limit, our rate limiter will allow up to 110 messages per minute.
3. **Elastic or Dynamic Throttling:** Under Elastic throttling, the number of requests can go beyond the threshold if the system has some resources available. For example, if

a user is allowed only 100 messages a minute, we can let the user send more than 100 messages a minute when there are free resources available in the system.

Few Other Things to Keep in Mind:

- If you are consuming an API which is rate limiting, you need to implement proper [Retry mechanism](#) in place. Whenever you get **429 Too Many Requests** Http Status Code in the API response you should queue up the request to retry and process later.

Use Cases where Client-Side Rate Limiting is Necessary:

It might seem that rate limiting is solely in the API providers' interest. In practice, the users or client of the service could also benefit from rate limiting. Clients who adhere to usage restrictions can always be assured that the service provider will fulfil their requests. If the user does not take the rate limiting into account, then they may have to face some sudden or unexpected rejection of their requests, which can affect the functionality of their solutions.

For example, a mobile application might be built upon the Facebook messaging API. When people use this application, they are actually using Facebook messaging in the background. In this case, it is the client who must take steps to ensure the application adheres to the rate limiting norms. If the rate limiting is overlooked at the client-side, users of this mobile application might face unexpected errors. It is in the best interest of API users to enforce rate limiting at the client-side.

No rate limiting

It is important to consider the option of performing no rate limiting as a floor in your design—that is, as a worst-case situation that your system must be able to accommodate. Build your system with robust error handling in case some part of your rate-limiting strategy fails, and understand what users of your service will receive in those situations. Ensure that you provide useful error codes and that you don't leak any sensitive data in error codes. Using timeouts, deadlines, and circuit-breaking patterns helps your service to be more robust in the absence of rate limiting.

System Design Interview

By now you must have gained deep knowledge and understanding of rate limiting. If you were a beginner, you can safely consider yourself an intermediate-to-advanced learner now as far as designing rate limiter and similar scalable distributed planet-scale systems are concerned. We have covered all the basics as well as advanced concepts and architecture design along with some coding and implementation. Leveraging the content covered in this chapter you should be easily able to build your own version of scalable rate limiter on public cloud like Microsoft Azure, Amazon Web Services, Google Cloud Platform, Oracle Cloud Infrastructure. **This is a huge advantage when you go to an interview.** Most candidates go to System Interview interview with just theoretical knowledge of system design by reading about common System Design interview questions. They blurt out buzzwords without knowing how they are actually implemented, without having deep understanding of them. By the end of the interview, this eventually becomes clear to the interviewer, and these candidates often end up not getting an offer for **senior position**. What they lack is

hands-on experience and consequently knowing how systems are actually built in tech companies.

Your practical experience and knowledge will show up eventually in your interview performance and will help you stand outside of the crowd. I want you to walk confidently into the interview room (virtual or in-person) and prove your potential and deep knowledge.

Remember what is most important in a System Design interview is to ask clarifying questions and making no assumptions. If you are making any assumptions, ask your interviewer if that assumption is valid for the system she/he has in mind for you to build.

If you are asked to design rate limiter, one of the first few questions you should ask is whether the rate limiter that your interviewer wants you to design is whether the rate limiting is done based on **rolling** window or **fixed** window, because based on that you would be able to propose the algorithms that would give efficient result.

Make sure you discuss all the trade-offs.

Also discuss how large of a traffic the system will be getting so that you can propose right scale-out strategy.

Solution 5:

A rate limiter restricts the intended or unintended excessive usage of a system by regulating the number of requests made to/from it by discarding the surplus ones. In this article, we dive deep into an intuitive and heuristic approach for rate-limiting that uses a sliding window. The other algorithms and approaches include [Leaky Bucket](#), [Token Bucket](#) and Fixed Window.

Rate limiting is usually applied per access token or per user or per region/IP. For a generic rate-limiting system that we intend to design here, this is abstracted by a configuration key key on which the capacity (limit) will be configured; the key could hold any of the aforementioned value or its combinations. The limit is defined as the number of requests number_of_requests allowed within a time window time_window_sec (defined in seconds).

The algorithm

The algorithm is pretty intuitive and could be summarized as follow

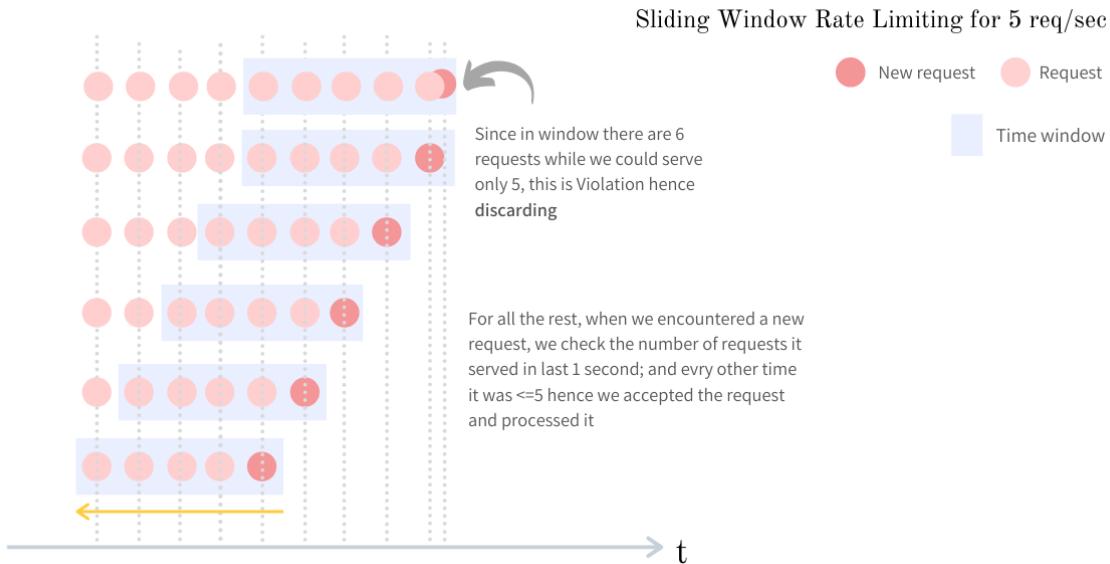
If the number of requests served on configuration key key in the last time_window_sec seconds is more than number_of_requests configured for it then discard, else the request goes through while we update the counter.

Although the above description of the algorithm looks very close to the core definition of any rate limiter, it becomes important to visualize what is happening here and implement it in an extremely efficient and resourceful manner.

Visualizing sliding window

Every time we get a request, we make a decision to either serve it or not; hence we check the number_of_requests made in last time_window_sec seconds. So this process of

checking for a fixed window of time_window_sec seconds on every request, makes this approach a sliding window where the fixed window of size time_window_sec seconds is moving forward with each request. The entire approach could be visualized as follows



The pseudocode

The core of the algorithm could be summarized in the following Python pseudocode. It is not recommended to put this or similar code in production as it has a lot of limitations (discussed later), but the idea here is to design the rate limiter ground up including low-level data models, schema, data structures, and a rough algorithm.

```
def is_allowed(key:str) -> Bool:
    """The function decides if the current request should be served or not.
    It accepts the configuration key `key` and checks the number of requests made against it
    as per the configuration.
```

The function returns True if the request goes through and False otherwise.

```
"""
current_time = int(time.time())

# Fetch the configuration for the given key
# the configuration holds the number of requests allowed in a time window.
config = get_ratelimit_config(key)

# Fetch the current window for the key
# The window returned, holds the number of requests served since the start_time
# provided as the argument.
start_time = current_time - config.time_window_sec
window = get_current_window(key, start_time)

if window.number_of_requests > config.capacity:
```

```

return False

# Since the request goes through, register it.
register_request(key, current_time)
return True

```

A naive implementation of the above pseudocode is trivial but the true challenge lies in making the implementation horizontally scalable, with low memory footprint, low CPU utilization, and low time complexity.

Design

Designing a rate limiter has to be super-efficient because the rate limiter decision engine will be invoked on every single request and if the engine takes a long time to decide this, it will add some overhead in the overall response time of the request. A better design will not only help us keep the response time to a bare minimum, but it also ensures that the system is extensible with respect to future requirement changes.

Components of the Rate limiter

The Rate limiter has the following components

- Configuration Store - to keep all the rate limit configurations
- Request Store - to keep all the requests made against one configuration key
- Decision Engine - it uses data from the Configuration Store and Request Store and makes the decision

Deciding the datastores

Picking the right data store for the use case is extremely important. The kind of datastore we choose determines the core performance of a system like this.

Configuration Store

The primary role of the Configuration Store would be to

- efficiently store configuration for a key
- efficiently retrieve the configuration for a key

In case of machine failure, we would not want to lose the configurations created, hence we choose a disk-backed data store that has an efficient get and put operation for a key. Since there would be billions of entries in this Configuration Store, using a SQL DB to hold these entries will lead to a performance bottleneck and hence we go with a simple key-value NoSQL database like [MongoDB](#) or [DynamoDB](#) for this use case.

Request Store

Request Store will hold the count of requests served against each key per unit time. The most frequent operations on this store will be

- registering (storing and updating) requests count served against each key - *write heavy*
- summing all the requests served in a given time window - *read and compute heavy*
- cleaning up the obsolete requests count - *write heavy*

Since the operations are both read and write-heavy and will be made very frequently (on every request call), we chose an in-memory store for persisting it. A good choice for such operation will be a datastore like [Redis](#) but since we would be diving deep with the core implementation, we would store everything using the common data structures available.

Data models and data structures

Now we take a look at data models and data structures we would use to build this generic rate limiter.

Configuration Store

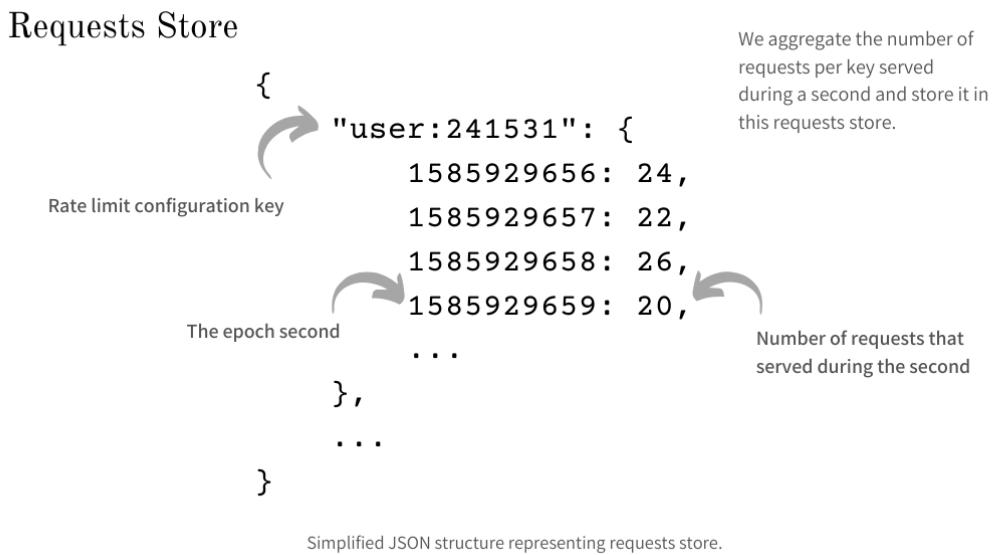
As decided before we would be using a NoSQL key-value store to hold the configuration data. In this store, the key would be the configuration key (discussed above) which would identify the user/IP/token or any combination of it; while the value will be a tuple/JSON document that holds `time_window_sec` and `capacity` (limit).

```
{  
  "user:241531": {  
    "time_window_sec": 1,  
    "capacity": 5  
  }  
}
```

The above configuration defines that the user with id 241531 would be allowed to make 5 requests in 1 second.

Request Store

Request Store is a nested dictionary where the outer dictionary maps the configuration key key to an inner dictionary, and the inner dictionary maps the epoch second to the request counter. The inner dictionary is actually holding the number of requests served during the corresponding epoch second. This way we keep on aggregating the requests per second and then sum them all during aggregation to compute the number of requests served in the required time window.



Implementation

Now that we have defined and designed the data stores and structures, it is time that we implement all the helper functions we saw in the pseudocode.

Getting the rate limit configuration

Getting the rate limit configuration is a simple get on the Configuration Store by key. Since the information does not change often and making a disk read every time is expensive, we cache the results in memory for faster access.

```

def get_ratelimit_config(key):
    value = cache.get(key)

    if not value:
        value = config_store.get(key)
        cache.put(key, value)

    return value
  
```

Getting requests in the current window

Now that we have the configuration for the given key, we first compute the `start_time` from which we want to count the requests that have been served by the system for the key. For this, we iterate through the data from the inner dictionary second by second and keep on summing the requests count for the epoch seconds greater than the `start_time`. This way we get the total requests served from `start_time` till now.

In order to reduce the memory footprint, we could delete the items from the inner dictionary against the time older than the start_time because we are sure that the requests for a timestamp older than start_time would never come in the future.

```
def get_current_window(key, start_time):
    ts_data = requests_store.get(key)
    if not key:
        return 0

    total_requests = 0
    for ts, count in ts_data.items():
        if ts > start_time:
            total_requests += count
        else:
            del ts_data[ts]

    return total_requests
```

Registering the request

Once we have validated that the request is good to go through, it is time to register it in the store and the defined function register_request does exactly that.

```
def register_request(key, ts):
    store[key][ts] += 1
```

Potential issues and performance bottlenecks

Although the above code elaborates on the overall low-level implementation details of the algorithm, it is not something that we would want to put in production as there are lots of improvements to be made.

Atomic updates

While we register a request in the Request Store we increment the request counter by 1. When the code runs in a multi-threaded environment, all the threads executing the function for the same key key, all will try to increment the same counter. Thus there will be a classical problem where multiple writers read the same old value and updates. To fix this we need to ensure that the increment is done atomically and to do this we could use one of the following approaches

- optimistic locking (compare and swap)
- pessimistic locks (always taking lock before incrementing)
- utilize atomic hardware instructions (fetch-and-add instruction)

Accurately computing total requests

Since we are deleting the keys from the inner dictionary that refers to older timestamps (older than the start_time), it is possible that a request with older start_time is executing while a request with newer start_time deleted the entry and lead to incorrect total_request calculation. To remedy this we could either

- delete entries from the inner dictionary with a buffer (say older than 10 seconds before the start_time),
- take locks while reading and block the deletions

Non-static sliding window

There would be cases where the time_window_sec is large - an hour or even a day, suppose it is an hour, so if in the Request Store we hold the requests count against the epoch seconds there will be 3600 entries for that key and on every request, we will be iterating over at least 3600 keys and computing the sum. A faster way to do this is, instead of keeping granularity at seconds we could do it at the minute-level and thus we sub-aggregate the requests count at per minute and now we only need to iterate over about 60 entries to get the total number of requests and our window slides not per second but per minute.

The granularity configuration could be persisted in the configuration as a new attribute which would help us take this call.

Other improvements

The solution described above is not the most optimal solution but it aims to prove a rough idea on how we could implement a sliding window rate limiting algorithm. Apart from the improvements mentioned above there some approaches that would further improve the performance

- use a data structure that is optimized for range sum, like segment tree
- use a running aggregation algorithm that would prevent from recomputing redundant sums

Scaling the solution

Scaling the Decision engine

The decision engine is the one making the call to each store to fetch the data and taking the call to either accept or discard the request. Since decision engine is a typical service engine we would put it behind a load balancer that takes care of distributing requests to decision engine instances in a round-robin fashion ensuring it scales horizontally.

The scaling policy of the decision engine will be kept on following metrics

- number of requests received per second
- time taken to make a decision (response time)
- memory consumption
- CPU utilization

Scaling the Request Store

Since the Request Store is doing all the heavy lifting and storing a lot of data in memory, this would not scale if kept on a single instance. We would need to horizontally scale this system and for that, we shard the store using configuration key and use consistent hashing to find the machine that holds the data for the key.

To facilitate sharding and making things seamless for the decision engine we will have a Request Store proxy which will act as the entry point to access Request Store data. It will abstract out all the complexities of distributed data, replication, and failures.

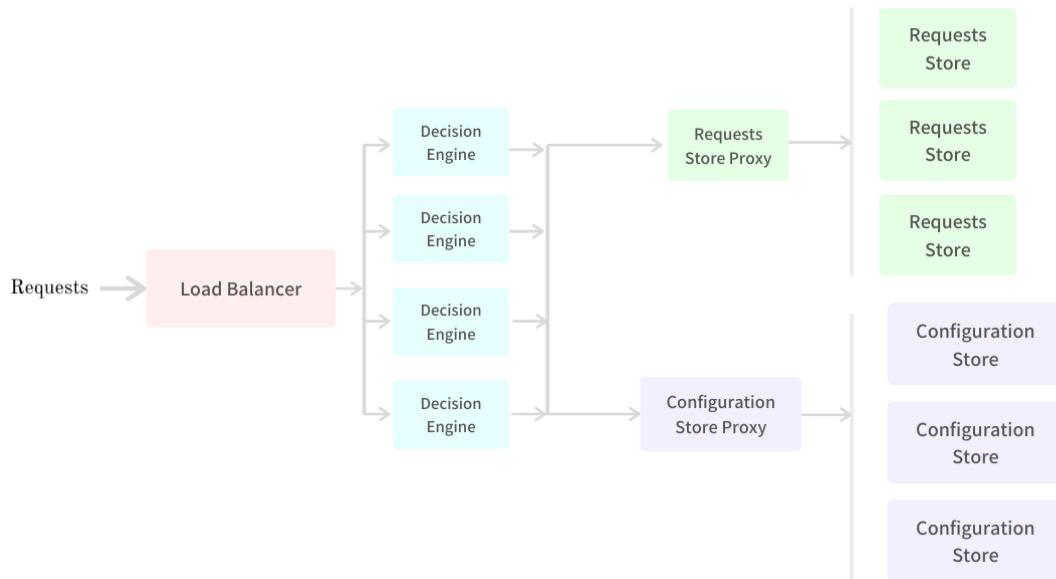
Scaling the Configuration Store

The number of configurations would be high but it would be relatively simple to scale since we are using a NoSQL solution, sharding on configuration key would help us achieve horizontal scalability.

Similar to Request Store proxy we will have a proxy for Configuration Store that would be an abstraction over the distributed Configuration Stores.

High-level design

The overall high-level design of the entire system looks something like this



Deploying in production

While deploying it to production we could use a memory store like Redis whose features, like Key expiration, transaction, locks, sorted, would come in handy. The language we chose for explaining and pseudocode was Python but in production to make things super-fast and concurrent we would prefer a language like Java or Golang. Picking this stack will keep our server cost down and would also help us make optimum utilization of the resources.

Solution 6:

What is Rate Limiting?

Rate limiting is a simple concept to grab, but it's vital for large-scale systems. It is used for security purposes as well as performance improvement. For example, a service can serve a limited number of requests per second. However, if a service is receiving a huge number of requests at a time, it might be too much for the server to handle. In that case, even the server might be shut down. To handle this type of problem, we would need some kind of limiting mechanism that would allow only a certain number of requests to the service.

If we want to put it simply, rate limiting is limiting some operations with some threshold. If those threshold values are crossed, the system will return errors. You may say rate limiting limits the number of operations performed within a given amount of time.

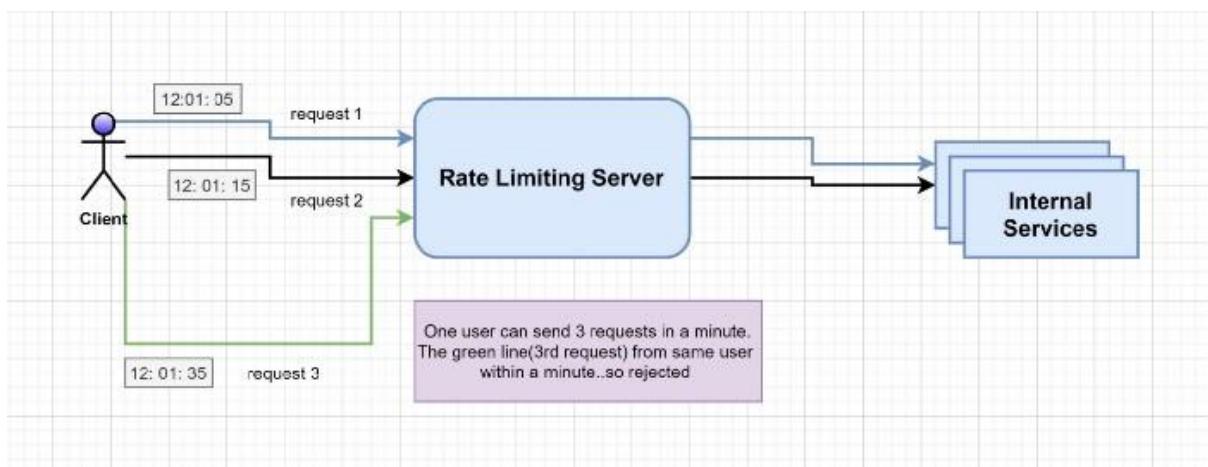


Figure 1: Rate limiter rejects 3rd request (Image by [Author](#))

Rate limiting can be implemented in tiers. E.g., a network request can be limited to 1 request per second, 4 requests per minute, 10 requests per 5 minutes.

Why do Need Rate Limiting?

Rate limiting is needed to protect a system from being brought down by hackers. To know the importance of rate-limiting, we have to know about the DOS attack.

The denial of service attack, which is mostly known as a DOS attack, is when hackers try to flood a system with many requests within a short period of time to shut the system down. Because a server has a limit of how many requests it can serve within a certain period of time. So, the system can't handle the floods of requests properly; it can not handle them properly.

Rate limiting can prevent that from happening as it protects the system from being flooded with intentional unnecessary requests. After a threshold value, the system returns an error.

Say, for example, in Leetcode or some website where we can execute our code, rate limiting may be needed so that users don't spam the code execution service needlessly.

We can rate-limit the system in various ways, like based on user id. Say, we can limit a user's request operation to a server for 5 operations/minute. Other ways to rate-limit can be based on IP address, region, etc.

Even we can use rate limit on the system as a whole. Like, the system may not handle more than 20K requests/minute. If the total user request exceeds that number in a minute, the system is gonna return errors.

Requirements and Goals of the System

In this part, we need to specify the features of the system. The requirements are divided into two parts:

Functional Requirements:

1. A client can send a limited number of requests to a server within a time window, e.g., 10 requests per second.
2. The client should get an error message if the defined threshold limit of request is crossed for a single server or across different combinations of servers.

Non-Functional Requirements:

1. The system should be highly available since it protects our service from external attacks.
2. Performance is an important factor for any system. So, we need to be careful that rate limiter service should not add substantial latencies to the system.

Memory Estimation:

Let's assume we are keeping all of the data in a hash-table. Let's say we have a structure of key-value pair. The key would be the hash value of user ID, and the value would be a structure of count and startTime, e.g., *UserId {Count, StartTime}* *UserId {Count, StartTime}*

Now, let's assume 'UserID' takes 8 bytes and 2 bytes for 'Count,' which can count up to 65k, which is sufficient for our use case. Although the end time will need 4 bytes, we can store only the minute and second parts. The hour part is not necessary if we are not considering a window of an hour. Now, it will take 2 bytes. So, we need a total of 12 bytes to store a user's data.

Now, if our hash-table has an overhead of 20 bytes for each record and we need to track 1 million users at any time, the total memory we need would be :

$$(12 + 20) \text{ bytes} * 1 \text{ million} \Rightarrow 32\text{MB}$$

If we need a 4-byte number to lock each user's record to resolve our atomicity problems, we would require a total of 36MB memory.

Domain Model/High-level Design:

When a new request arrives from the client, the Server first asks the Rate Limiter to decide if it will be served or rejected. If the request is not rejected, then it'll be passed to the internal API servers.

Rate Limiter's responsibility is to decide which client request will be served and which request will be declined.

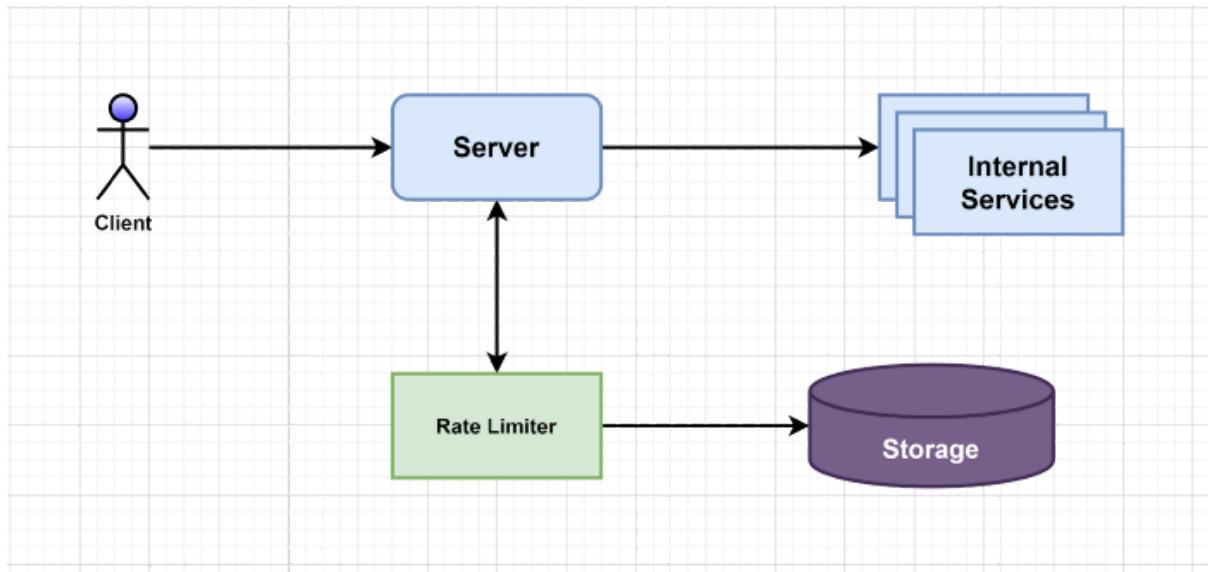


Figure: Domain Model/High-Level Design (Image by [Author](#))

Fixed Window algorithm:

In this algorithm, we may consider a time window from the start to end time unit. For example, our window period can be considered 0–60 seconds, a minute, irrespective of the time frame at which the client requests the server. We already have thought of a structure of count and startTime, e.g., *UserId {Count, StartTime}*

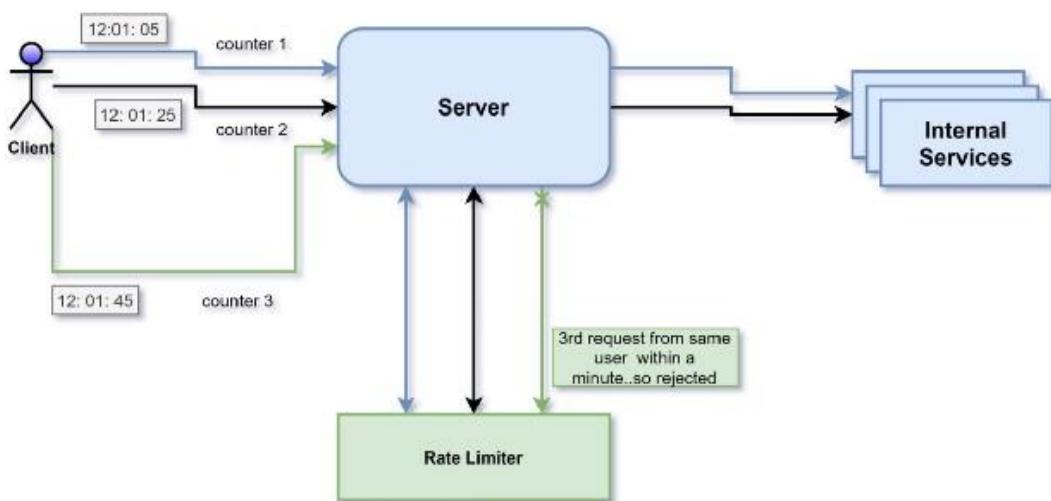


Figure I: Rate limiter allows two requests per minute from the client (Image by [Author](#))

Figure I displays the scenario where a client can make two requests per minute. So, request 1 and request 2 are allowed by the rate limiter. But the third request is blocked as it can within 1 minute from the same user. This type of algorithm is named as **Fixed Window** algorithm. In this algorithm, a period is considered 1 minute(for this example) irrespective of the time frame at which the two API requests have been allowed.

After 1 minute of the first request from a specific client, the start time is reset for that user. This is a simple approach for rate limiters. But there are some drawbacks to this approach.

Drawback:

It can allow twice the number of allowed requests per minute. Imagine the user sends two requests at the last second of a minute(12:01:59) and immediately makes two more requests at the first second(12:02:00) of the next minute. It may seem a minor problem for only two requests. But if the number of requests is much higher(for example, 50 requests) or the time frame is much lower, e.g., 15 seconds, it can cause a huge number of requests for a server. Especially we are considering here only one user. Imagine what will happen to a service that serves millions of users.

Besides, in a distributed system, the “read-and-then-write” behavior can create a race condition. For example, imagine if the client’s current request count is 2 and makes two more requests. If two separate processes serve each of these requests and concurrently read the Count value before either of them updated it, the process would result in the client not hitting the rate limit. Thus, we may need to implement locking each record.

Sliding Window Algorithm:

If we keep track of each request per user in a time frame, we may store the timestamp of each request in a Sorted Set in our ‘value’ field of hash-table. But, it would take a lot of memory. So, we can use a sliding window with the counters. Now, consider this if we keep track of request counts for each user using multiple fixed time windows.

For example, if we have an hourly window, when we receive a new request to calculate the limit, we can count requests each minute and calculate the sum of all counters in the past hour. This would reduce the memory we need for the set.

Suppose we rate-limit at 500 requests per hour with an additional limit of 10 requests per minute. This means that the client has exceeded the rate limit when the sum of the counters in the past hour exceeds the limit request(500).

For a smaller time frame, the client can’t send more than ten requests per minute. This would be a hybrid and reasonable consideration as we may think that none of the real users would send such frequent requests. Even if they do, they will get success with retries since their limits get reset every minute.

Problems in Distributed Environment:

In the case of distributed systems, the algorithms we discussed will have some problems. You may check the figure below to get the problem. A user is allowed to request 3 times

within a minute. The scenario is that the user already made 2 requests and made two new requests within 2 seconds. And the requests from the user went to two different load balancers, then to two different rate limiter services. As the DB already contains the count value 2, both the Rate Limiter service gets the value below the threshold and permits the requests. So, we are now allowing 4 requests from the same user within a minute, which breaks the limit of the rate limiter. This is the inconsistency problem.

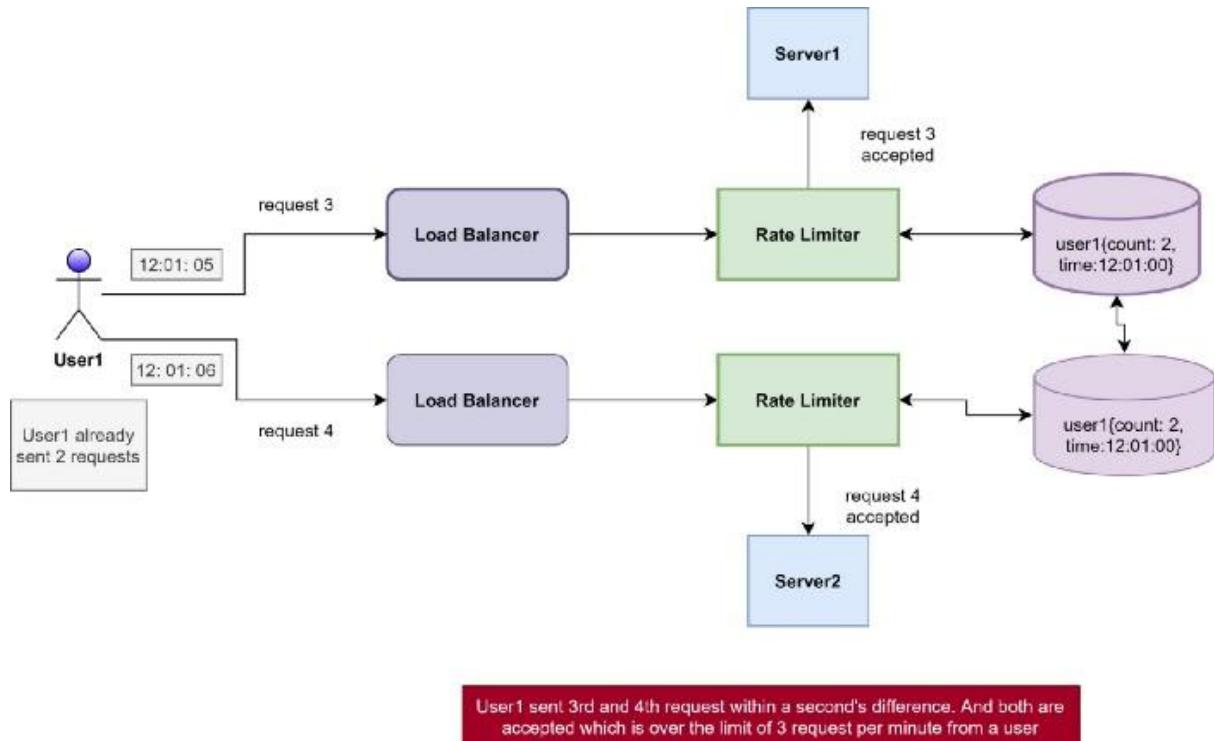


Figure1: Problems in a distributed environment (Image by [Author](#))

As a solution, we may use sticky session load balancing to ensure that one user's request will always go to the same rate limiter service. But the problem here is that it's not a fault-tolerant design. Because if the Rate limiter service is down, user requests can not be served by the system.

We can solve this problem using locks. Once a service uses counter data from the database, it will lock it. So, other services can not use it before the counter data is updated. But this procedure also has its own share of problems. It will add an extra latency for the locking. So, we need to decide between availability and performance trade-offs.

Caching:

This system may get huge benefits from caching recent active users. Servers can quickly check if the cache contains the counter value for the user before hitting backend servers. This should be faster than going to the DB each time.

We may use the **Write-back cache** strategy by updating all counters and timestamps in cache only. Then, we can write to the database done at fixed intervals, e.g., 1 hour. This can ensure minimum latency added to the user's requests, which should improve the Lock latency problem.

When the server reads data, it can always hit the cache first. This will be useful when the user has hit their maximum limit. The rate limiter reads data without any updates.

We may use the Least Recently Used (LRU) as a cache eviction policy for our system.

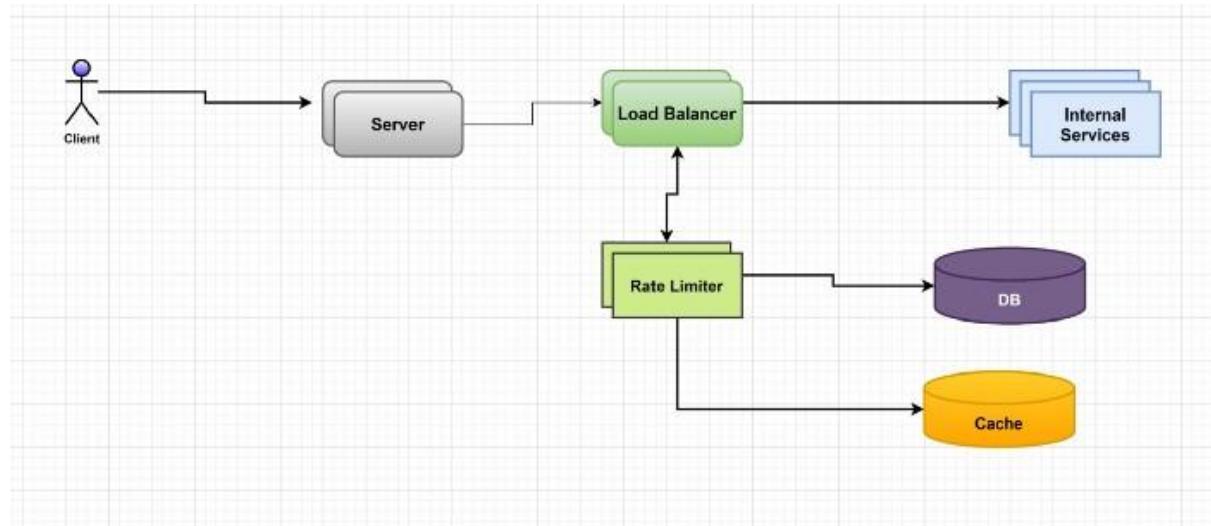


Figure 1: Final Design of Rate Limiter (Image by [Author](#))

DoS attack:

To prevent the system from a DoS attack, we can take other strategies to rate limit by IP address or users.

IP: In this strategy, we limit requests per IP; It might not be a good way to differentiate between normal users and attackers. But, it's still better to have some protection rather than not have anything at all.

The biggest problem with IP-based rate limiting is when multiple users share a single public IP. For example, in an internet cafe or smartphone, users are using the same gateway. Another problem could be, there are lots of IPv6 addresses available to a hacker from even one computer.

User: After user authentication, the system will provide the user a token which the user will pass with each request. This will ensure that we can rate limit an API that has a valid authentication token. But the problem is that we have to rate-limit the login API itself.

The weakness of this strategy would be that a hacker can perform a denial of service attack against a user by providing wrong credentials up to the limit; after that, the actual user may not be able to log in.

So, we can use both the approach together. However, this may result in more cache entries with more details per entry. So, we would need more memory and storage.

Conclusion:

A rate limiter may limit the number of events an entity (user, IP, etc.) can perform in a time window. For example, a user can send only five requests per minute. We can shard based

the data on the ‘UserID’ to distribute the user’s data. We should use Consistent Hashing for fault tolerance and replication. We can have different rate limiters for different APIs for each user or IP.

The task of a rate limiter is to limit the number of requests to or from a system. Rate limiting is used most often to limit the number of incoming requests from the user in order to prevent DoS attacks. The system can enforce it based on IP address, user account, etc.

We should be careful about the fact that normal DoS attacks may be prevented by rate-limiting. But in the case of distributed DoS attack, it may not be enough.

11. Designing Twitter

Solution 1:

Don’t jump into the technical details immediately when you are asked this question in your interviews. Do not run in one direction, it will just create confusion between you and the interviewer. Most of the candidates make mistakes here and immediately they start listing out some bunch of tools or frameworks like MongoDB, Bootstrap, MapReduce, etc. Remember that your interviewer wants **high-level ideas** about how you will solve the problem. It doesn’t matter what tools you will use, but how you define the problem, how you design the solution, and how you analyze the issue step by step.

You can put yourself in a situation where you’re working on real-life projects. Firstly, define the problem and clarify the problem statement. In this question, we will compress Twitter to its MVP (minimum viable product). Nobody expects you to design the whole service. So, we will only design the **core features** of Twitter instead of everything.

1. Discuss The Core Features

So firstly divide the whole system into several core components and talk about some core features. If some other features your interviewer wants to include he/she will mention there. For now, we are going to consider the following features on Twitter...

- The user should be able to tweet in just a few seconds.
- The user should be able to see Tweet Timeline(s)
- **Timeline:** This can be divided into three parts...
 1. *User timeline:* User sees his/her own tweets and tweets user retweet. Tweets that users see when they visit their profile.
 2. *Home Timeline:* This will display the tweets from people users follow. (Tweets when you land on twitter.com)
 3. *Search timeline:* When users search some keywords or #tags and they see the tweets related to that particular keywords.
- The user should be able to follow another user.
- Users should be able to tweet millions of followers within a few seconds (5 seconds)

2. Naive Solution (Synchronous DB queries)

To design a big system like Twitter we will firstly talk about the Naive solution. That will help us in moving towards high-level architecture. You can design a solution for the two things:

1. **Data modeling:** You can use a relational database like MySQL and you can consider two tables **user table** (**id, username**) and a **tweet table** [**id, content, user(primary key of user table)**]. User information will be stored in the user table and whenever a user will tweet a message it will be stored in the tweet table. Two relations are also necessary here. One is the user can follow each other, the other is each feed has a user owner. So there will be a one-to-many relationship between the user and the tweet table.
2. **Serve feeds:** You need to fetch all the feeds from all the people a user follows and render them in chronological order.

3. Limitation of Architecture (Point Out Bottleneck)

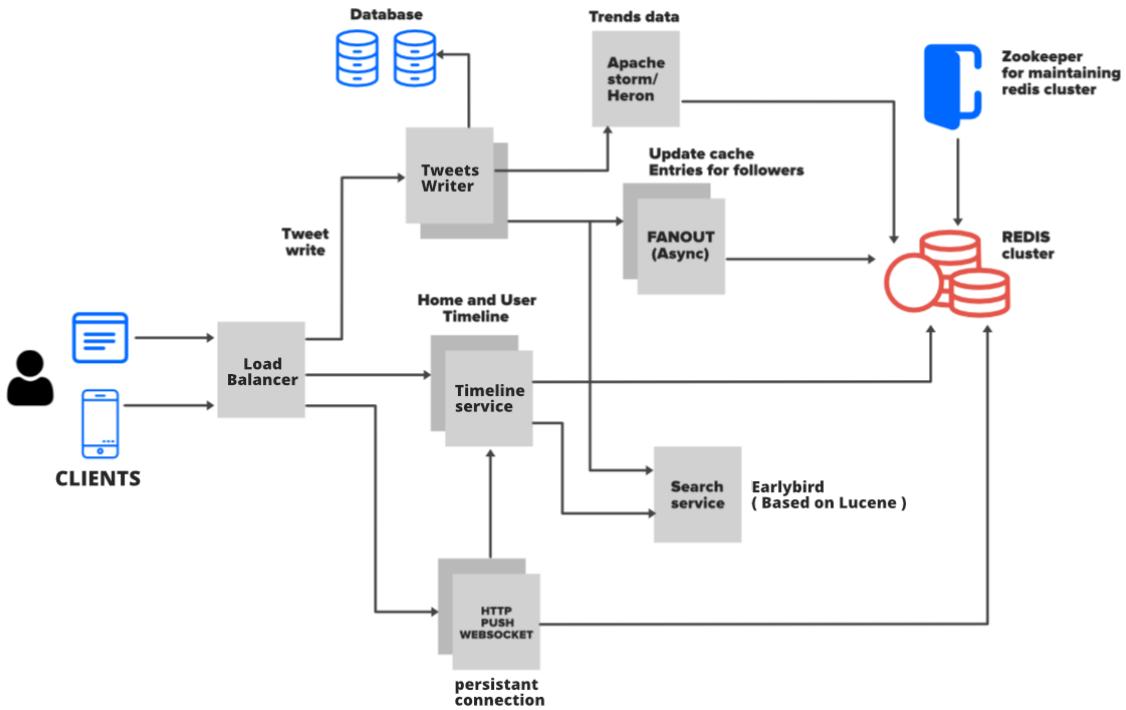
You will have to do a big **select** statement in the tweet table to get all the tweets for a specific user whomsoever he/she is following, and that's also in chronological order. Doing this every time will create a problem because the tweet table will have huge content with lots of tweets. We need to optimize this solution to solve this problem and for that, we will move to the high-level solution for this problem. Before that let's understand the characteristics of Twitter first.

4. Characteristics of Twitter (Traffic)

Twitter has **300M** daily active users. On average, every second **6, 000 tweets** are tweeted on Twitter. Every second **6, 00, 000 Queries** are made to get the timelines. Each user has on average 200 followers and some users like some celebrities have millions of followers. This characteristic of Twitter clears the following points...

1. Twitter has a **heavy read** in comparison to writing so we should care much more about the **availability** and scale of the application for the heavy read on Twitter.
2. We can consider **eventual consistency** for this kind of system. It's completely ok if a user sees the tweet of his follower a bit delayed
3. Space is not a problem as tweets are limited to 140 characters.

High-Level Solution



As we have discussed that Twitter is read-heavy so we need a system that allows us to read the information faster and also it can scale horizontally. **Redis** is perfectly suitable for this requirement but we can not solely dependent on Redis because we also need to store a copy of tweets and other users' related info in the Database. So here we will have the basic architecture of Twitter which consists of three tables...**User Table**, **Tweet Table**, and **Followers Table**.

- Whenever a user will create a profile on Twitter the entry will be stored in the User table.
- Tweets tweeted by a user will be stored in the Tweet table along with the User_id. Also, the User table will have **1 too many relationships** with the Tweet table.
- When a user follows another user, it gets stored in Followers Table, and also caches it Redis. The User table will have **1 too many relationships** with the Follower table.

1. User Timeline Architecture

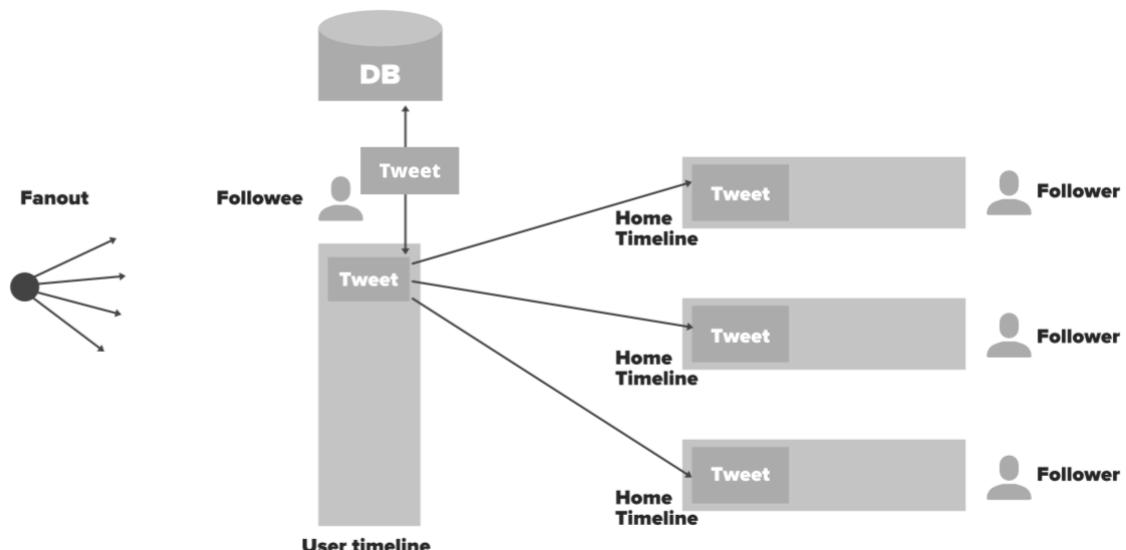
- To get the User Timeline simply go to the user table get the user_id, match this user_id in the tweet table and then get all the tweets. This will also include retweets, save retweets as tweets with original tweet references. Once this is done sort the tweet by date and time and then display the information on the user timeline.
- As we have discussed that Twitter is read-heavy so the above approach won't work always. Here we need to use another layer i.e **caching layer** and we will save the data for user timeline queries in Redis. Also, keep saving the tweets in Redis, so when someone visits a user timeline he/she can get all the tweets made by that user. Getting the data from Redis is much faster so it's not much use to get it from DB always.

2. Home Timeline Architecture

- A user's Home Timeline contains all the latest tweets of the person and the pages that the user follows. Well, here you can simply fetch the users whom a user is following, for each follower fetch all the latest tweets, then merge all the tweets, sort all these tweets by date and time and display them on the home timeline. This solution has some **drawbacks**. The Twitter home page loads much faster and these queries are heavier on the database so this huge search operation will take much more time once the tweet table grows to millions. Let's talk about the solution now for this drawback...

Fanout Approach: Fanout simply means spreading the data from a single point. Let's see how to use it. Whenever a tweet is made by a user (Followee) do a lot of **preprocessing** and distribute the data into different users (followers) home timelines. In this process, you won't have to make any database queries. You just need to go to the cache by user_id and access the home timeline data in Redis. So this process will be much faster and easier because it is an in-memory we get the list of tweets. Here is the complete flow of this approach...

1. User X is followed by three people and this user has a cache called user timeline. X Tweeted something.
2. Through Load Balancer tweets will flow into back-end servers.
3. Server node will save tweet in DB/cache
4. The server node will fetch all the users that follow User X from the cache.
5. The server node will inject this tweet into the in-memory timelines of his followers (fanout)
6. All followers of User X will see the tweet of User X in their timeline. It will be refreshed and updated every time a user will visit his/her timeline.



What will happen if a celebrity will have millions of followers? Is the above method efficient in this scenario?

Weakness (Edge Case): The interviewer may ask the above question. If there is a celebrity who has millions of followers then Twitter can take up to 3-44 minutes for a tweet to flow from Eminem(a celebrity) to his million followers. You will have to update the millions of home timelines of followers which is not scalable. Here is the solution...

Solution [Mixed Approach (In-memory+Synchronous calls)]:

1. Precompute the home timeline of User A (follower of Eminem) with everyone except Eminem's tweet(s)
2. Every user maintains the list of celebrities in the cache as well as whom that user is following. When the request will arrive (tweet from the celebrity) you can get the celebrity from the list, fetch the tweet from the user timeline of the celebrity and then mix the celebrity's tweet at runtime with other tweets of User A.
3. So when User A access his home timeline his tweet feed is merged with Eminem's tweet at load time. So the celebrity tweet will be inserted at runtime.

Other Optimization: For inactive users don't compute the timeline. People who don't log in to the system for a quite long time (say more than 20 days.).

3. Searching

Twitter handles searching for its tweets and #tags using **Earlybird** which is a real-time, reverse index based on Lucene. Early Bird does an **inverted full-text indexing** operation. It means whenever a tweet is posted it is treated as a document. The tweet will be split into tags, words, and #tags, and then these words are indexed. This indexing is done at a big table or distributed table. In this table, each word has a reference to all the tweets which contain that particular word. Since the index is an exact string-match, unordered, it can be extremely fast. Suppose if a user searches for '*election*' then you will go through the table, you will find the word '*election*', then you'll figure out all the references to all the tweets in the system, then it gives all the results that contain the word '*election*'.

Twitter handles thousands of tweets per second so you can't have just one big system or table to handle all the data so it should be handled through a distributed approach. Twitter uses the strategy **scatter and gather** where it set up the **multiple servers or data center** which allow indexing. When Twitter gets a query (let's say `#geeksforgeeks`) it sends the query to all the servers or data centers and it queries every **Early Bird shard**. All the early bird that matches with the query return the result. The results are returned, sorted, merged, and reranked. The ranking is done based on the number of retweets, replies, and the popularity of the tweets.

So far we have talked about all the core features and components of Twitter. There are some other in-depth components you can talk about. For example, you can talk about **Trends / Trending topics** (using Apache Storm and Heron framework), you can talk about **notifications** and how to incorporate **advertisement**.

Solution 2:

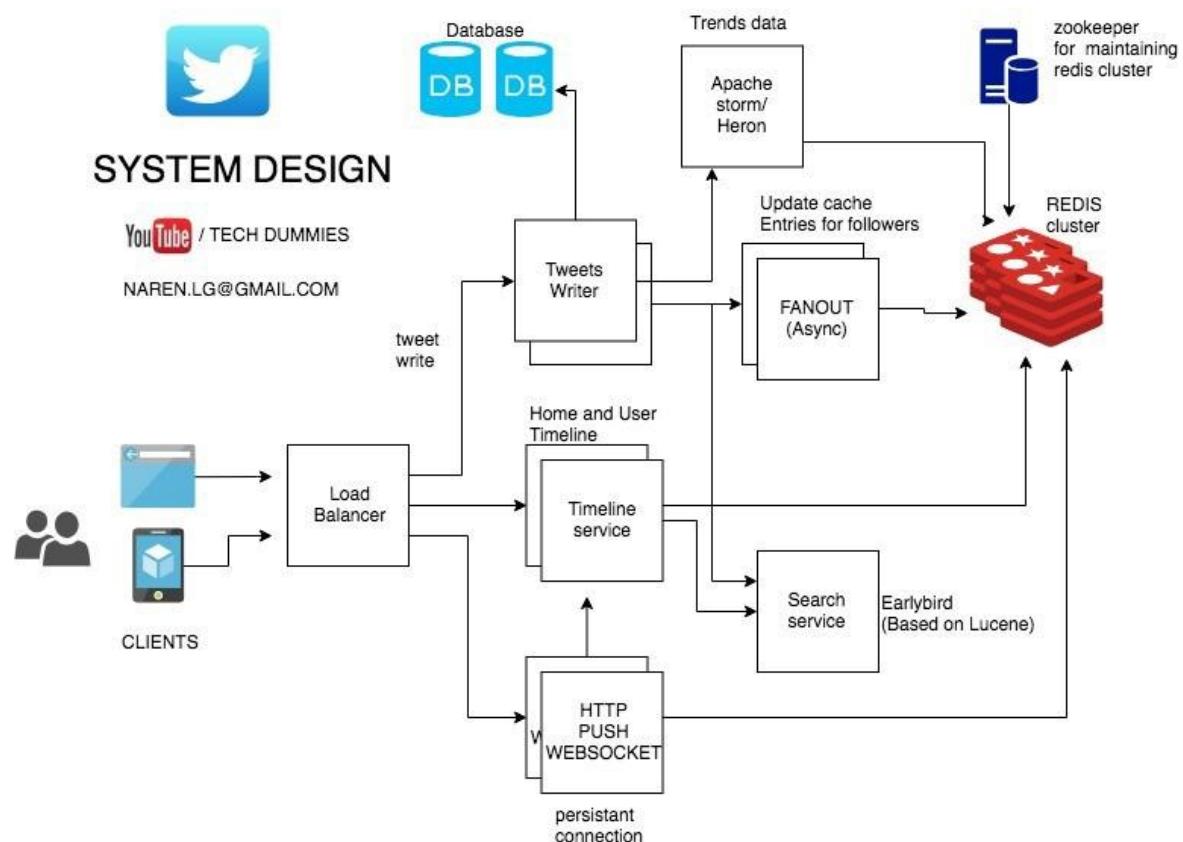
Traffic: Twitter now has 300M worldwide active users. Every second on an average, around 6,000 tweets are tweeted on Twitter. Also, every second 6,00,000 Queries made to get the timelines!!

User Features?

- The user should be able to tweet as fast as possible
- The user should be able to see Tweet Timeline(s)
- User timeline: Displaying user's tweets and tweets user retweet
- Home timeline: Displaying Tweets from people user follow
- Search timeline: Display search results based on #tags or search keyword
- The user should be able to follow another user
- Users should be able to tweet millions of followers within a few seconds (5 seconds)
- The user should see trends

Considerations before designing twitter

- If you see all of the features, it looks like read heavy, compared to write
- Its ok to have Eventual consistency, It's not much of a pain if the user sees the tweet of his follower a bit delayed
- Space is not a problem as tweets are limited to 140 characters



So lets design how to provide different timelines

As we know the system is read heavy let's use REDIS to access most of the information faster and also to store data but don't forget to store a copy of tweet and other users related info in Database.

Basic Architecture of Twitter service consists of a User Table, Tweet Table, and Followers Table

- User Information is stored in User Table
- When a user tweets it gets stored in the Tweet Table along with User ID.
- User Table will have 1 to many relationships with Tweet Table
- When a user follows another user, it gets stored in Followers Table, and also cache it Redis

So now how to build USER TIMELINE? which is shown below

The screenshot shows a user profile for 'Liam Mac Attack' on the left, featuring a profile picture, the name 'Liam Mac Attack', a link to 'View my profile page', and statistics: 159 TWEETS, 66 FOLLOWING, and 22 FOLLOWERS. Below this is a 'Compose new Tweet...' input field. To the right is a 'Who to follow' section with links to profiles for 'Intel', 'Dave Shea', and 'Matthew Bilotti'. Further down is a 'San Francisco trends' section with hashtags like '#NameTuesday', '#MomentsICanNeverForget', '#YouWasSexyUntil', '#FirstQuestionsAsked', and 'Drew Brees'. On the right side, under the heading 'Tweets', is a list of recent tweets from various users:

- crystal @crystal
@keerthi @lukester @mcgee @magnuson I would love that :)
In reply to Keerthi Prakash
- Lane Collins @lane
Taking a walk on foggy NW 23rd. 17 mins
- Timmy Mezzy @timmymezzy
Awesome new music video by our own @scottspedel. Check it out: youtube.com/watch?v=z2z8jO...
View video 28 mins
- Barack Obama @BarackObama
A day in the life of a Philadelphia music teacher who wants Congress to pass the American Jobs Act: OFA.BO/PeNsU6 28 mins
- Arrested Development @bluthquotes
RIP Patrice. tmz.com/2011/11/29/pat... twitpic.com/7lq468
Retweeted by Danny Hertz
View photo 34 mins
- Keerthi Prakash @keerthi
@lukester @mcgee @crystal @magnuson we should all go out for lunch sometime. I would say to Zuppa bcoz that's where we had our 1st team lunch! 43 mins

- Fetch all the tweets from Global Tweet Table/Redis for a particular user
- Which also includes retweets, save retweets as tweets with original tweet reference
- Display it on user timeline, order by date time

As optimization save user timeline in the cache, eg: celebrities timelines are accessed million times so it's not much use to get it from DB always.

Home timeline

twitter  Search Home Profile Messages What's happening



Jill Dawnsoll

@jilldawnsoll

Ecommerce analyst and consultant.

+ Follow
▼ □

Tweets Favorites Following Followers Lists

jilldawnsoll Jill Dawnsoll
White Paper: Brand Value Creation, Communications and Equity <http://bit.ly/kSrrLy>
18 minutes ago

jilldawnsoll Jill Dawnsoll
White Paper: The Evolving Opportunities of Social Media <http://bit.ly/kwn0u5>
38 minutes ago ☆ Favorite RT Retweet Reply

jilldawnsoll Jill Dawnsoll
White Paper - Maximizing Social Media Influence for Online Retailing <http://bit.ly/eWg5EB>
58 minutes ago

jilldawnsoll Jill Dawnsoll
White Paper - Word of Mouth Marketing Identification, Engagement and Management <http://bit.ly/h2zHGv>
1 hour ago

About

3,132 tweets 1 following

Following

Similar to

About Help Businesses

Strategy 1:

- Fetch all the users whom this user is following (from followers table)
- Fetch tweets from global tweet table for all
- Display it on home timeline
- Drawback: This huge search operation on relational DB is NOT Scalable. Though we can use sharding etc, this huge search operation will take time once the tweet table grows to millions.

Strategy 2: Solution is a write based fanout approach. Do a lot of processing when tweets arrive to figure out where tweets should go. This makes read time access fast and easy. Don't do any computation on reads. With all the work being performed on the write path ingest rates are slower than the read path. So precompute timelines for every active user

and store it in the cache, so when the user accesses the home timeline, all you need to do is just get the timeline and show it

You can store this data in DB but what's the point?

A simple flow of tweet looks like,

- User A Tweeted
- Through Load Balancer tweet will flow into back-end servers
- Server node will save tweet in DB/cache
- Server node will fetch all the users that follow User A from the cache
- Server node will inject this tweet into in-memory timelines of his followers
- Eventually, all followers of User A will see the tweet of User A in their timeline

BUT does it work always? is it efficient?

If you see Celebrities they have Millions of Followers

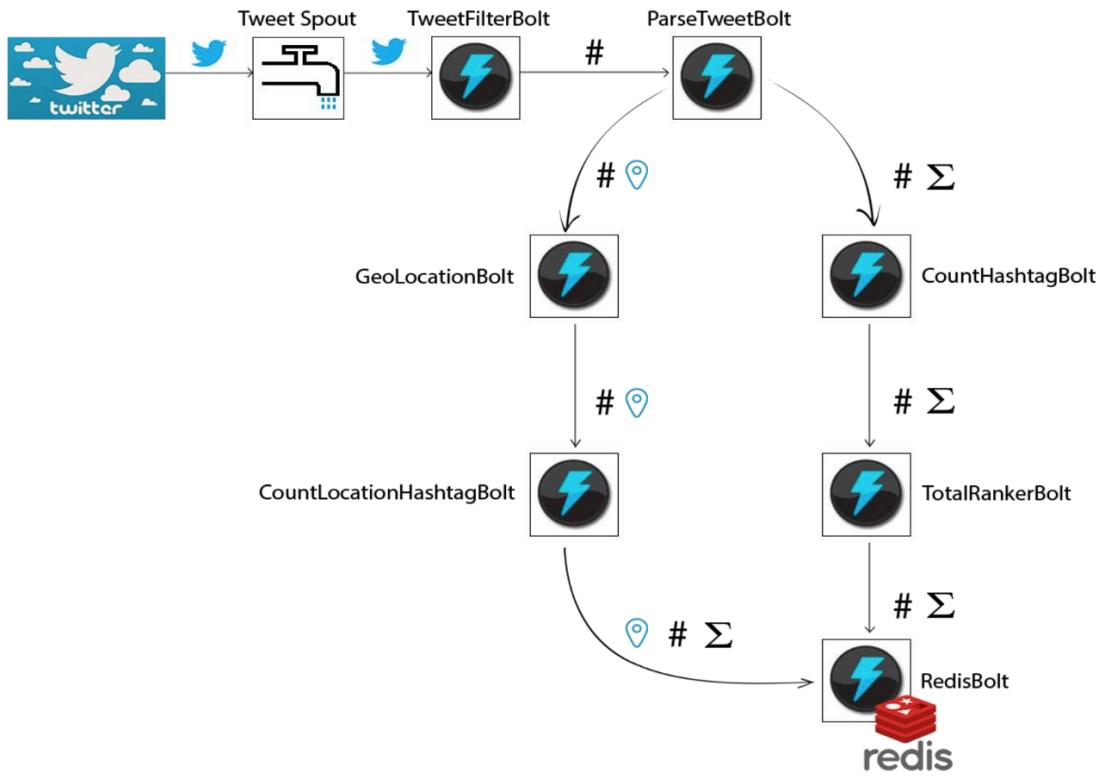
Twitter system and it can take up to 4 minutes for a tweet to flow from TaylorSwift to her 83 million followers. So If a person with millions of the followers on Twitter tweets, We need to update millions of lists which is not very scalable. So what we can do is,

- Precomputed home timeline of User A with everyone except celebrity tweet(s)
- When User A access his timeline his tweet feed is merged with celebrity tweet at load time.
- For every user have the mapping of celebrities list and mix their tweets at runtime when the request arrives and cache it.

What are the other optimizations we can make?

- Don't compute timeline for inactive users who don't log in to the system for more than 15 days!!!

So how Trends / Trending topics are calculated?



Twitter uses Apache Storm and Heron framework to compute trending topics

These tasks run on many containers, These applications create a real-time analysis of all tweets send on the Twitter social network which can be used to determine the so-called trending topics.

Basically, method implies the counting of the most mentioned terms in the poster tweets in the Twitter social network.

The method is known in the domain of data analysis for the social network as “Trending Hashtags” method. Suppose two subjects A and B, the fact that A is more popular than B is equivalent to the fact that the number of mentions of the subject A is greater than the number of mentions of the subject B

Information needed

1. The number of mentions of a subject (hashtags)
2. Total time taken to generate that volume of tweets

TweetSpout: Represents a component used for issuing the tweets in the topology

TweetFilterBolt: Reads the tweets issued by the TweetSpout and executes the filtering. Only tweets that contain coded messages using the standard Unicode. also, violation and CC checks are made.

ParseTweetBolt: Processes the filtered tweets issued as tuples by the component TweetFilterBolt. Taking into consideration that the tuple is filtered, at this level we have the guarantee that each tweet contains at least one hashtag

CountHashtagBolt: Takes the tweets that are parsed through the component ParseTweetBolt and counts each hashtag. This is to get the hashtag and number of references to it

TotalRankerBolt: Makes a total ranking of all the counted hashtags. Converts count to ranks in one more pipeline.

GeoLocationBolt: It takes the hashtag issued by the ParseTweetBolt, with the location of the tweet.

CountLocationHashtagBolt: Presents a functionality similar to the component CountHashtagBolt but uses one more dimension i.e. Location

RedisBolt: inserts into redis

Searching:

Early Bird uses inverted full-text index. This means that it takes all the documents, splits them into words, and then builds an index for each word. Since the index is an exact string-match, unordered, it can be extremely fast. Hypothetically, an SQL unordered index on a varchar field could be just as fast, and in fact I think you'll find the big databases can do a simple string-equality query very quickly in that case.

Lucene does not have to optimize for transaction processing. When you add a document, it need not ensure that queries see it instantly. And it need not optimize for updates to existing documents.

However, at the end of the day, if you really want to know, you need to read the source. Both things you reference are open source, after all.

It has to scatter-gather across the datacenter. It queries every Early Bird shard and asks do you have content that matches this query? If you ask for "New York Times" all shards are queried, the results are returned, sorted, merged, and reranked. Rerank is by social proof, which means looking at the number of retweets, favorites, and replies.

Databases:

- Gizzard is Twitter's distributed data storage framework built on top of MySQL (InnoDB). InnoDB was chosen because it doesn't corrupt data. Gizzard is just a datastore. Data is fed in and you get it back out again. To get higher performance on individual nodes a lot of features like binary logs and replication are turned off. Gizzard handles sharding, replicating N copies of the data, and job scheduling.
- Cassandra is used for high velocity writes, and lower velocity reads. The advantage is Cassandra can run on cheaper hardware than MySQL, it can expand easier, and they like schemaless design.
- Hadoop is used to process unstructured and large datasets, hundreds of billions of rows.

- Vertica is being used for analytics and large aggregations and joins so they don't have to write MapReduce jobs.

And I believe this article helps you to learn about how Twitter works, if not completely at least a bit.

Solution 3:

Functional Requirements

- Tweet - should allow you to post text, image, video, links, etc
- Re-tweet - should allow you to share someone's tweets
- Follow - this will be a directed relationship. If we follow Barack Obama, he doesn't have to follow us back
- Search

Non Functional Requirements

- Read heavy - The read to write ratio for twitter is very high, so our system should be able to support that kind of pattern
- Fast rendering
- Fast tweet.
- Lag is acceptable - From the previous two NFRs, we can understand that the system should be highly available and have very low latency. So when we say lag is ok, we mean it is ok to get notification about someone else's tweet a few seconds later, but the rendering of the content should be almost instantaneous.
- **Scalable** - 5k+ tweets come in every second on twitter on an average day. On peak times it can easily double up. These are just tweets, and as we have already discussed read to write ratio of twitter is very high i.e. there will be an even higher number of reads happening against these tweets. That is a huge amount of requests per second.

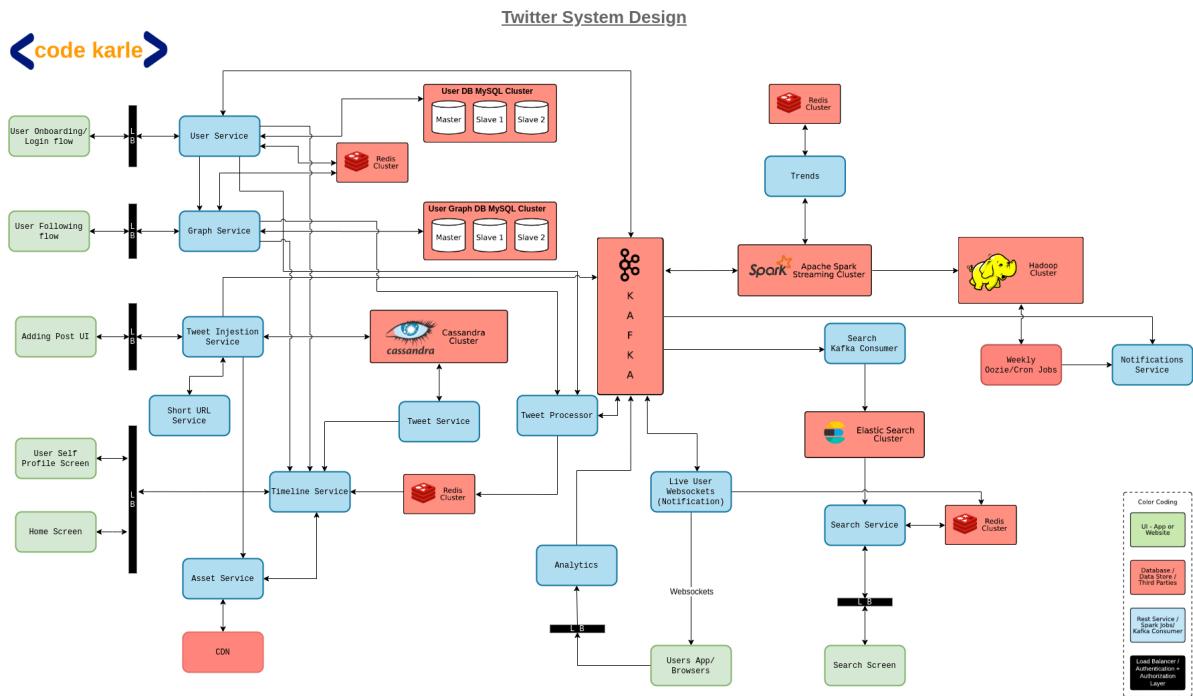
So how do we design a system that delivers all our functional requirements without compromising the performance? Before we discuss the overall architecture, let's split our users into different categories. Each of these categories will be handled in a slightly different manner.

1. **Famous Users:** Famous users are usually celebrities, sportspeople, politicians, or business leaders who have a lot of followers
2. **Active Users:** These are the users who have accessed the system in the last couple of hours or days. For our discussion, we will consider people who have accessed twitter in the last three days as active users.
3. **Live Users:** These are a subset of active users who are using the system right now, similar to online on Facebook or WhatsApp..
4. **Passive Users:** These are the users who have active accounts but haven't accessed the system in the last three days.
5. **Inactive Users:** These are the "deleted" accounts so to speak. We don't really delete any accounts, it is more of a soft delete, but as far as the users are concerned the account doesn't exist anymore.

Now, for simplicity, let us divide the overall architecture into three flows. We will separately look at the onboarding flow, tweet flow, and search and analytics side of the system.

Note: Remember how twitter is a very read-heavy system? Well, while designing a read-heavy system, we need to make sure that we are precomputing and caching as much as we can to keep the latency as low as possible.

System Architecture



Onboarding flow

We have a **User Service** that will store all the user-related information in our system and provide endpoints for login, register, and any other internal services that need user-related information by providing GET APIs to fetch users by id or email, POST APIs to add or edit user information and bulk GET APIs to fetch information about multiple users. This user service sits on top of a **User DB** which is a **MySQL** database. We use MySQL here as we have a finite number of users and the data is very relational. Also, the User DB will mostly power the writes and the reads related to user details will be powered by a **Redis** cache which is an image of User DB. When user service receives a GET request with a user id, it will firstly lookup in the Redis cache, if the user is present in the Redis it will return the response. Otherwise, it will fetch the information from User DB, store it in Redis, and then respond to the client.

User-follow flow

The “follow” related requests will be serviced by a **Graph Service**, which creates a network of how the users are connected within the system. Graph service will expose APIs to add follow links, get users followed by a certain user-id or the users following a certain user-id. This Graph service sits on top of a **User Graph DB** which is again a **MySQL DB**. Again, the

follow-links won't change too frequently, so it makes sense to cache this information in a **Redis**. Now in the follow flow we can cache two pieces of information - who are the **followers** of a particular user, and who is the user **following**. Similar to the user service, when graph service receives a get request it will firstly lookup in the Redis. If Redis has the information it responds to the user. Otherwise, it fetches the information from the graph DB, stores it in Redis, and responds to the user.

Now, there are some things we can conclude based on a user's interaction with Twitter, like their interests, etc. So when such events occur, an **Analytics Service** puts these events in a **Kafka**.

Now, remember our Live users? Say U1 is a live user following U2 and U2 tweets something. Since U1 is live, it makes sense that U1 gets notified immediately. This happens through the **User Live Websocket service**. This service keeps an open connection with all the live users, and whenever an event occurs that a live user needs to be notified of, it happens through this service. Now based on the user's interaction with this service we can also track for how long the users are online, and when the interaction stops we can conclude that the user is not live anymore. When the user goes offline, through the websocket service an event will be fired to Kafka which will further interact with user service and save the last active time of the user in Redis, and other systems can use this information to accordingly modify their behavior.

Tweet flow

Now a tweet could contain text, images, videos, or links. We have something called an **Asset service** which takes care of uploading and displaying all the multimedia content in a Tweet. We have discussed the nitty-gritty of asset service in the [Netflix design article](#), so check that out if you are interested.

Now, we know that tweets have a constraint of 140 characters which can include text and links. Thanks to this limit we cannot post huge URLs in our tweets. This is where a **URL shortener service** comes in. We are not going into the details of how this service works, but we have discussed it in our [Tiny URL article](#), so make sure to check it out. Now that we have handled the links as well, all that is left is to store the text of a tweet and fetch it when required. This is where the **tweet ingestion service** comes in. When a user tries to post a tweet and hits the submit button, it calls the tweet ingestion service which stores the tweet in a permanent data store. We use **Cassandra** here because we will have a huge amount of tweets coming in every day and the query pattern we require here is what Cassandra is best for. To know more about why we use the database solutions we do check out our article about [choosing the best storage solutions](#).

Now, the tweet ingestion service, as the name suggests, is only responsible for posting the tweets and doesn't expose any GET APIs to fetch tweets. As soon as a tweet is posted, the tweet ingestion service will fire an event to **Kafka** saying a tweet id was posted by so and so user id. Now on top of our Cassandra, sits a **Tweet service** that will expose APIs to get tweets by tweet id or user id.

Now, let's have a quick look at the users' side of things. On the read-flow, a user can have a user timeline i.e. the tweets from that user or a home timeline i.e. tweets from the people a

user is following. Now a user could have a huge list of users they are following, and if we make all the queries at runtime before displaying the timeline it will slow down the rendering. So we cache the user's timeline instead. We will precalculate the timeline of active users and cache it in a **Redis**, so an active user can instantaneously see their timeline. This can be achieved with something called a **Tweet processor**.

As mentioned before, when a tweet is posted, an event is fired to Kafka. Kafka communicates the same to the tweet processor and creates the timeline for all the users that need to be notified of this recent tweet and cache it. To find out the followers that need to be notified of this change tweet service interacts with the graph service. Suppose user U1, followed by users U2, U3, and U4 posts a tweet T1, then the tweet processor will update the timelines for U2, U3, and U4 with tweet T1 and update the cache.

Now, we have only cached the timelines for active users. What happens when a passive user, say P1, logs in to the system? This is where the **Timeline Service** comes in. The request will reach the timeline service, timeline service will interact with the user service to identify if P1 is an active user or a passive user. Now since P1 is a passive user, its timeline is not cached in Redis. Now the timeline service will talk to the graph service to find a list of users that P1 follows, then queries the tweet service to fetch tweets of all those users, caches them in the Redis, and responds back to the client.

Now we have seen the behavior for active and passive users. How will we optimize the flow for our live users? As we have previously discussed, when a tweet is successfully posted an event will be sent to Kafka. Kafka will then talk to the tweet processor which creates timelines for active users and saves them in Redis. But here if the tweet processor identifies that one of the users that need to be updated is a live user, then it will fire an event to Kafka which will now interact with the live websocket service we briefly discussed before. This websocket service will now send a notification to the app and update the timeline.

So now our system can successfully post tweets with different types of content and has some optimization built in to handle active, passive, and live users in a somewhat different manner. But it is still a fairly inefficient system. Why? Because we completely forgot about our famous users! If Donald Trump has 75 million followers, then every time Trump tweets about something, our system needs to make 75 million updates. And this is just one tweet from one user. So this flow will not work for our famous users.

Redis cache will only cache the tweets from non-famous users in the precalculated timelines. Timeline service knows that Redis only stores tweets from normal users. It interacts with graph service to get a list of famous users followed by our current user, say U1, and it fetches their tweets from the tweet service. It will then update these tweets in Redis and add a timestamp indicating when the timeline was last updated. When the next request comes from U1, it checks if the timestamp in Redis against U1 is from a few minutes back. If so, it will query the tweet service again. But if the timestamp is fairly recent, Redis will directly respond back to the app.

Now we have handled active, passive, live, and famous users. As for the inactive users, they are already deactivated accounts so we need not worry about them.

Now what happens when a famous user follows another famous user, let's say Donald Trump and Elon Musk? If Donald Trump tweets, Elon musk should be notified immediately even if the other non-famous users are not notified. This is handled by the tweet processor again. Tweet processor, when it receives an event from Kafka about a new tweet from a famous user, let's say, Donald Trump, updates the cache of the famous users that follow Trump.

Now, this looks like a fairly efficient system, but there are some bottlenecks. Like Cassandra - which will be under a huge load, Redis - which needs to scale efficiently as it is stored completely in RAM, and Kafka - which again will receive crazy amounts of events. So we need to make sure that these components are horizontally scalable and in case of Redis, don't store old data that just unnecessarily uses up memory.

Now coming to **search** and **analytics**!

Remember the tweet **ingestion service** we discussed in the previous section? When a tweet is added to the system, it fires an event to Kafka. A search consumer listening to Kafka stores all these incoming tweets into an **Elasticsearch** database. Now when the user searches a string in **Search UI**, which talks to a **Search Service**. Search service will talk to elastic search, fetch the results, and respond back to the user.

Now assuming an event occurred and people are tweeting or searching about it on Twitter, then it is safe to assume that more people will search for it. Now we shouldn't have to query the same thing again and again on elasticsearch. Once the search service gets some results from elasticsearch, it will save them in Redis with a time-to-live of 2-3 minutes. Now when the user searches something, Search service will firstly lookup in Redis. If the data is found in Redis it will be responded back to the user, otherwise, the search service will query elasticsearch, get the data, store it in Redis, and respond back to the user. This considerably reduces the load on elasticsearch.

Let's go back to our Kafka again. There will be a **spark streaming consumer** connected to Kafka which will keep track of trending keywords and communicate them to the **Trends service**. This could further be connected to a **Trend UI** to visualize this data. We don't need a permanent data store for this information as trends will be temporary but we could use a Redis as a cache for short-term storage.

Now you must have noticed we have used Redis very heavily in our design. Now even though Redis is an in-memory solution, there is still an option to save data to disk. So in case of an outage, if some of the machines go down, you still have the data persisted on the disk to make it a little more fault-tolerant.

Now, other than trends there is still some other analytics that can be performed like what are people from India talking about. For this, we will dump all the incoming tweets in a **Hadoop cluster**, which can power queries like the most re-tweeted posts, etc. We could also have a **weekly cron job** running on the Hadoop cluster, which will pull in the information about our passive users and send out a weekly newsletter to them with some of the most recent tweets that they might be interested in. This could be achieved by running some simple ML algorithms that could tell the relevance of tweets based on the previous searches and reads

by the users. The newsletters can be sent via a notification service that can talk to user service to fetch the email ids of the users.

Solution 4:

Before approaching the solution, it is essential for the candidate to deep dive into how Twitter works and know the basic functionalities.

Twitter is a social networking service where a user can post and read posts which are called tweets. Over the years, there has been an exponential increase in the user base of Twitter. The frequent tweets about various political issues, tech news, etc., add to heavy daily traffic. On average, 600 tweets are generated per second, and almost 600,000 tweets are read by the users every second.

An average user has nearly 200 followers, and the celebrities can have followers in millions. Hence, Twitter deals with a massive amount of tweets and has a heavy user base.

The basic functionalities of Twitter are as follows:-

1. Create an account
2. Create tweets
3. Read tweets
4. Users can delete their tweets but not update/edit their posted tweets.
5. Users can comment, retweet, share, and like the tweets.
6. View timelines: Home Timeline, User Timeline, Search Timeline
7. Follow other users
8. Users must be able to check the trending hashtag in their region.

Twitter System Design

Generally, candidates attempt to design a monolithic application, i.e., a single unified unit, in a System Design Interview. However, designing a service like Twitter as a monolithic service shows that the candidate is not experienced in designing distributed systems. Nowadays, companies also have started building applications using microservices architecture, i.e., breaking the service into a collection of smaller independent units.

Hence, the candidates should discuss Twitter system design as a **microservices architecture** instead of a monolithic architecture.

After going through the functionalities, we can notice the following:-

- Operations are read-heavy rather than write.
- Space won't be an issue as there is a character limit of 140.
- The system need not be strongly consistent. Eventual consistency would be fine as it won't be an issue if the user sees the tweet a bit delayed.

Requirements

Before jumping into the solution, let's discuss the requirements for the application.

Functional requirements:

The functional requirements are as follows:

- Post Tweets
- Read Tweets
- Generate timeline
- React to tweets
- Messaging system

Non-Functional requirements:

The non-functional requirements are as follows:

- High Availability
- Low latency
- Eventual consistency.

Low-Level Design

Before going to the optimized approach, one can always start by giving the naive approach to design the platform by communicating the initial thoughts on the system. We can start with a system supporting a few hundred users and scale it afterward.

The basic requirement for the platform would be a database. One can go for relational databases like MySQL, MS SQL Server, or cloud relational databases like Google Cloud SQL, etc. This database will contain the user table and tweet table.

User table (userID, userName): It will store the information about the user like the userID and the userName. Here the [primary key](#) can be the userId attribute.

Tweet table(tweetID, content, userID, date): It will store the tweet's id, the tweet's content, user id, and date tweeted. Here the primary key can be the tweetId attribute. When a user tweets, the tweet gets stored in the tweet Table along with the userID.

Relationships established between tables:

1. Self-referential relationship: Self-referential relationship is the relationship of one record with other records in the same table. Here the self-referential relationship is established in the user table.

2. User table and tweet table: It will be a one-to-many relationship between the user and the tweet table. The foreign key will be userID in this case.

Limitation Of Naive Approach

In the above approach, every time a user wants to get the feed, the select query must be applied to the tweet table. This is not practically possible due to the large user base. In this approach, the loading and displaying results will not be very optimal, leading to a bad experience for the user.

High-Level Design

As we had mentioned above, the operations in Twitter are **read-heavy**. So, the system we design should allow the user to go through the tweets efficiently. To make the system scalable, we would be using **horizontal scaling** as they are more resilient and scale well as the users increase. We can create three tables in the database: user table, tweet table, and followers table.

User table: The user's information is stored when a user creates a profile on Twitter.

Tweet table: The tweets will be stored here, and they will have a one-to-many relationship with the user table, as discussed above.

Followers table: When a user follows another user on Twitter, it gets stored in the followers table. Caching is done here to avoid redoing the same operations and save time. It also has a one-to-many relationship with the user table.

Architecture and Components

Now let's discuss the platform's architecture, components used for the client app, caching, etc.

Client App

The Twitter app is built as a progressive web app (PWA) to work on desktop as well as mobile devices. Features like push notification, immediate loading, offline browsing, etc., are achieved using the service workers. [React](#), along with [Redux](#), is used to handle Twitter's data on the frontend. A proxy server called a [reverse proxy](#), used to retrieve resources on behalf of a client from one or more servers, is used to create a tweet on the client-side.

Backend

Backend is designed using [node.js](#) and Express on the server-side. The server interacts with Redis Clusters and the database. For the APIs (Application Programming Interface), [Rest architecture](#) is used.

- Login API- Post API to authenticate the user.
- Signup API- Post API to create a user.
- Get all followers- Get API /api/users/\${userName}/followers?page=1&size=30
- Get all following- Get API api/users/\${userName}/followees?page=1&size=30
- Get all Tweets- Get API /api/users/\${userName}/tweets?page=1&size=30
- Create a tweet- Post API /api/users/\${userName}/tweets

To know more about APIs and request methods, refer to the blog [Introduction to API and its usage](#).

Databases

Twitter uses various databases depending on the requirements. The databases used are:-

- **Hadoop** – Used to store massive data like trends Analytics, storing user logs, recommendations, etc.
- **MySQL and Manhattan as master DB** – Used to store data of end-users. Manhattan is a NoSQL database used to store tweets, messages, etc. It can provide real-time multi-tenant scalable distributed DB and serve millions of queries fast.
- **MemCache and Redis** – Used to store cache data which is used to generate a timeline.
- **FlockDB** – Used to store social graphs, i.e., how users are connected.
- **Metrics DB** – Used to store data metrics of Twitter.
- **BlobStore** – Used to store binary large objects and images, videos, and other files.

Content Delivery Network (CDN)

CDN is used to serve the tweet media content over a local network in particular regions.

Now let's see how Twitter performs functions like searching, creating user timelines, etc.

Generate user timeline

How does the user timeline work on Twitter?

To generate the user timeline, we have to get the UserId from the user table and map it to the userId in the tweet table. This returns a list of the tweets, including the retweets made by the user, etc. The tweet list is then sorted by date and time and then displayed on the user timeline.

This approach won't be efficient for a large user base. So some of the user timeline queries are stored in the **Redis server for caching**. This avoids the need for direct access to the database and increases the application performance. This caching layer is useful when some other user wants to access our tweets. The caching layer directly gets the data from the Redis server and makes it accessible to the other user.

Generate home timeline

How does twitter display all the tweets on the user's home page?

The home timeline would contain the tweets of the user and the pages or the users that they follow. To generate the home timeline, first, collect the userIds of the account the user is following. After that, fetch the tweets made by those userIds and merge the tweets in the order of the time generated.

As you might have guessed, the above approach is going to serve a considerable time overhead. This is because the Twitter home page loads fast, and the queries will become heavy on the database when the tweet table grows to millions. To solve this issue, the **fanout approach** is used.

Fanout Approach: When a user makes a tweet, it is injected into their followers' timeline in the fanout approach. In this process, queries are not required, and the time overhead issue is resolved here. This injecting of tweets is made possible by a Redis server that helps at the

caching layer, saves some of the user's tweets queries as cache, and uses them as and when required.

Fanout Approach

Let's understand fanout with the help of the figure. Suppose three people follow a user, and this user has data stored in the cache. When the user tweets through the [load balancer](#), the tweet will flow into backend servers. The server node will save the tweet in the database and fetch all the user's followers from the cache.

The server node will then inject this tweet into the in-memory timelines of the followers through the fanout process. And finally, the home timeline is updated.

Is the above method efficient for a celebrity with millions of followers?

It is not possible to update millions of home timelines efficiently, so the fanout approach fails here. In this scenario, follow the following steps:-

1. Generate the home timeline of a follower of the celebrity with everyone except the celebrity tweets
2. In the cache, maintain the list of celebrities the user follows and the common people the user follows.
3. When a celebrity makes a tweet, get the celebrity from the celebrity list and fetch the tweet from the user timeline of the celebrity.
4. Merge the celebrity's tweet at runtime with the home timeline of the follower.

To optimize this further, timeline generation can be avoided for users who haven't logged in for a long time.

Generate search timeline

How does searching work on Twitter?

Earlybird is the system used to manage real-time tweet-searching or hashtags by Twitter. It does an **inverted full-text indexing operation** when a tweet is posted. To do this, the tweet is treated as a document, splits them into words, and then builds an index for each word. This indexing is done at a vast and distributed table where each word references all the tweets that contain that particular word. As the index is an exact string-match and unordered, it is speedy.

Twitter tweets are handled through a distributed approach rather than a single big system as it has to take a vast amount of tweets per second. So Twitter uses the **scatter and gather** strategy where multiple data centers that allow indexing is set up.

When Twitter receives a query, it sends the query to all the data centers. Then it queries every Early Bird [shard](#) and checks if the content matches with the query. All the matches are then returned as results. This result is displayed in the search timeline after sorting, merging, and ranking. The ranking is done based on the number of retweets, replies, and the popularity of the tweets.

Trending Hashtags

How does Twitter decide trending hashtags in real-time?

Twitter uses the frameworks: Apache Storm and Heron to decide the trending hashtags. The applications generate a real-time analysis of the tweets sent on Twitter to determine the trending hashtags.

This process used for the data analysis of Twitter is called the “**Trending Hashtags**” method. In this process, the number of mentions of a particular hashtag and the total time to get reactions are considered.

Trending Hashtags method

Let's look at the individual components of the Trending hashtags method.

TweetSpout: Component used to issue the tweets in the topology.

TweetFilterBolt: The component reads the tweets and filters the tweets issued by TweetSpout. Filtering is performed only on those tweets that contain coded messages using the standard Unicode. Violation checks are also performed at this stage.

ParseTweetBolt: The component issues the filtered tweets as tuples with at least one hashtag present.

CountHashtagBolt: Processes the parsed tweets and counts each hashtag. This is done to get the hashtag and the number of references to it.

TotalRankerBolt: Ranking of all the counted hashtags is performed.

GeoLocationBolt: It identifies the location of the parsed tweet.

CountLocationHashtagBolt: It is similar to the CountHashtagBolt component with an extra location parameter.

RedisBolt: The component inserts data into Redis.

Overview of Twitter System Design

Twitter System Design

When a tweet is made, the API call hits the load **balancer**, and then that call hits the **Twitter writer**. The Twitter writer sends the copy of tweets to the **database**, **Apache storm** (for analyzing trending hashtags), the **fanout service** (to update the timeline), and the **search service** (for indexing).

If the user wants to search for something, an API call is made to the load balancer, and then the request is forwarded to the **timeline service**. The call is then transferred to the **search service**, where the searching operation is efficiently performed using the scatter and gather strategy. Similarly, when the user requests a home timeline or user timeline, the request

reaches the timeline service, which directly approaches **Redis**. Redis figures out the appropriate timeline and returns the result in **JSON** (JavaScript Object Notation) format.

HTTP push WebSocket is responsible for handling the real-time connection with the application. This service should be able to handle millions of connections at any given point in time.

ZooKeeper is a highly reliable coordination service for distributed components. Twitter runs about thousands of nodes in any given cluster for Redis. Most of the data is stored in Redis in big clusters, which requires coordination between the nodes. It also keeps track of which all servers are online/ offline and coordinates between the nodes accordingly.

Frequently Asked Questions

What is meant by caching?

Caching is a process in which copies of data are a temporary storage location (or cache) to have faster access to the data.

How can sharding of data be done to manage the gigantic data?

The sharding of data can be done based on the following parameters: userID, tweetID, and tweetID & create time.

What is the difference between horizontal and vertical scaling?

In horizontal scaling, more machines are added to the pool of resources.

In vertical scaling, more power is added to the existing machine.

What is meant by sharding?

Sharding is the horizontal scaling of a database system that is accomplished by breaking the database up into smaller chunks. These chunks are called “shards,” separate database servers containing a subset of the overall dataset.

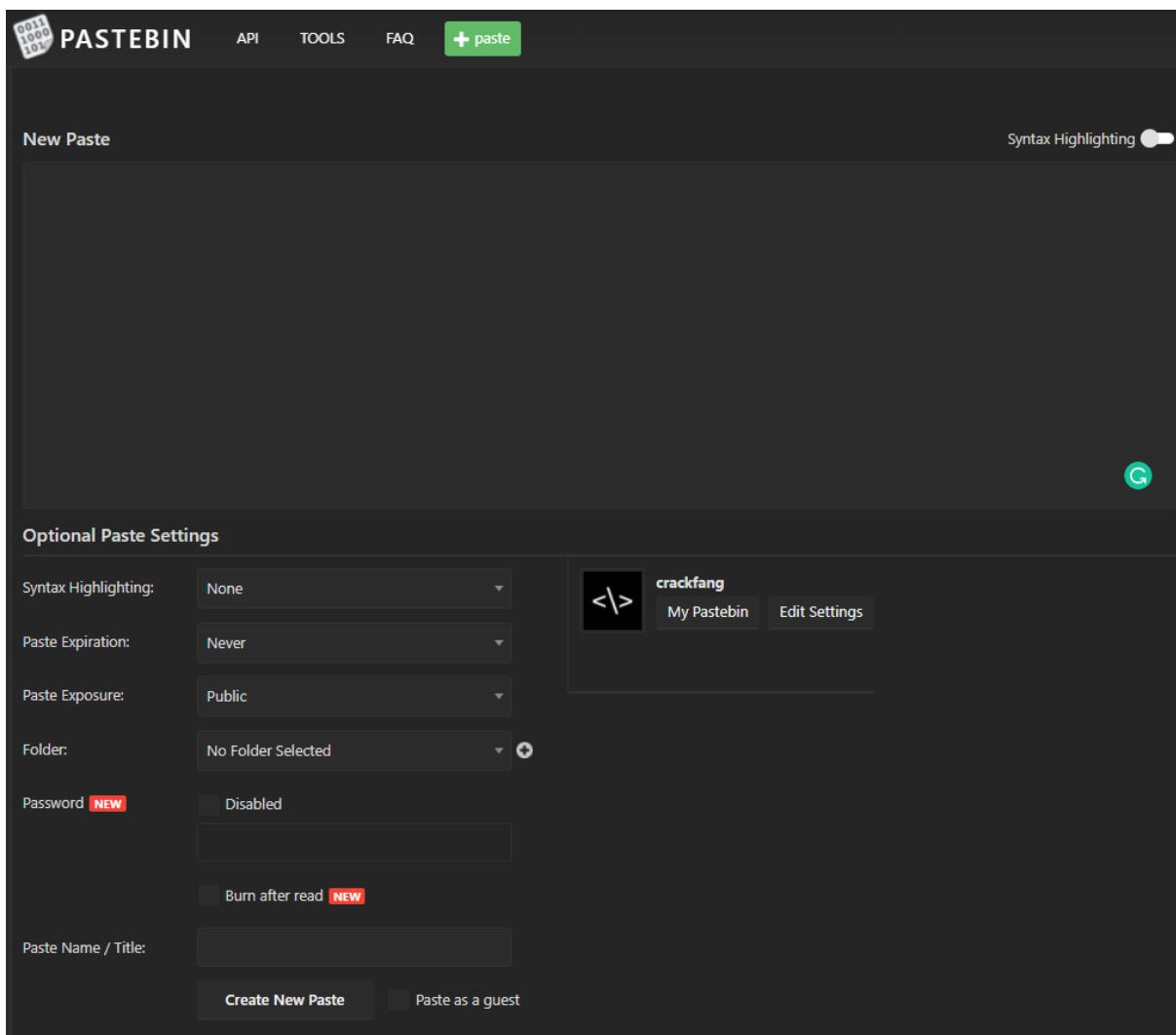
What is meant by the fanout approach?

Fanout approach is a way by which messages are broadcasted to multiple locations parallelly.

12. Designing Pastebin

Solution 1:

Similar Services: pasted.co, hastebin.com, chopapp.com



[Pastebin.com](#) like services enable users to store plain text or images over the network (typically the Internet) and generate unique URLs to access the uploaded data. Such services are also used to share data over the web quickly, as users would need to pass the URL to let other users see it.

If you haven't used pastebin.com before, please try creating a new 'Paste' there and spend some time going through different options their service offers. This will help you a lot in understanding this chapter better.

Requirements and Goals of the System

Our Pastebin service should meet the following requirements:

Functional Requirements

1. Users should upload or “paste” their data and get a unique URL to access it.
2. Users will only be able to upload text.
3. Data and links will automatically expire after a specific timespan; users should also specify expiration time.

4. Users should optionally be able to pick a custom alias for their paste.

Non-Functional Requirements

1. The system should be highly reliable, any data uploaded should not be lost.
2. The system should be highly available. This is required because if our service is down, users will not access their Pastes.
3. Users should be able to access their Pastes in real-time with minimum latency.
4. Paste links should not be guessable (not predictable).

Extended Requirements

1. Analytics, e.g., how many times a redirection happened?
2. Our service should also be accessible through REST APIs by other services.

Some Design Considerations

Pastebin shares some requirements with the URL Shortening service, but there are some additional design considerations we should keep in mind.

What should be the limit on the amount of text a user can paste at a time?

We can limit users not to have Pastes bigger than 10MB to stop the abuse of the service.

Should we impose size limits on custom URLs?

Since our service supports custom URLs, users can pick any URL they like, but providing a custom URL is not mandatory. However, it is reasonable (and often desirable) to impose a size limit on custom URLs to have a consistent URL database.

Capacity Estimation and Constraints

Our services would be read-heavy; there will be more read requests compared to new Pastes creation. Therefore, we can assume a 100:1 ratio between read and write.

Traffic estimates

Pastebin services are not expected to have traffic similar to Twitter or Facebook. Let's assume that we get 1 million new pastes added to our system every day. Assuming a 100:1 read:write ratio, this leaves us with 100 million reads per day.

New Pastes per second: $1M / (24 \text{ hours} * 3600 \text{ seconds}) \approx 10 \text{ pastes/sec}$

Paste reads per second: $100M / (24 \text{ hours} * 3600 \text{ seconds}) \approx 1200 \text{ reads/sec}$

Storage estimates

Users can upload a maximum of 10MB of data; commonly, Pastebin like services are used to share source code, configs, or logs. Such texts are not huge, so let's assume that each paste, on average, contains 100 KB.

At this rate, we will be storing 100 GB of data per day.

$$1M * 10KB = 100 \text{ GB/day}$$

If we want to store this data for 5 years, we would need a total storage capacity of 180 TB.

$$100 \text{ GB} * 365 \text{ days} * 5 \text{ years} \approx 180 \text{ TB}$$

With 1M pastes every day, we will have approximately 2 billion Pastes in 5 years.

$$1M * 365 * 5 \approx 2 \text{ Billion}$$

We need to generate and store keys to identify these pastes uniquely. If we use base64 encoding ([A-Z, a-z, 0-9, ., -]) we would need six letters strings:

$$64^6 \approx 68.7 \text{ billion unique strings}$$

If it takes one byte to store one character, the total size required to store 3.6B keys would be:

$$2 \text{ B} * 6 = 12 \text{ GB}$$

12 GB is negligible compared to 180 TB. To keep some margin, we will assume a 70% capacity model (meaning we don't want to use more than 70% of our total storage capacity at any point), which raises our storage needs up to 260 TB.

Bandwidth estimates

We expect 10 new pastes per second for write requests per second, resulting in 1 MB of ingress per second.

$$10 * 100 \text{ KB} = 1 \text{ MB/s}$$

As for read requests, we expect 1000 requests per second. Therefore, the total data egress (sent to users) will be 100 MB/s.

$$1000 * 100 \text{ KB} \Rightarrow 100 \text{ MB/s}$$

Although total ingress and egress are not big, we should keep these numbers in mind while designing our service.

Memory estimates

We can cache some of the hot pastes that are frequently accessed. Following the 80-20 rule, meaning 20% of hot pastes generate 80% of traffic, we would like to cache these 20% pastes.

Since we have 100 M read requests per day, to cache 20% of these requests, we would need:

$$0.2 * 100 \text{ M} * 100 \text{ KB} \approx 2 \text{ TB}$$

Feel free to play with this [Excel file](#) to get the estimates specific to your system.

Database Design

A few observations about the nature of the data we are going to store:

1. We need to store approximately 2 Billion records.

$$1 \text{ Million per day} * 365 \text{ days} * 5 \text{ years} \approx 2 \text{ Billion}$$

1. Each metadata object we are going to store would be small (less than 100 bytes).
2. Each paste object we store can be medium size (average 100KB, max 10 MB).
3. There are no relationships between records, except if we want to store which user created what Paste.
4. Our service is read-heavy. 100:1 read:write ratio.

Database Schema

We would need two tables, one for storing information about the Pastes and the other for users' data.

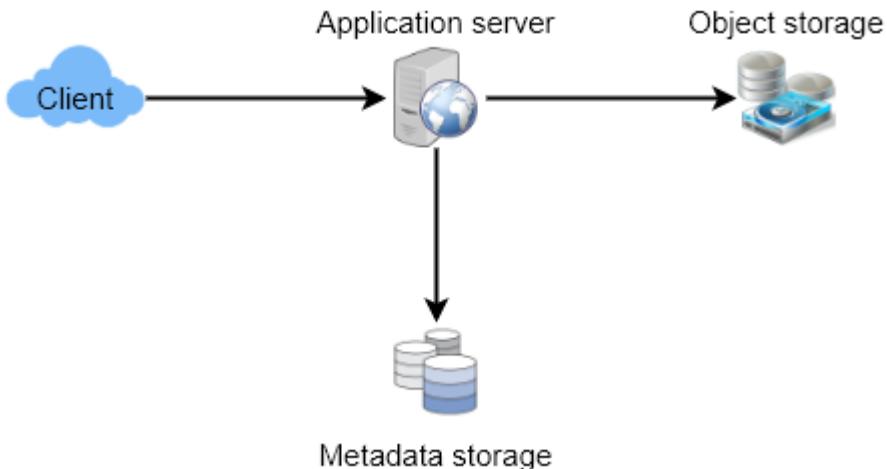
pastes		users	
hash	varchar(6)	user_id	int
user_id	int	name	varchar(32)
content_path	varchar(256)	email	varchar(32)
url	varchar(256)	created_at	timestamp
created_at	timestamp	date_of_birth	timestamp
expiration_date	timestamp	last_login	timestamp

High-Level Design

We need an application layer that will serve all the read and write requests at a high level.

The application layer will talk to a storage layer to store and retrieve data.

We can segregate our storage layer with one database storing metadata related to each paste, users, etc., while the other storing paste contents in some block storage or a database. This division of data will allow us to scale them individually.



Component Design

Application layer

Our application layer will process all incoming and outgoing requests. The application servers will be talking to the backend data store components to serve the requests.

How to handle a write request?

Upon receiving a write request, our application server will generate a six-letter random string, which would serve as the key of the paste (if the user has not provided a custom key). The application server will then store the contents of the paste and the generated key in the database. After the successful insertion, the server can return the key to the user.

One possible problem here could be that the insertion fails because of a duplicate key. Since we are generating a random key, there is a possibility that the newly generated key could match an existing one. In that case, we should regenerate a new key and try again. We should keep retrying until we don't see a failure due to the duplicate key. We should return an error to the user if the custom key they have provided is already present in our database.

Another solution to the above problem could be to run a standalone Key Generation Service (KGS) that generates random six letters strings beforehand and stores them in a database (let's call it key-DB).

KGS will make sure all the keys inserted in key-DB are unique. Whenever we want to store a new paste, we will take one of the already generated keys and use it. This approach will

make things quite simple and fast since we will not be worrying about duplications or collisions.

KGS can use two tables to store keys, one for keys not used yet and one for all the used keys. As soon as KGS gives some keys to any application server, it can move these to the used keys table. KGS can always keep some keys in memory so that whenever a server needs them, it can quickly provide them. As soon as KGS loads some keys in memory, it can move them to the used keys table. This way, we can make sure each server gets unique keys. If KGS dies before using all the keys loaded in memory, we will be wasting those keys. We can ignore these keys, given the vast number of keys we have.

Isn't KGS a single point of failure?

Yes, it is. To solve this, we can have a standby replica of KGS, and whenever the primary server dies, it can take over to generate and provide keys.

Can each app server cache some keys from key-DB?

Yes, this can surely speed things up. Although in this case, if the application server dies before consuming all the keys, we will end up losing those keys. This could be acceptable since we have 68B unique six letters keys, many more than we require. (We require only 2 Billion)

How to handle a paste read request?

Upon receiving a read request, the application service layer contacts the datastore. The datastore searches for the key, and if it is found, returns the paste's contents. Otherwise, an error code is returned.

Datastore layer

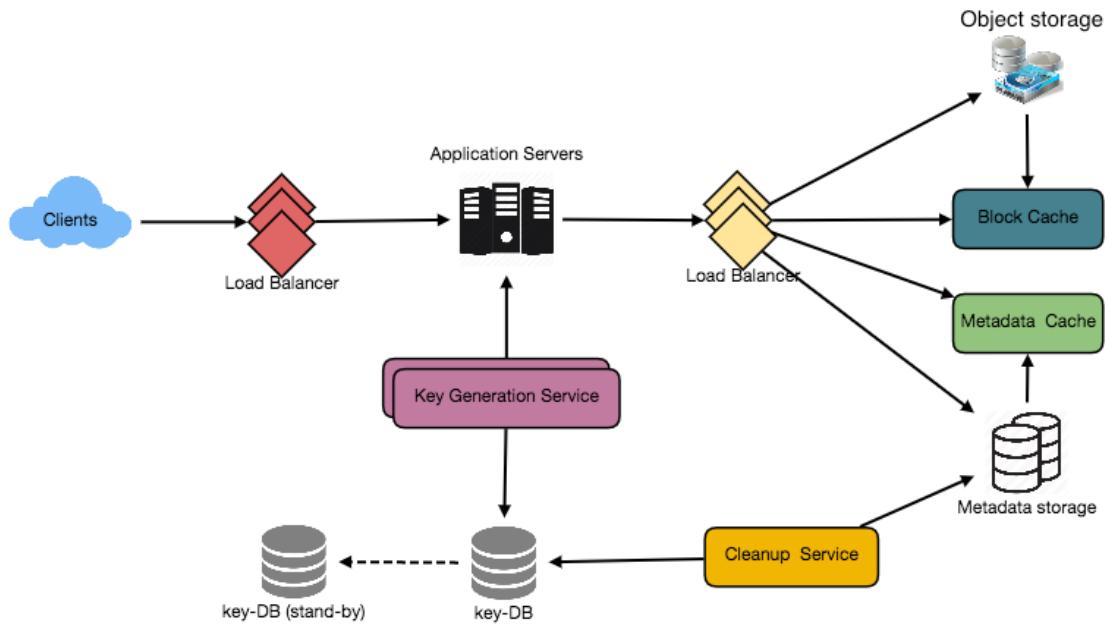
We can divide our datastore layer into two:

Metadata database

We can use a relational database like MySQL or a Distributed Key-Value store like Redis or Memcached. The former would be preferable for an established firm and the latter for a fast-growing startup. Refer to [this article](#) for more details on choosing the type of database.

Block storage

We can store our contents in a distributed key-value block storage to enjoy benefits offered by NoSQL like HDFS or S3. Whenever we feel like hitting our full capacity on content storage, we can quickly increase it by adding more servers.



Cache

We can cache pastes that are frequently accessed. We can use some off-the-shelf solution like Memcached that can store full-text contents with their respective hashes. The application servers, before hitting backend storage, can quickly check if the cache has desired paste.

How much cache should we have?

We can start with 20% of daily traffic, and we can adjust how many cache servers we need based on clients' usage patterns. As estimated above, we need 600GB of memory to cache 20% of the daily traffic. Since a **modern-day server can have 256GB of memory**, we can easily fit all the cache into three machines or use some smaller servers to store all these hot URLs.

Which cache eviction policy would best fit our needs?

When the cache is full, and we want to replace a link with a newer/hotter URL, how would we choose? The **Least Recently Used (LRU)** can be a reasonable policy for our system. Under this policy, we discard the least recently used URL first. Then, we can use a [Linked Hash Map](#) or a similar data structure to store our URLs and Hashes, keeping track of which URLs are accessed recently.

To further increase the efficiency, we can replicate our caching servers to distribute the load between them. For further details, refer to [this Caching article](#).

How can each cache replica be updated?

Whenever there is a cache miss, our servers would be hitting the backend database. Instead, we can update the cache and pass the new entry to all the cache replicas whenever this happens. This is also known as the Write-Thorough policy. Refer to [this Caching article](#)

for more details. Each replica can update its cache by adding a new entry. If a replica already has that entry, it can simply ignore it.

Load Balancer (LB)

We can add a Load balancing layer at three places in our system:

1. Between Clients and Application servers
2. Between Application Servers and Database servers
3. Between Application Servers and Cache servers

Initially, a simple Round Robin approach can be adopted; that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is that if a server is dead, LB will take it out of the rotation and stop sending any traffic. However, a problem with Round Robin LB is, it won't consider server load. Therefore, if a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load and adjusts traffic based on that. Refer to [this Load Balancer article](#) for more details.

Purging or DB cleanup

Should entries stick around forever, or should they be purged? If a user-specified expiration time is reached, what should happen to the paste?

If we chose to search for expired pastes to remove them continuously, it would put a lot of pressure on our database.

Instead, we can slowly remove expired pastes and do a lazy cleanup. Our service will ensure that only expired pastes will be deleted, although some expired pastes can live longer but will never be returned to users.

- Whenever a user tries to access an expired paste, we can delete the link and return an error to the user.
- A separate Cleanup service can run periodically to remove expired pastes from our storage and cache. This service should be very lightweight and scheduled to run only when the user traffic is expected to be less.
- We can have a default expiration time for each paste (e.g., two years).
- After removing an expired paste, we can put the key back in the key-DB to be reused.

Security and Permissions

Can users create private pastes or allow a particular set of users to access a paste?

We can store permission levels (public/private) with each paste in the database.

We can also create a separate table to store UserIDs that have permission to see a specific paste. If we are storing our data in a NoSQL key-value or wide-column database, the key for the table storing permissions would be the ‘Hash’ (or the KGS generated ‘key’), and the columns will store the UserIDs of those users that have permissions to see the paste.

If a user does not have permission and tries to access a paste, we can send an error (HTTP 401) back.

Solution 2:

Design Pastebin.com (or Bit.ly)

Design Bit.ly - is a similar question, except pastebin requires storing the paste contents instead of the original unshortened url.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **User** enters a block of text and gets a randomly generated link
 - Expiration
 - Default setting does not expire
 - Can optionally set a timed expiration
- **User** enters a paste's url and views the contents
- **User** is anonymous
- **Service** tracks analytics of pages
 - Monthly visit stats
- **Service** deletes expired pastes
- **Service** has high availability

Out of scope

- **User** registers for an account
 - **User** verifies email
- **User** logs into a registered account
 - **User** edits the document
- **User** can set visibility
- **User** can set the shortlink

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
- Following a short link should be fast
- Pastes are text only
- Page view analytics do not need to be realtime
- 10 million users
- 10 million paste writes per month
- 100 million paste reads per month
- 10:1 read to write ratio

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

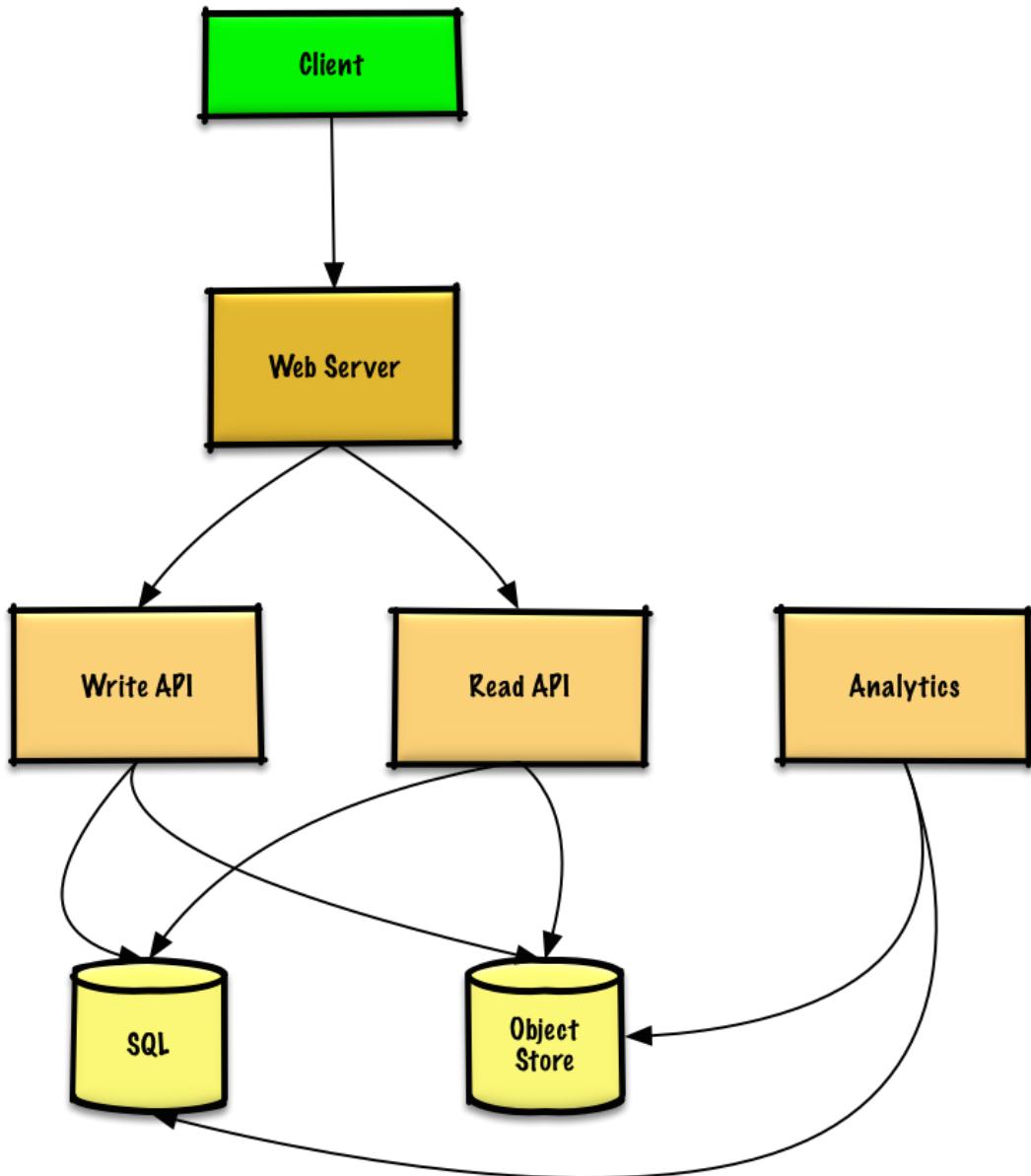
- Size per paste
 - 1 KB content per paste
 - shortlink - 7 bytes
 - expiration_length_in_minutes - 4 bytes
 - created_at - 5 bytes
 - paste_path - 255 bytes
 - total = ~1.27 KB
- 12.7 GB of new paste content per month
 - 1.27 KB per paste * 10 million pastes per month
 - ~450 GB of new paste content in 3 years
 - 360 million shortlinks in 3 years
 - Assume most are new pastes instead of updates to existing ones
- 4 paste writes per second on average
- 40 read requests per second on average

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Step 3: Design core components

Dive into details for each core component.

Use case: User enters a block of text and gets a randomly generated link

We could use a [relational database](#) as a large hash table, mapping the generated url to a file server and path containing the paste file. Instead of managing a file server, we could use a managed **Object Store** such as Amazon S3 or a [NoSQL document store](#).

An alternative to a relational database acting as a large hash table, we could use a [NoSQL key-value store](#). We should discuss the [tradeoffs between choosing SQL or NoSQL](#). The following discussion uses the relational database approach.

- The **Client** sends a create paste request to the **Web Server**, running as a [reverse proxy](#)
- The **Web Server** forwards the request to the **Write API** server
- The **Write API** server does the following:
 - Generates a unique url
 - Checks if the url is unique by looking at the **SQL Database** for a duplicate
 - If the url is not unique, it generates another url
 - If we supported a custom url, we could use the user-supplied (also check for a duplicate)
 - Saves to the **SQL Database** pastes table
 - Saves the paste data to the **Object Store**
 - Returns the url

Clarify with your interviewer how much code you are expected to write.

The pastes table could have the following structure:

```
shortlink char(7) NOT NULL
expiration_length_in_minutes int NOT NULL
created_at datetime NOT NULL
paste_path varchar(255) NOT NULL
PRIMARY KEY(shortlink)
```

Setting the primary key to be based on the shortlink column creates an [index](#) that the database uses to enforce uniqueness. We'll create an additional index on created_at to speed up lookups (log-time instead of scanning the entire table) and to keep the data in memory. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

To generate the unique url, we could:

- Take the [MD5](#) hash of the user's ip_address + timestamp
 - MD5 is a widely used hashing function that produces a 128-bit hash value
 - MD5 is uniformly distributed
 - Alternatively, we could also take the MD5 hash of randomly-generated data
- [Base 62](#) encode the MD5 hash
 - Base 62 encodes to [a-zA-Z0-9] which works well for urls, eliminating the need for escaping special characters
 - There is only one hash result for the original input and Base 62 is deterministic (no randomness involved)
 - Base 64 is another popular encoding but provides issues for urls because of the additional + and / characters
 - The following [Base 62 pseudocode](#) runs in O(k) time where k is the number of digits = 7:

```
def base_encode(num, base=62):
    digits = []
```

```
while num > 0
    remainder = modulo(num, base)
    digits.push(remainder)
    num = divide(num, base)
digits = digits.reverse
```

- Take the first 7 characters of the output, which results in 62^7 possible values and should be sufficient to handle our constraint of 360 million shortlinks in 3 years:

```
url = base_encode(md5(ip_address+timestamp))[:URL_LENGTH]
```

We'll use a public [REST API](#):

```
$ curl -X POST --data '{ "expiration_length_in_minutes": "60", \
"paste_contents": "Hello World!" }' https://pastebin.com/api/v1/paste
```

Response:

```
{
  "shortlink": "foobar"
}
```

For internal communications, we could use [Remote Procedure Calls](#).

Use case: User enters a paste's url and views the contents

- The **Client** sends a get paste request to the **Web Server**
- The **Web Server** forwards the request to the **Read API** server
- The **Read API** server does the following:
 - Checks the **SQL Database** for the generated url
 - If the url is in the **SQL Database**, fetch the paste contents from the **Object Store**
 - Else, return an error message for the user

REST API:

```
$ curl https://pastebin.com/api/v1/paste?shortlink=foobar
```

Response:

```
{
  "paste_contents": "Hello World"
  "created_at": "YYYY-MM-DD HH:MM:SS"
  "expiration_length_in_minutes": "60"
}
```

Use case: Service tracks analytics of pages

Since realtime analytics are not a requirement, we could simply **MapReduce** the **Web Server** logs to generate hit counts.

Clarify with your interviewer how much code you are expected to write.

```
class HitCounts(MRJob):
```

```
    def extract_url(self, line):
        """Extract the generated url from the log line."""
        ...
```

```
    def extract_year_month(self, line):
        """Return the year and month portions of the timestamp."""
        ...
```

```
    def mapper(self, _, line):
        """Parse each log line, extract and transform relevant lines.
```

Emit key value pairs of the form:

```
(2016-01, url0), 1
(2016-01, url0), 1
(2016-01, url1), 1
"""
url = self.extract_url(line)
period = self.extract_year_month(line)
yield (period, url), 1
```

```
def reducer(self, key, values):
    """Sum values for each key.
```

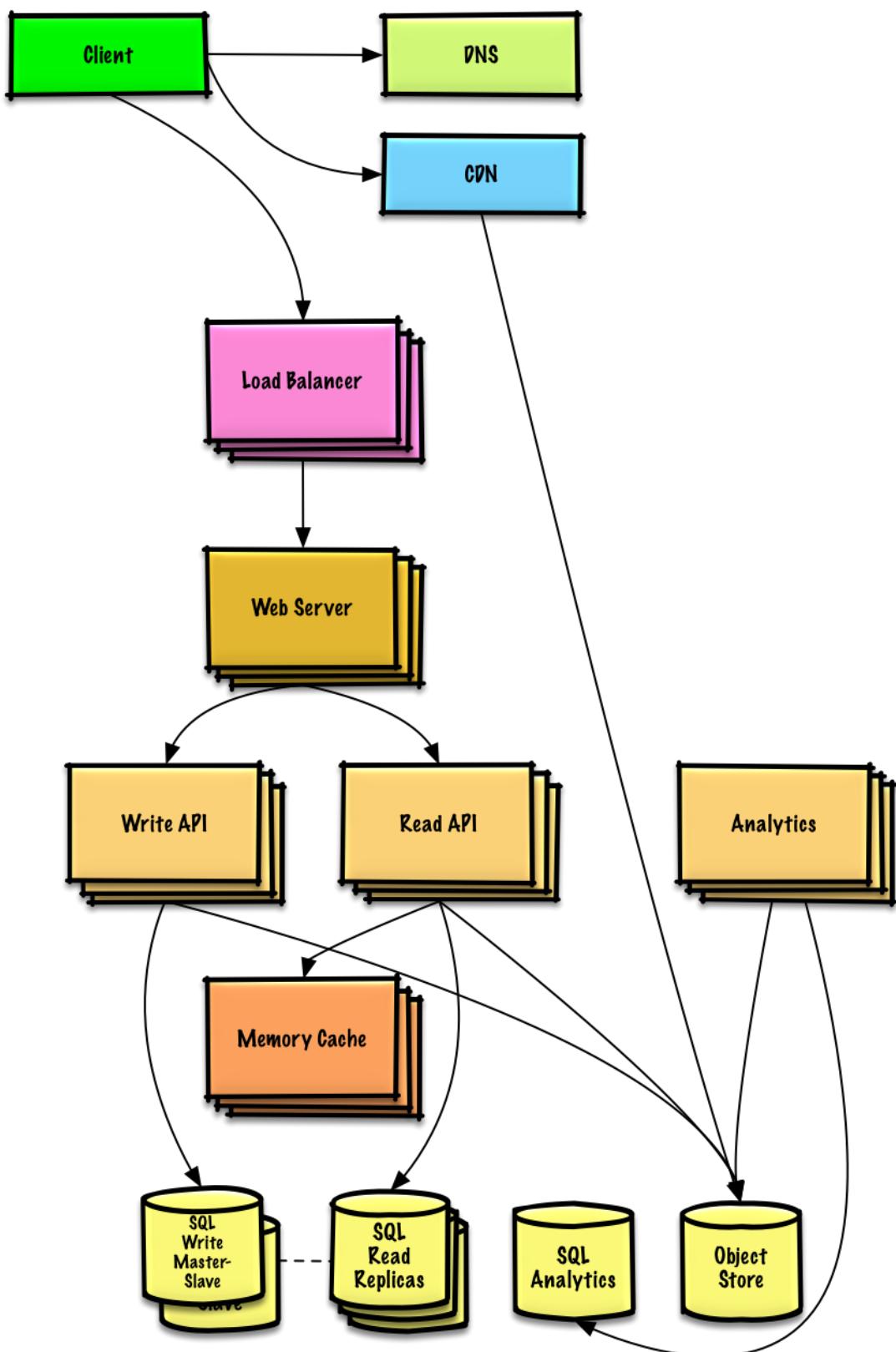
```
(2016-01, url0), 2
(2016-01, url1), 1
"""
yield key, sum(values)
```

Use case: Service deletes expired pastes

To delete expired pastes, we could just scan the **SQL Database** for all entries whose expiration timestamp are older than the current timestamp. All expired entries would then be deleted (or marked as expired) from the table.

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.



Important: Do not simply jump right into the final design from the initial design!

State you would do this iteratively: 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers?** **CDN?** **Master-Slave Replicas?** What are the alternatives and **Trade-Offs** for each?

The **Analytics Database** could use a data warehousing solution such as Amazon Redshift or Google BigQuery.

An **Object Store** such as Amazon S3 can comfortably handle the constraint of 12.7 GB of new content per month.

To address the 40 average read requests per second (higher at peak), traffic for popular content should be handled by the **Memory Cache** instead of the database. The **Memory Cache** is also useful for handling the unevenly distributed traffic and traffic spikes. The **SQL Read Replicas** should be able to handle the cache misses, as long as the replicas are not bogged down with replicating writes.

4 average paste writes per second (with higher at peak) should be doable for a single **SQL Write Master-Slave**. Otherwise, we'll need to employ additional SQL scaling patterns:

- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

We should also consider moving some data to a **NoSQL Database**.

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)

- [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#)
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know.](#)

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

Solution 3:

Seems incomplete ans from source

Step 1: Outline Use cases and Constraints

Functional requirements

- User enters block of text and gets a randomly generated url

- User enters a paste's url and view its content
- User login is optional

Non-Functional requirements

- Expiration
 - Default settings doesn't expire
 - User specifies a time expiration
- Service tracks analytics of pages
 - Like monthly visit stats, countries, platform etc.
- Service should delete expired pastes
- Service has high availability

Constraints

- Traffic is not evenly distributed
- Following a short link should be fast
- Pastes are text only
- Page view analytics do not need to be realtime

Assumptions and Estimation

- 10 million users
- 10 million paste writes per month
- 100 million paste reads per month
- 10:1 read to write ratio

Calculate usage

- Size per paste
 - Roughly 1.5 KB
- New paste per month
 - Roughly 15 GB = 10M new paste * 1.5 KB per paste
 - About 180 GB per year = 15GB * 12
 - Assume most are new pastes instead of updates to existing ones
- 4 paste writes per second on average
 - 10M / 30 days 24 hours 3600 secs
- 40 read requests per second on average

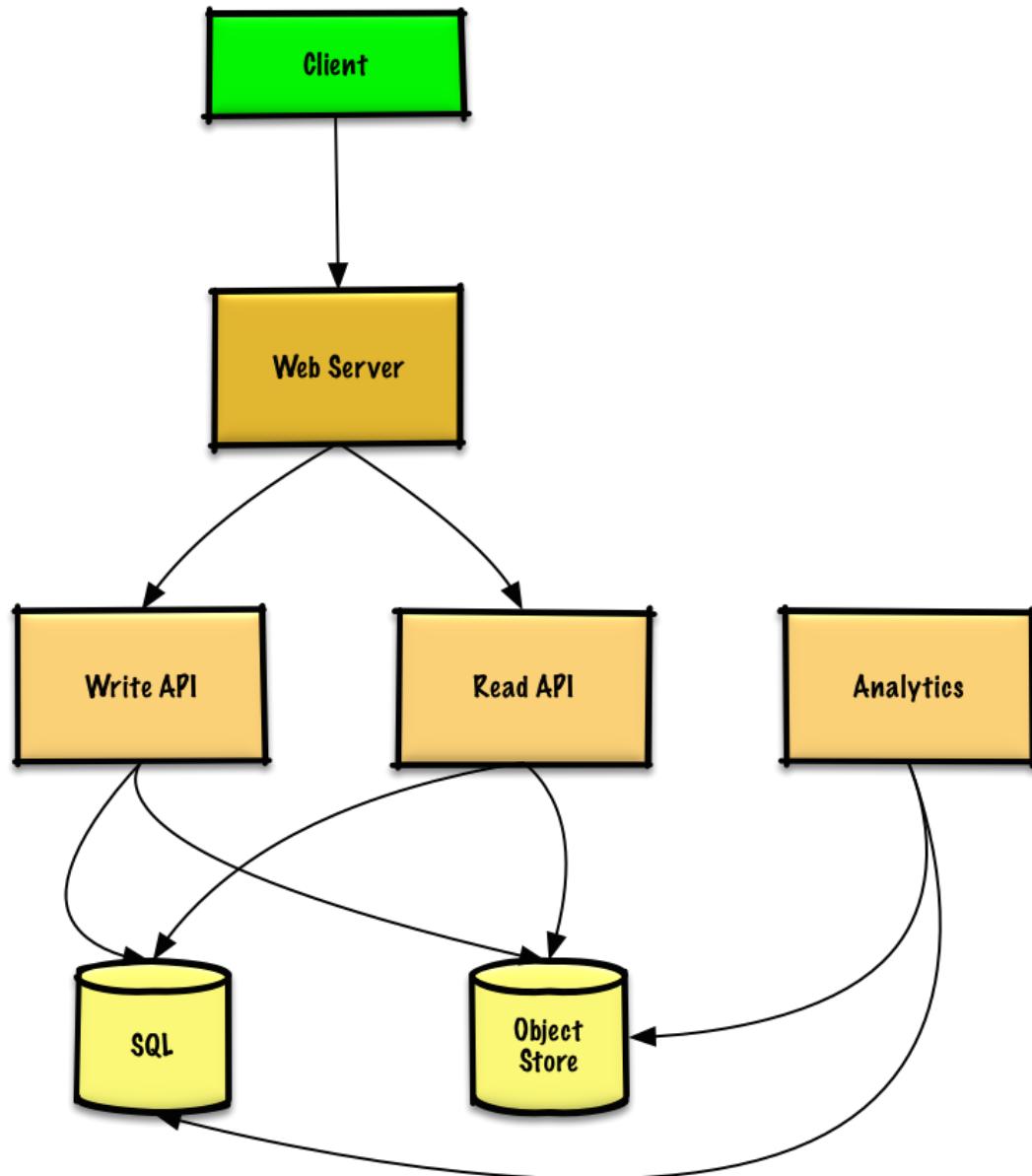
Handy conversion guide

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.

Step 3: Design core components



Use case: User enters a block of text and gets a randomly generated link

We could use a relational database as a large hash table, mapping the generated url to a file server and path containing the paste file.

Instead of managing a file server, we could use a managed Object Store such as Amazon S3 or a NoSQL document store.

An alternative to a relational database acting as a large hash table, we could use a NoSQL key-value store. **We should discuss the tradeoffs between choosing SQL or NoSQL.**

The following discussion uses the relational database approach:

- The Client sends a create paste request to the Web Server, running as a **reverse proxy**
- The Web Server forwards the request to the Write API server
- The Write API server does the following:
 - Generates a unique url
 - Checks if the url is unique by looking at the SQL Database for a duplicate
 - If the url is not unique, it generates another url
 - If we supported a custom url, we could use the user-supplied (also check for a duplicate)
 - Saves to the SQL Database pastes table
 - Saves the paste data to the Object Store
 - Returns the url

Clarify with your interviewer how much code you are expected to write.

We'll create an index on shortlink and created_at to speed up lookups (log-time instead of scanning the entire table) and to keep the data in memory. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer

Generate Unique URL

- Take the MD5 hash of the user's ip_address + timestamp
 - MD5 is a widely used hashing function that produces a 128-bit hash value
 - MD5 is uniformly distributed
 - Alternatively, we could also take the MD5 hash of randomly-generated data
- Base 62 encode the MD5 hash
 - Base 62 encodes to [a-zA-Z0-9] which works well for urls, eliminating the need for **escaping special characters**
 - There is only one hash result for the original input and and Base 62 is deterministic (no randomness involved)
 - Base 64 is another popular encoding but provides issues for urls because of the additional + and / characters
 - Take the first 7 characters of the output, which results in 62^7 possible values and should be sufficient to handle our constraint of 360 million shortlinks in 3 years:

We'll use a public REST API:

```
$ curl -X POST --data '{ "expiration_length_in_minutes": "60", \
"paste_contents": "Hello World!" }' https://pastebin.com/api/v1/paste
```

Response:

```
{
```

```
        "shortlink": "foobar"  
    }
```

For internal communications, we could use **Remote Procedure Calls**.

Use case: User enters a paste's url and views the contents

- The Client sends a get paste request to the Web Server. The Web Server forwards the request to the Read API server
- The Read API server does the following:
 - Checks the SQL Database for the generated url
 - If the url is in the SQL Database, fetch the paste contents from the Object Store
 - Else, return an error message for the user

REST API:

```
$ curl https://pastebin.com/api/v1/paste?shortlink=foobar
```

Response:

```
{  
    "paste_contents": "Hello World"  
    "created_at": "YYYY-MM-DD HH:MM:SS"  
    "expiration_length_in_minutes": "60"  
}
```

Use case: Service tracks analytics of pages

Since realtime analytics are not a requirement, we could simply MapReduce the Web Server logs to generate hit counts.

Use case: Service deletes expired pastes To delete expired pastes, we could just scan the SQL Database for all entries whose expiration timestamp are older than the current timestamp. All expired entries would then be deleted (or marked as expired) from the table.

Step 4: Scale the design

Solution 4:

Pastebin is an online content hosting service where users can store text files(i.e source code snippets, etc) called "pastes" over the internet, either anonymously or attached to an account. Each paste has its unique URL and can be shared with other users for quick access. Pastebin was created in 2002 and has still managed to remain relevant and available today with over 17 million daily active users.

Key Features/Use Cases

Functional Features:

These are the features that are absolutely essential for the user. They are the core provisions of the system. They are:

- Users should be able to upload plain text ("Pastes")
- Upon uploading the paste user should be provided with a unique URL to access it.
- Users should only be able to upload text.
- Users should be able to set an expiry time for URLs, after which the content is deleted. By default, a paste should not expire unless otherwise specified
- Users are anonymous
- Users will get access to a paste's content when they enter the paste's URL

Non-Functional Features

These are quality constraints that the system must satisfy in order to ensure that system as a whole performs well. They are:

- The system should be highly reliable, data uploaded should not be lost.
- The system should be highly available. So that users can always access their pastes.
- Users should be able to access their Pastes in real-time with minimum latency.
- Analytics: Monthly visit statistics, etc.

Constraints and Capacity Assumptions

Key assumptions

- 17 million users
- 10:1 read to write ratio
- 10 million paste writes per month
- 100 million paste reads per month
- Fast paste retrieval

Traffic Estimates

We can expect our traffic to not be as intensive as a web application like Twitter or Facebook. With our modest estimate of 100 million paste reads per month and a 10 to 1 read, write ratio. We get:

- an average paste read rate of 39 reads/sec ($100,000,000 / 30 * 24 * 3600$)
- an average paste creation rate of 4 writes/sec ($10,000,000 / 30 * 24 * 3600$)

Storage Estimates

We can set a limit of at most 10Mb per paste content, but on average we can estimate that a paste content size should be about 10kb plus the size of all the other metadata attached to the paste, like short link, created_at, paste_path, etc. Bringing the average paste size to be about 11kb.

This sums up to about 110 GB of new pastes to be stored every month and about 3.96 TB of paste in 3 years.

Diving into The Design

High-Level Design

At the high level, we are going to have an application layer and a storage layer. The application layer will be made up of a web server that will forward requests to the read or write server. The storage layer will be made up of a relational database for storing meta information and an object store/file server for storing content.

Core Component Design

Handling Write Requests (New Paste Creation)

Upon receiving a written request, we will generate a unique URL that will act as the key to the paste's content. In essence, we use a SQL database as a hashtable, we store the unique URL as the key along with meta information like creation time and expiration date and the most important part, a link to where the object content is stored(a file path if a file server is used or the URL to the object if we use a managed object store like amazon s3)

In essence, when the Client sends a create paste request to the webserver

- The web server forwards it to the write Server
- The write server first generates a unique URL maybe by base64 encoding the md5 hash of the current timestamp and a random nonce (the client's IP address).
- We check for the uniqueness of the URL by checking with the DB and keep on regenerating URLs until we get a unique one.
- The generated Url is then inserted into the SQL table along with other meta information
- The Paste data is then stored in an object store
- Finally it returns the newly created URL

Handling Reads

When a client sends a read request (pastes a URL) the web server forwards it to the read server.

- The read server queries the SQL database for the URL
- If the URL exists the paste content is retrieved from the datastore
- else it returns an error

Dealing with expired pastes

We can apply a lazy strategy to avoid overwhelming the database server with frequent deletions. We can check for expiration when a paste read occurs, if a paste has expired we can mark it as such and return an error. We can then have a cron job or service that periodically deletes the expired pastes from the database at periods of inactivity.

Scaling The System

We will need to scale our system to be able to handle our estimated numbers and still ensure high availability and consistency. We will have to horizontally scale and have multiple webservers, read servers and write servers. In addition to that, there are other considerations to be made.

Caching

We can introduce caching to deal with lots of reads. We have already estimated that we will get about 100 times as many reads compared to writes. We can then speed up reads by caching the more popular pastes in memory. So our read servers will first check the cache for a paste before hitting the DB.

Some factors to have in mind:

- We have to ensure that our cache remains consistent with original content in the case of modifications
- We have to provide a mechanism for replacing entries in the cache when it gets full. We can use the LRU (Least Recently Used) policy to replace pastes, that way we can always have the most popular pastes in our cache

Load Balancing

We can introduce a load balancer between the client and webserver to prevent a single point of failure modes and to distribute the traffic load efficiently between servers

Database

We can add extra SQL Read Replicas to scale and handle cache misses. Since we have a relatively low amount of writes (paste creations) a single SQL Write Master-Slave should suffice.

Since we are using Amazon's S3 object store for storing content it can comfortably scale to handle our size estimations

For Analytics, we could use existing solutions to scale, like Google BigQuery and Amazon Redshift.

13. Designing Twitter Search

Solution 1:

Functional Requirements

- Users must be able to search tweets— show the top 100 results.
- Search results need not be personalized i.e. it is expected that the same search query will return the same results for 2 different users, simultaneously querying the system.
- Search results should show the latest and trending tweets.

- Search results should be ranked in reverse chronological order.

Non Functional Requirements

- **Availability** — Search functionality should be always available
- **Low Latency** — Search results should be returned with very low latency (50ms)
- **Reliability** — Latest tweets should be available for search within 10s of creation.

Traffic and Throughput Requirements

- Number of DAU = 500 million
- Number of search queries per second = 1 million
- Average length of search queries = 10
- Number of tweets per second = 10K
- Peak number of tweets per second = 1 million
- Maximum length of tweets = 140 characters

The “On My Computer” Approach

Idea is to create an **inverted index of the terms** in the tweets mapping to the Tweet ids.

Assuming that we have a service for storing and retrieving tweets, when a new tweet comes in, first insert it into the tweet store, retrieve the tweet_id as well as the original tweet, then tokenize the tweet into words or terms.

Create a HashMap A: term -> [list of tweet_ids]

Update the list of tweet_ids by appending the latest tweet at the end of the list corresponding to the term.

For a given search query, again tokenize it into terms, lookup HashMap A to retrieve multiple lists of tweet_ids, one corresponding to each term.

Then do an union operation on the tweet_ids and return 100 tweet_ids to the tweet service which will find the tweets corresponding to these tweets.

How to ensure time sorted list of tweets ?

The tweet_id is comprised of 2 components: UNIX Timestamp in Milliseconds (41 bits) + 10 bit auto-incrementing integer.

Since there could be 1 million tweets per second i.e. 1000 tweets per second, thus in the same millisecond there could be 1000 different tweets. We can store them in a 10 bit (1024 possible integers) integer.

Thus tweet_ids are already sorted by time in the above list.

What is the data structure for storing the list of tweet_ids ?

Note that tweets can be also deleted by accounts.

One simple data structure that can allow O(1) insertion and deletion is doubly linked list.

But in Linked List memory is not contiguously assigned (random memory access) and the memory overhead for creating a Node pointer object is high.

To avoid referring the term in multiple places. We will maintain an auto-incrementing term_id. Thus we have the following HashMaps:

A: term -> term_id

B: term_id -> [id_0 -> id_1 -> id_2 -> -> id_M] (doubly linked list)

C: term_id -> DLL Tail Address

D: hash(term_id+tweet_id) -> DLL Node Address

The last HashMap is required in order to delete a Node in O(1) time complexity from the inverted index.

The 3rd HashMap is required to get tweet ids in reverse chronological order.

The other option is using **arrays** which allows sequential memory access which is faster as compared to random access.

But the drawback with array based approach, is that we do not know beforehand how many tweet_ids will be there for a term. If we pre-allocate a small array, then if the number of tweet_ids is more, then we have to either increase the length of the array or create a new array to store the future elements.

If the size of the arrays are small then we have to resize too often. All reads and writes are blocked during the time we will be migrating to a larger array. But if the size of the array is too large then there will be lots of wasted memory addresses which will have to be cleaned up explicitly with a GC before we run into OOM.

One strategy is to create array slices of size 1, 2, 4, 8, 16, ... and so on.

For each term, we have K slices. If the term occurs for 1st time, then we create the first slice of size 1. If the term occurs next time, then we create a slice of size 2 and insert the new tweet_id in that slice. Similarly if the term occurs in more than 7 tweets, we create a new slice of slice 8 and insert the new tweet_id in the first empty slot from the beginning and so on.

Note that the slices need not be contiguous in memory as they are created as and when required whereas each slice is contiguously laid out in memory.

To track which slice and which index in the slice to insert the new tweet_id, we can maintain a HashMap: term -> (curr_slice_number, curr_position)

The slice_number and position are updated with each insertion of term in the inverted index.

A: term -> term_id

B: term_id -> [[id_0], [id_1, id_2], [id_3, id_4, id_5, NULL]]

C: term_id -> (curr_slice_number, curr_position)

But deletion from inverted index is not efficient with this strategy.

In order to support efficient deletion and editing of tweets, we can go ahead with the DLL approach.

What happens when a tweet is edited ?

When a term is changed in the tweet, for the new term, standard insertion process is followed.

For the terms removed, for each of those terms and tweet_id, lookup the HashMap:

hash(old_term_id+tweet_id) -> DLL Node Address

To fetch the node pointers in the inverted index. Then delete those nodes from the DLLs to update the inverted index.

This multistep process for a single term is a single transaction and thus for any failure, we have to undo all previous changes accordingly.

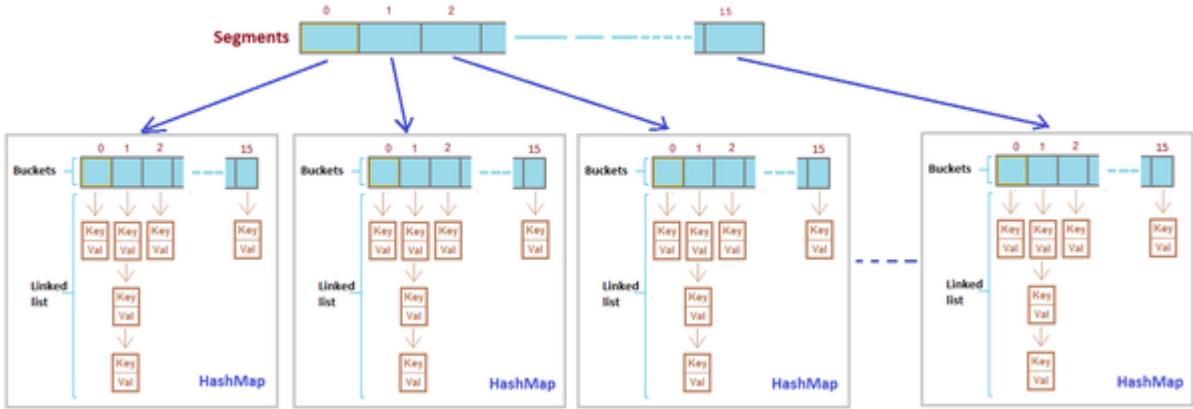
How to handle concurrent read-write ?

If one process is updating the inverted index for a term and another process wants to read the term simultaneously, there are few edge cases:

- New tweet_id was added by the write process but **term_id-> DLL Tail Address** is not updated yet, thus read process will read from old tail backwards and will miss the newly added node.
- Tweet edited and before write process could update inverted index, read process reads the removed node in the inverted index and assumes that the term is present.

Since mutexes are acquired at the HashMap level and not at individual key level, thus the entire Map becomes unavailable during writes. Assuming we have 10K writes per second, that would slow down all reads.

We can use something similar to the **ConcurrentHashMap** implementation in JAVA.



Segments in Concurrent HashMap

Divide the HashMap arrays into **equal sized segments**.

For e.g. if the HashMap array is of length 1024, then we can have 32 segments of size 32 each. These segments are independent.

During write only one segment will be in use by a thread and thus just acquire lock for that segment only whereas all the other 31 segments are free to be read/write by the other threads.

Thus for an index j in the HashMap array, the segment is obtained by performing $\text{int}(j/32)$ and the index within the segment is $j \% 32$.

During addition of a new tweet_id, for each term_id acquire write lock on the segment $\text{int}(\text{hash}(\text{term_id})/32)$.

During insertion, the lock is acquired on the hashmap “term_id-> DLL Tail Address” in the appropriate segment before writing the inverted index so that all threads trying to get the tail for term_id will wait for the lock to release, meanwhile any other threads that accesses segments other than $\text{int}(\text{hash}(\text{term_id})/32)$, will be able to read/write without waiting.

```
lock segment  $\text{int}(\text{hash}(\text{term\_id})/32)$  of hashmap term_id-> DLL Tail Address {
    get tail node
    update inverted_index by adding new node and update tail
    update "term_id-> DLL Tail Address" hashmap with new tail
    release_lock()
}
```

During edit of a tweet_id, for each old_term_id acquire write lock on the segment $\text{int}(\text{hash}(\text{old_term_id})/32)$ for all the following hashmaps:

term_id -> [id_0 -> id_1 -> id_2 -> -> id_M]

term_id -> DLL Tail Address

hash(term_id+tweet_id) -> DLL Node Address

Note: we need to lock all the hashmaps because the tail may also be updated during edits.

```
lock segment int(hash(old_term_id)/32) of all hashmaps {  
    update tweets table for tweet_id  
    get relevant node address from hash(old_term_id+tweet_id)  
    delete node in inverted index  
    update tail node if required in "term_id-> DLL Tail Address"  remove node from  
hash(old_term_id+tweet_id)  
    release_lock()  
}
```

What are the drawbacks of the above approach ?

If the system crashes all HashMaps are lost thus system becomes unavailable.

Number of tweets per second = 10K

Number of tweets for 5 years = 1.6 trillion

Assuming a power law distribution of terms i.e. say if the highest occurring term occurs in x tweets, then the 2nd highest occurs in m^x tweets for $m < 1$, the next highest in m^{2x} tweets and so on.

Thus $x + xm + xm^2 + xm^3 + \dots + xm^{k-1} = 1.6$ trillion for k terms

$x(1-m^k)/(1-m) = 1.6$ trillion

For large enough k , $x/(1-m) = 1.6$ trillion

For $x = 1$ million (we can always exclude all terms which occurs in more than 1 million tweets), we have $m = 0.999999375$

Also $xm^{k-1} \geq 1$ (i.e. at-least one tweet) or k is approximately 22 million

Thus number of unique terms is approximately equal to 22 million (i.e. term_id is of 25 bits)

Assuming that average number of terms per tweet is 15.

Size of the inverted index = Number of terms*25 bits + Number of tweets*15*51 bits = 22 million*25 bits + 1.6 trillion*15*51 bits = 140TB.

140TB is very huge to hold on a single RAM.

Distributed Hash Tables

How many instances of the inverted index are required to store 140TB ?

Assuming 16GB machine, total partitions = $140*1024/16 = 8960$.

How to partition the inverted index ?

Assign integers 0 to K-1 to the K partitions. Then based on the value of P = hash(term_id) % K, assign term_id to partition P.

How many replicas for the inverted index ?

Using replicas serves 2 purposes:

1. Fault tolerance — Each partition has multiple copies thus even if one copy (machine) crashes other copies can still serve the request.
2. Load Balancing — With high throughput of 1 million QPS, one machine for a partition may not be sufficient. If one machine can handle X QPS, then with M machines we can handle M*X QPS.

The expected search QPS is 1 million at peak. Assuming that we have 8 core CPU machines and search latency of 50ms. Then QPS served from one machine is:

QPS single machine = $8 * 1000 / 50 = 160$

To serve 1 million QPS, we need minimum of $1 \text{ million} / 160$ machines i.e. 6250 machines.

But earlier we have seen that to store 140TB we need at-least 8960 machines (one partition of 16GB in each machine). Thus we have 8960 partition and in order to achieve fault tolerance, lets say we have 3 replicas for each partition thus in total we have $8960 * 3$ machines = 27K

With 27K machines, throughput we can achieve is $27K * 160 = 4.3$ million QPS which is good enough.

What is the replication strategy for the inverted index ?

For each partition we should have a single master and 2 replicas.

During writes, all writes happen at the master node, which is then replicated to the 2 replicas. All reads happen from both master and replicas.

If we have all master nodes (multi-master) i.e. writes can happen in any of the 3 nodes, then there could be inconsistencies. It could happen that 2 concurrent requests for the same term_id sees different states (in different master nodes) and thus each process will update based on their master state and this could lead to conflicts.

But indexing is faster with multi-master since the tweets can be indexed into the nearest inverted index server from the user. But given a delay of 10s, we can live with single master for now.

Although within a single master node we can handle concurrency as seen above.

With a single master node, all writes are consistent and there are no issues related to conflicts. We can say that a write is successful only if the master is updated successfully and any one replica node is updated also.

During read, we read from any 2 nodes (master+replica1 or master+replica2 or replica1+replica2). Then based on whichever has the higher last_updated timestamp, we consider results from that instance.

The choice of at-least 2 nodes for write and 2 nodes for reads is that during write even if one node is not updated successfully (maybe it went offline), during reads at-least one node is there which was successfully updated during writes.

How tweets are ingested ?

Tweets are ingested into a Kafka queue for processing it into the inverted index. The use of queue is to make the **indexing flow asynchronous** because clients need not have to wait for the inverted index to be sucessfully updated before getting a response that their tweet is sucessful.

With an expected delay of 10s between tweet ingested and tweet indexed, using some queueing theory, it can be shown that the delay is expressed as:

delay = $1/(QPS \text{ of consumer} - QPS \text{ of producer})$

Assuming QPS of producer is 10K (10K writes per second)

QPS of consumer = $1/\text{delay} + 10K/s = 10K/s$

Assuming each consumer can handle 1000 QPS, we need 10 consumers to handle 10K tweets per second.

How to generate IDs in distributed manner ?

Note that tweet ids are being generated by distributed system instead of a single machine. With our earlier approach of using IDs of the form:

UNIX Timestamp in Milliseconds (41 bits) + 10 bit auto-incrementing integer.

It might happen that 2 machines generate the same ID. This could happen if 2 machines writing the new tweet at the same millisecond and both reads the highest auto-incrementing integer as X before both computing X+1 instead of one of them computing X+2.

This could be resolved by using another field in the ID formula which is the machine ID. We will keep 13 bits for the machine ID i.e. we are assuming that we will need maximum 8192 machines for distributed ID generation.

UNIX Timestamp in Milliseconds (41 bits) + 10 bit auto-incrementing integer + 13 bit machine ID = Total 64 bits.

Thus if 2 machines generate the same ID one of them will have higher machine ID than the other thus breaking the tie.

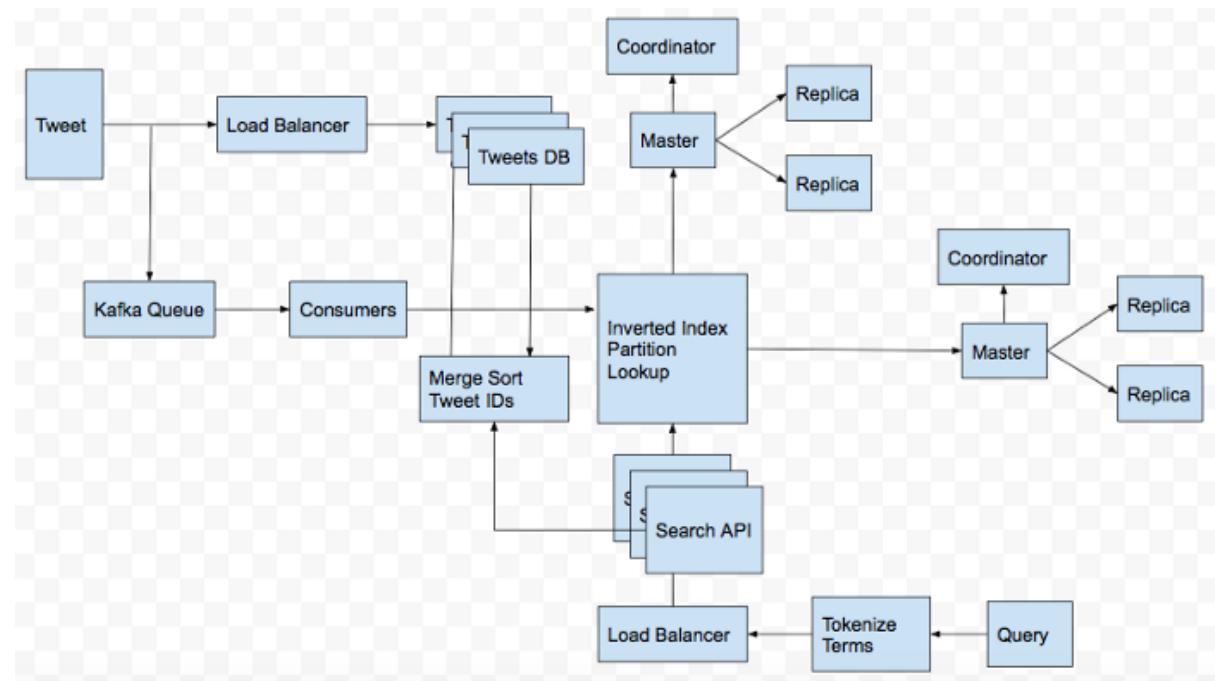
How to persist the inverted index and the other maps ?

Although we have taken care of fault tolerance by the use of replication, but it could still happen that all replicas for a partition crashes leading to loss of data for search.

To overcome this, we can consider the following approaches:

- Recreate the partition from tweets table. This can take somewhere from few hours to few days depending on the size of data. Thus the system will not work correctly during this period.
- Run periodic jobs to persist the partition in a filesystem or DB. The use case for DB is not really there as we need to scan all rows to recreate the partition. Thus storing each row of partition as a row in a file seems more cost effective as well as useful.

The Pipeline



Useful Resources

- <http://notes.stephenholiday.com/Earlybird.pdf>
- <https://www.javainuse.com/java/javaConcurrentHashMap>
- <https://dzone.com/articles/how-concurrenthashmap-works-internally-in-java>
- <https://web.mit.edu/modiano/www/6.263/lec5-6.pdf>

Solution 2:

Requirements

- Let us assume that Twitter has 1.5 billion total users, with 800 million daily active users.
- On average, Twitter gets 500 million tweets every day.
- The average size of a tweet is 300 bytes.
- Let us assume there will be 500M searches every day.

- The search query will consist of multiple words combined with and or.

Capacity Estimation and Constraints

We have 500 million tweets, and each tweet is 300 bytes. Total space required is
 $500 \text{ million} * 300 \text{ bytes} = 150 \text{ GB/day}$

System APIs

We can have soap or rest API to expose our search APIs

```
search(api_dev_key,search_terms,max_results_to_return,sort,page_number)
```

This API will return a list of tweets matching the search_terms.

Parameter

api_dev_key: API developer key

search_terms: A String to be searched

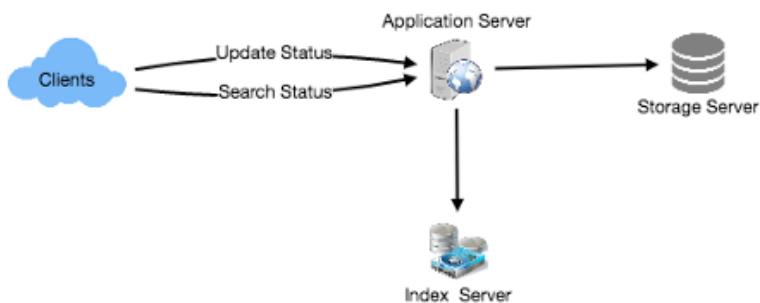
max_results_to_return : Number of tweets to return

sort: Latest First or most liked or best matched

page_number: This will specify the page_number you want to display

High-Level Design

We will store all the status in the storage server and also build an index server that can keep track of which word appears in which tweet. This index will help us quickly find tweets that users are trying to search for.



High Level Design for twitter search

DB Design

We can use a NoSQL database here as we have we need to scale the system and store tweets in a distributed environment. For the demonstration of this example, we will be using MYSQL, where we will shard the DB based on tweetid. So we will feed Tweets.id to hash function which will help us find the storage server

Table structure will look like as follows:

Tweets		Users	
id	int	id	int
content	varchar(150)	user_name	varchar
created_at	timestamp	number_of_tweets	int
user_id	int		
likes	int		
longitude	int		
latitude	int		

Table Structure for storage of tweets

Table Structure for storage of tweets

Now let us built the most complicated Index Server. Since our tweets consist of words, we will build our index across words. So indexes in most layman terms are nothing just distributed hash table where the key is the word, and the set of tweet id that contains the words are the values. There are approx 250k words in English and approx 250 k nouns, so there will be 500 k indexes. If we assume, there will be 300B tweets per year. We will get 600 B tweets for two years. if we assume each tweet id to be 5bytes then to store all tweet id we will need :

$$600B * 5 \text{ bytes} = 3000 \text{ GB}$$

If we assume, there are 40 words in a tweet, and if we remove a, an, the and other stop words and assume only 20 words contain information so they only those words need to be indexed. It concludes that each tweet id will be stored 20 times in our index. So the space required will be

$$(20 * 3000) \text{ GB} \Rightarrow 60,000 \text{ GB} \Rightarrow 60 \text{ TB}$$

So we assume a high-end server could save 150 GB of data we would need 400 servers to handle this sort of load. So we need to apply to shard

We can use either of two technique to shard our data :

1. **Shard based on words:** We first calculate the hash based on word and then find the server based on the hash.

Problems with this approach :

1. There might be too much load on the single server when a search term becomes hot.
2. No uniform distribution of load

2. **Shard based on tweet id:** We calculate the hash based on tweet id and find the server based on this hash. All the servers maintain the index for all the words . while fetching results, the result is first fetched from each server then compiled together on a central server and returned to the user.

Caching Strategy

We can use Redis to store all the hot tweets in the cache instead of serving it from the DB. We will use LRU (Least Recently used) as our eviction strategy for removing data from the data from our systems.

Ranking Tweets

Generally, a ranking score ie, RS, is precomputed for all the tweets and stored along with the tweet id. RS may be computed based on the number of likes, number of comments, etc. When fetching the result, each server sorts the tweets based on RS, and then the final central server combines these sorted results and sorts them based on RS and serves them to the user for further processing.

Solution 3:

Twitter users can update their status whenever they want to update irrespective of time. Each status or we can say it as **Tweets** consists of a plain string or text, and I intend to design a system that allows searching over all the user tweets. In this blog, I have given importance to the **Tweet Search Functionality**.

The predominant factors where I have given more emphasis throughout the blogs are given below:

- Functional and Nonfunctional Requirements for the Twitter Search System
- Estimation of initial Capacity:-considering all Constraints
- Rest APIs needed for the Applications
- High-Level Design architecture
- Detailed system explanations with Components Design Diagram
- High Scalability & Fault Tolerance
- Caching using Redis Cluster
- Load Balancers for Load balancing
- Feedback/Rating Service

Requirements for the Twitter Search System

Let's consider the functional and non-functional requirements: As you know the fact here that Twitter has more than 1 billion total users with **600 million daily active users**. So, if we calculate the average then Twitter gets every day more than 200 million tweets approximately. Then if you calculate the average size of a tweet, it will come to around 300 bytes. Then, in this case, we have to assume there will be **500M** searches every day. The search queries generally consist of multiple words combined with AND/OR. So, our goal here is to come up with a system that can efficiently store and query tweets.

Estimation of Initial Capacity:

Storage Capacity: Since as per our calculation, we will have every day 200 million new tweets, and then if we calculate the average it will come around 200 bytes. So, let's find out how much storage we will need here:

200M * 200 equals to = approx. 40GB/day

So, the calculation goes like this: - Total storage per second: 40GB / 24hours / 3600sec approximately = 0.4MB/second

Designing REST API

Now we have to implement REST APIs to expose the functionality of our service; the following could be the definition of the search API:

Sample Handler method Signature

```
search(api_key, search_words, max_results, sort, page_token)
```

Parameters

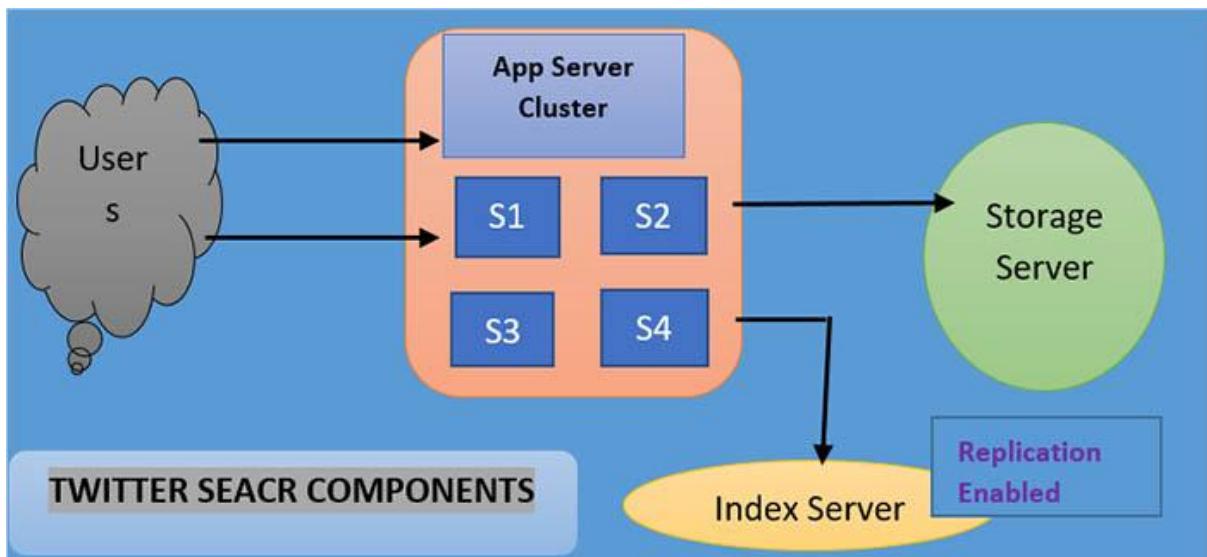
- `api_key` (string): The API key of a registered account. This will be used to, among other things, restrict users based on their allocated permissions/quota.
- `search_words` (string): A string containing the search terms.
- `max_results` (number): Number of tweets to return.
- `sort` (number): Optional sort mode: Latest first (0 - default), Best matched (1), Most liked (2).
- `page_token` (string): This token will specify a page in the result set that should be returned.

Response Body: (JSON)

The response body returned will be a JSON which will contain information about the below fields:

- Several tweets that match with the search query and return type will be a List here.
- Each tweet that, is returned will have the `user_ID` & `User_Name` then we will have `Tweet_ID` & `Tweet_Text` field.
- Also, the JSON will contain the creation timestamp & number of likes as well.

High-Level Design Architecture



At the high level, I have explained here how we can design our backend system to store all the tweets in our SQL / NoSQL database and the next step is creating or building an index. The significance of this index is to keep track of those words which appear in specific tweets. In this way, with the help of our index, we can quickly find the search tweets when a user tries to search for a specific tweet.

Storage: We have to consider how much storage space is needed here to store the new tweets of size 400GB every day. As you can, guess this is going to be a huge amount of data, so, we definitely have to think of some advanced data storage mechanism which is data partitioning and database sharding scheme. These mechanisms will share/distribute the data efficiently to clusters of multiple servers. Let's consider for the next five years Plan, then we are going need the following storage:

So, the calculation goes like this $40\text{GB} * 365\text{days} * 5\text{years}$ approximately = 70TB

If you do not want to be saturated or more than **80-85%** full at any time, we approximately will need 80TB of total storage. Let's assume that we want to keep an extra copy of all tweets for fault tolerance; then, our total storage requirement will be **200 TB**. If we assume a modern server can store up to **1TB of data**, we will need **70-80** such servers to hold all of the required data for the next five years.

Let's start with a simplistic design where we store the tweets in a MySQL/Oracle database. You can even opt for NoSQL Db like Mongo or Cassandra. Here, better we can assume that we store the tweets in a table having two columns, Tweet ID and Tweet message. Let's assume we partition our data based on Tweet ID. If our Tweet IDs are particular system-wide, we will outline a hash characteristic that can map a Tweet ID to a storage server wherein, we will save that tweet object.

How can we build here the unique Tweet IDs throughout our backend system? If we consider that we may get every day **400M** new tweets, then let's calculate how many tweet objects will be created in five years?

So, the calculation goes like this:- $200M * 365 \text{ days} * 5 \text{ years}$ equals approx. 370 billion

It means we will require around the number which is of 5 bytes size here to identify the Tweet IDs uniformly or uniquely. We can consider here one thing that we have a service, that will be generating a unique Tweet ID when we are going to store an object (To note here the Tweet ID I have mentioned here is similar to Tweet ID what I already mentioned in Designing Twitter section). Then you can feed this Tweet ID to the hash function to find the storage server and store our tweet object there.

Index: What should our index suppose to be designed or formed? Since our tweet queries will consist of words, hence we have to build an index that can tell us which word comes in which tweet object. In this case, we have to estimate first our index will look like how big or how small.

If we want to build an index for all the English words and some famous nouns like people names, address or city names, etc., and if we assume that we have around **200K-250K** English words and around 150K nouns, then we will have **350-400k** total words in our index. So, considering the above fact, we have to assume that the average length of a word is 4-5 characters. If we are keeping our index in memory, we need **1-1.5MB** approximately of memory to store all the words:

400K (Taking a maximum of) * (Taking a maximum of) 5 = approx. 2 MB

One more thing here we have to consider is that we want to keep the index in memory for all the tweets from only the past **1 year**. You can increase to 2 Years, but I have considered the basic level here. Since we will be getting approximately **370 B tweets in 5 years**, it will give us **74 B** tweets in one year. In our case, as we have considered that each Tweet ID will be 5 bytes, let's calculate how much memory will we need to store all the Tweet IDs?

In our case, it will be $74 B * 5 = \text{approximately } 370 \text{ GB}$

So, our index what I mentioned here should be like a distributed hash table, where 'value' can be a list of Tweet IDs which will have from all those tweets which contain that word and 'key' will be the search term or search word.

Assuming on average we have **20-30 words** in each tweet messages and since we will not be considering for indexing the prepositions and other small words like 'the', 'an', 'and' etc.: these we will be excluded from indexing, So, assume here that we will have approximately 10 words which need to be indexed in each tweet message. It means each Tweet ID will be stored **10 times** in our index. So, let's calculate how much total memory we may need for storing our index:

$(370 * 10) + 0.4 \text{ MB} \text{ approximately equals to } = 1.5 \text{ TB}$

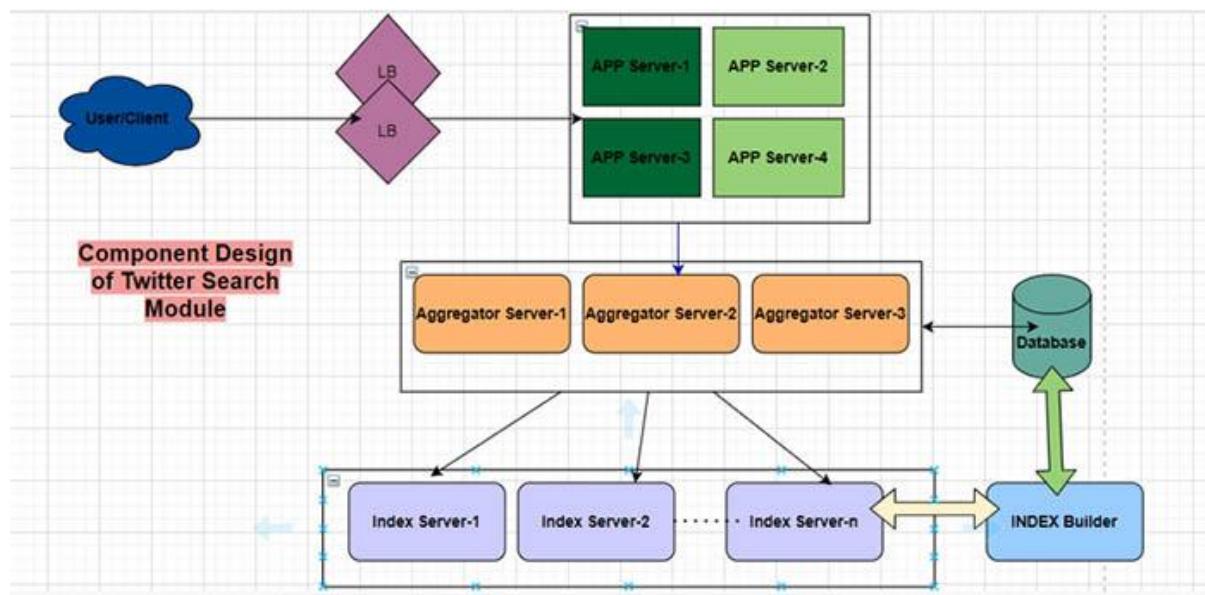
If we assume that a high-end server has **130GB of memory**, we would need **15 such servers** to hold our index, and then we can partition our data based on two criteria.

Word-Based Sharing: When creating our index, we will repeat through all the words in the tweet and calculate the hash of each word to find the server where it will be indexed. To get

all the tweets that contain a specific word, then in this case just query to that specific server that contains that word or term.

We can have side effects with this approach where we may get few issues which I have explained below:

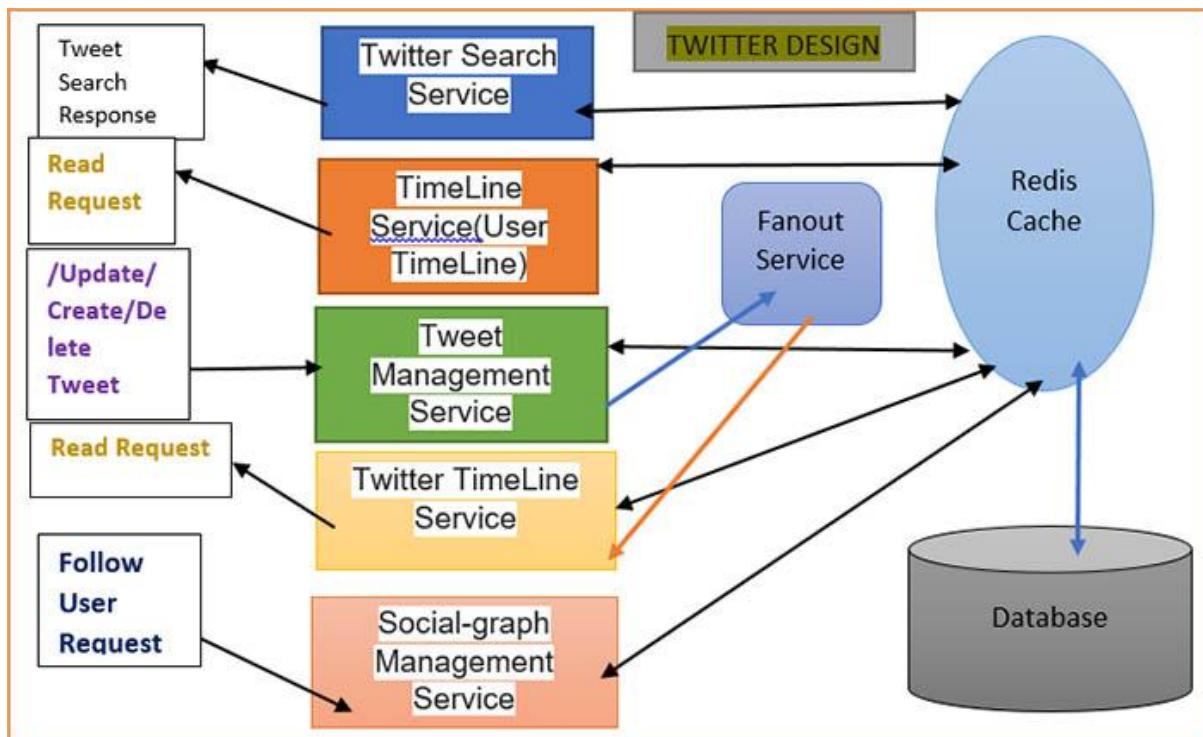
1. In case the Tweet search word or term becomes popular or we say it is a technical term as "HOT", Then in this case there will be a lot of queries that need to be triggered on the server which holds that word. As this will cause a very high load on the server then it will affect the performance of our Search service.
2. After some days or years, there may be few words that will end up storing a lot of Tweet IDs compared to others, therefore, to maintain a proper or uniform distribution of words while tweets are growing is a little bit tricky. To get rid of these situations, we have to go for Consistent Hashing, or another alternative option is to repartition our data.
3. Sharding is based on the tweet object: While storing, we will pass the Tweet ID to our hash function to find the server and index all the tweet words on that particular server or server. So, when you will be querying for a particular word, we have to query all the servers. In this case, each one of the servers will return a set of Tweet IDs. Then finally a centralized server will aggregate these results to return them to the user.



System Designing for Twitter Search Service

Detailed Component Design

Here, we directly apply [several microservices](#) like the search service to its service user timeline, home timeline, and social graph, fan out and of course, there is a data store and caching layer as well. You can find here we actually go deep down into designing the Twitter search service and also you can get to know how the fan-out service works and how the home timeline service works as well.



Fan-out service is possible for forwarding that to it from the tweet service to search service and the user home timeline service. In case there are other components or microservices in the Twitter service architecture for example notification or trending service, it is the fellow service that is possible for forwarding the tweets to those services as well.

The Fan-out service comparison of multiple distributed queues and whenever a tweet service receives a tweet from a user it passes the tweet to the fan-out service by calling an API exposed by the fan-out service which is called **push-tweet**. The push-tweet API inserts the tweet in one of the distributed queues. The distributed queues at the first stage are assigned by the hash of the user ID of the user who posted the tweet. The function push() returns immediately after posting the tweet to one of the distributed queues.

How does the Twitter Search Service module work?

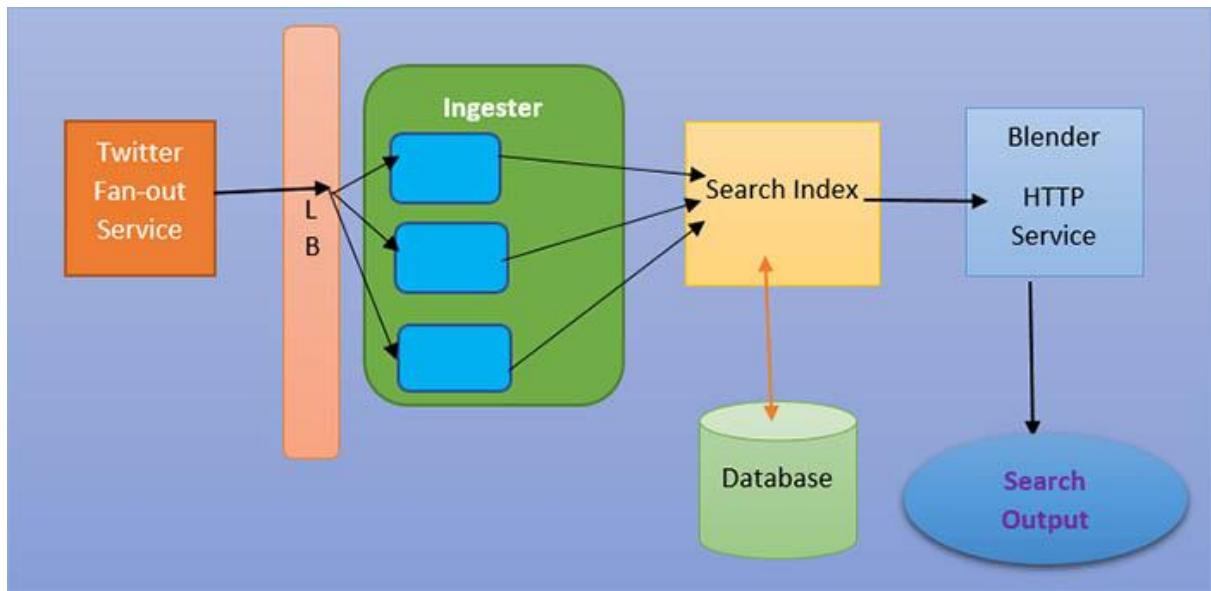
The search service is formed on basis of the following logical components. Wo we have **Ingestor**, **Search index**, and **Blender**, and of course we have data store with Cache. The Ingestor or ingestion engine is responsible for tokenizing the Tweet into several terms or keywords. Let me give you an example where a user posts a tweet that "**Bangalore is becoming a hub of IT companies**". Eventually, the fan-out service will forward this tweet to the **Search service** and it will call an API to forward this to it to the Ingestor.

The Ingestor will first organize this into following different terms with each word separately. It will then filter out those terms or words which are not useful to be used for search. These terms are called **Stop Words**. So, for example, it will remove, or discard "is" and "of" words from the whole sentence because they are not useful for searching.

The Ingestor can get the listing of these phrases this is the prevent phrases both shape a few configurations or a few databases that it has the following step is to carry out a

technique known as Stemming at the closing phrases to discover the foundation phrase. Stemming is the technique of decreasing inflected or every so often known as derived phrases to the phrase stem or root phrase. For example, the root word for Bangalore is Bangalore and for IT Hub it is IT Hub. But here companies actually can become a company.

The Ingester component then passes the tweet along with the root terms to the search index component. It is to be noted that Ingester would comply with at least three application servers and we can of course always increase these application servers as the load on the service increases.



The search index module will generally create an inverted index. An inverted index is nothing but an "Index Datastructure" which stores or saves a mapping among the contents together with phrases to its area within the report or set of files that are in our case might be ready of tweets. For the search index, the search-index for micro provider or thing shops the phrases together with the **Tweet ID** within the datastore.

Now there is more than one process or we can say multiple approaches to shard the data store. The first and easy approach to shard the data store is by "**Sharding by words**." In this approach, we shard the database by the hash of the word, which means different words will be in different partitions, and then a single partition will hold all the tweets that contain that word.

In this case to find the tweets containing a specific word we just need to go to a single partition. So, in that case, the search is very fast however the call from the Ingester to a search index to generate this inverted index will be slower, because it's possible now that a tweet may contain 9, 10, 12, 13, or even more terms.

So, we would be going to different partitions to write the word – to- tweet ID mapping in all those partitions for all the different terms or words which are Tweet contains.

In our example of the tweet, we are saying that "**Bangalore is becoming a hub of IT companies**, we have four different terms or words. When Ingester will send the **Write**

operation to the search index we will go to four different partitions for the word “**Bangalore**”, “**hub**”, IT, and company to store the Tweet ID mapping to all those words. There are some other issues with this approach for example if a word becomes hot then there will be lots of queries that come into this partition storing that word.

To mitigate that we can add **Caching mechanism** in front of the data store, and we can also add a large number of replicas for that particular partition in the distributed cache. The second approach is shared by Tweet ID. In this case, the write operation will go to a single partition and all the mapping between the words with Tweet ID will be stored there. With this Sharding approach, the write operation is very fast because we are just going to a single partition.

Search requires sending the request to all the partitions to read the list of tweets that contain a specific word. Then an aggregator server combines all those results from those different partitions before sending the final result to the user. The issue with this approach is that every single search query will hit all the partitions and thus will put extra pressure on the data store.

The Third sharding approach is to level shard by words and that Tweet ID. In this approach, we first perform the first layer sharding by the hash of the word to determine a group of partitions, and then we used the second layer sharding by Twitter ID to determine the person in that group where the word- to- Tweet ID mapping should be stored.

In this case to search tweets for a word we only need to go to only a small set of partitions. The search index will also have the same design that it will have at least three application servers and also it will have a distributed cache and a data store with it.

Fault Tolerance

1. What will happen when an index server is out of Order? We can have replication here where a secondary replica of each server and if the primary server dies it can take control after the failover. So here both servers primary and secondary in this case will be holding the same copy of the index.
2. Let's say both primary and secondary servers go into shutdown state at the same time, then in that case, what you are going to do? In this case, we may allocate a new web server and rebuild the same index on it. But the question comes how you are going to do it? In this case, you are not aware of what terms or words, or tweets were kept on this server.
3. If we want to use a brute-force mechanism solution approach which is :-> '**Sharding based on the tweet object**', then it works in such a way that it iterates through the whole database to filter the Tweet IDs. Then, with the help of the hash function, it finds out all the tweets that are required and which need to be stored on this server. But just think yourself how it is really inefficient because when the new server builds up or starts up I mean during that gap of the rebuild of the new server, we will not be able to serve any query from it, hence we are going to missing some tweets that should have been seen by the user.
4. So, the primary consideration is here how efficiently we can fetch a mapping between the index server and the tweet to obtain consistency. In this case, we have to go for the approach, where we need to build a reverse index that will map all the Tweet IDs

to their index server. Our Index-Builder server will be holding this information. And we have to just build a Hash table where the ‘key’ will be the index server number and the ‘value’ will be a HashSet containing all the Tweet IDs which were already kept at that index server.

5. One thing observes here is that we are storing/maintaining all the Tweet IDs in a HashSet; what it does is that this enables us to add/remove tweets quickly from our index server. So now, during the switch over time when an index server has to rebuild itself, it can simply query the Index-Builder server so that it will fetch all that tweets for which it can again build the index once after it stores those tweets. This approach is the best approach and surely be fast. Also, we can maintain a replica of the Index-Builder server for achieving fault tolerance.

Caching

To deal with hot tweets, the [Java spring boot development](#) team can introduce Caching Mechanism in front of our database. They can use **Memcached or Redis**, which can store all such hot tweets in memory as key-value pairs in the Redis cache. We can set up the Redis cluster for High availability. Application servers, before hitting the backend database, can check if the cache has that tweet. So that the loads on the database will be less. Based on the User’s usage patterns or during heavy traffic or low traffic, you can adjust the number of cache servers you need.

Scaling & Load Balancing

In our design architecture, we can implement a load balancing layer at two places in our System environment:

1. **Clients --LB-- Application Servers**
2. **Application Servers --LB-- Backend Server.**

Initially, a very basic Round Robin Load balancing approach can be leveraged for our requirement model that uniformly routes the incoming requests to the underlying backend Apache Web servers. This Load Balancer (LB) ideally does not introduce any performance issue or overhead in case of heavy Load. Another benefit of this Load balancing approach is that it discards the servers which are already dead, and it manages the traffic through other active servers and does not send any request to dead servers by taking out of the rotation.

But there are also few drawbacks of this Round Robin Load balancer. It does not consider the server load in case if it is heavily loaded or less occupied. What I mean to say here is that even though the server is heavily loaded, the LB continues sending the traffic to that server. To overcome the above drawback, another powerful intelligent Load balancing technique can be used in our case. What this LB does the extra thing? It manages the load efficiently by keeping track of the load of each server by querying the backend servers periodically for knowing their health status as well as the load percentage and then accordingly adjust traffic.

Feedback & Rating

How can we achieve ranking and Feedback functionality where Search Results have to be ranked on basis of few factors such as social graph distance/popularity or relevance etc. that? Let's assume on basis of Popularity we want to rank the tweets by the number of likes or comments on a specific tweet is getting, etc. In such a case, our Ranking or Rating algorithm can programmatically derive a 'number' based upon the likes' count, we can call it a Popularity number. Then we have to store and map it with the index.

Before returning the results to the aggregator server, what needs to be done here:- Each partition can filter the results and then sort primarily based totally on this number of likes. The next step is that the aggregator server collects all these results, adds up this result, and then applies a sorting algorithm based on that specific number which we derived earlier (i.e. the Popularity Number), and finally gives the response as the top results to the end-user.

Solution 4:

Design the Facebook feed and **Design Facebook search** are similar questions.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions. Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **User** posts a tweet
 - **Service** pushes tweets to followers, sending push notifications and emails
- **User** views the user timeline (activity from the user)
- **User** views the home timeline (activity from people the user is following)
- **User** searches keywords
- **Service** has high availability

Out of scope

- **Service** pushes tweets to the Twitter Firehose and other streams
- **Service** strips out tweets based on users' visibility settings
 - Hide @reply if the user is not also following the person being replied to
 - Respect 'hide retweets' setting
- Analytics

Constraints and assumptions

State assumptions

General

- Traffic is not evenly distributed
- Posting a tweet should be fast

- Fanning out a tweet to all of your followers should be fast, unless you have millions of followers
- 100 million active users
- 500 million tweets per day or 15 billion tweets per month
 - Each tweet averages a fanout of 10 deliveries
 - 5 billion total tweets delivered on fanout per day
 - 150 billion tweets delivered on fanout per month
- 250 billion read requests per month
- 10 billion searches per month

Timeline

- Viewing the timeline should be fast
- Twitter is more read heavy than write heavy
 - Optimize for fast reads of tweets
- Ingesting tweets is write heavy

Search

- Searching should be fast
- Search is read-heavy

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

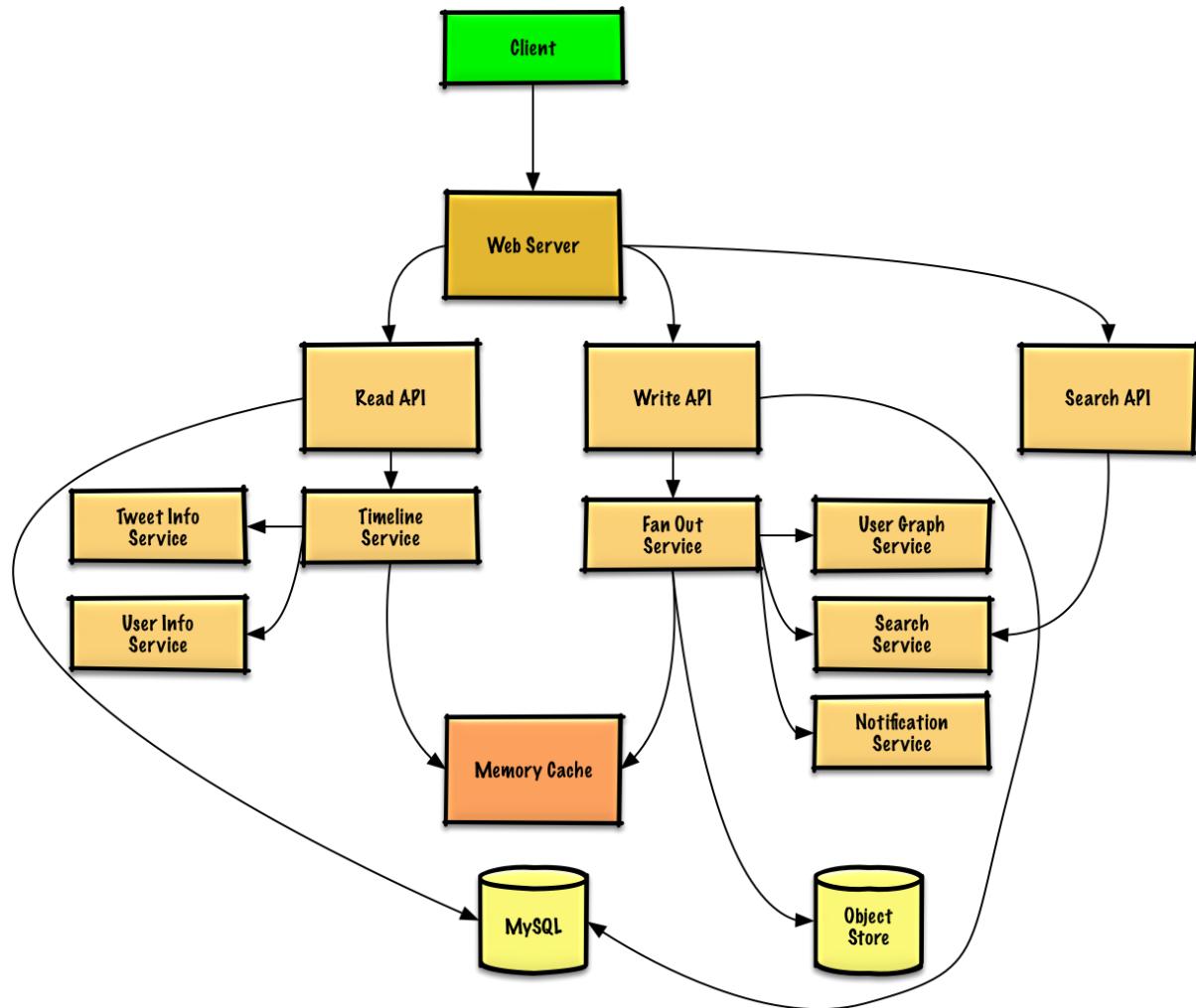
- Size per tweet:
 - tweet_id - 8 bytes
 - user_id - 32 bytes
 - text - 140 bytes
 - media - 10 KB average
 - Total: ~10 KB
- 150 TB of new tweet content per month
 - 10 KB per tweet * 500 million tweets per day * 30 days per month
 - 5.4 PB of new tweet content in 3 years
- 100 thousand read requests per second
 - 250 billion read requests per month * (400 requests per second / 1 billion requests per month)
- 6,000 tweets per second
 - 15 billion tweets per month * (400 requests per second / 1 billion requests per month)
- 60 thousand tweets delivered on fanout per second
 - 150 billion tweets delivered on fanout per month * (400 requests per second / 1 billion requests per month)
- 4,000 search requests per second
 - 10 billion searches per month * (400 requests per second / 1 billion requests per month)

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Step 3: Design core components

Dive into details for each core component.

Use case: User posts a tweet

We could store the user's own tweets to populate the user timeline (activity from the user) in a [relational database](#). We should discuss the [use cases and tradeoffs between choosing SQL or NoSQL](#).

Delivering tweets and building the home timeline (activity from people the user is following) is trickier. Fanning out tweets to all followers (60 thousand tweets delivered on fanout per second) will overload a traditional [relational database](#). We'll probably want to choose a data store with fast writes such as a **NoSQL database** or **Memory Cache**. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

We could store media such as photos or videos on an **Object Store**.

- The **Client** posts a tweet to the **Web Server**, running as a [reverse proxy](#)
- The **Web Server** forwards the request to the **Write API** server
- The **Write API** stores the tweet in the user's timeline on a **SQL database**
- The **Write API** contacts the **Fan Out Service**, which does the following:
 - Queries the **User Graph Service** to find the user's followers stored in the **Memory Cache**
 - Stores the tweet in the *home timeline of the user's followers* in a **Memory Cache**
 - O(n) operation: 1,000 followers = 1,000 lookups and inserts
 - Stores the tweet in the **Search Index Service** to enable fast searching
 - Stores media in the **Object Store**
 - Uses the **Notification Service** to send out push notifications to followers:
 - Uses a **Queue** (not pictured) to asynchronously send out notifications

Clarify with your interviewer how much code you are expected to write.

If our **Memory Cache** is Redis, we could use a native Redis list with the following structure:

tweet n+2	tweet n+1	tweet n
8 bytes	8 bytes	8 bytes
tweet_id	tweet_id	tweet_id
user_id	meta	user_id
meta		meta

The new tweet would be placed in the **Memory Cache**, which populates the user's home timeline (activity from people the user is following).

We'll use a public [REST API](#):

```
$ curl -X POST --data '{ "user_id": "123", "auth_token": "ABC123", \
  "status": "hello world!", "media_ids": "ABC987" }' \
  https://twitter.com/api/v1/tweet
```

Response:

```
{  
  "created_at": "Wed Sep 05 00:37:15 +0000 2012",  
  "status": "hello world!",  
  "tweet_id": "987",  
  "user_id": "123",
```

```
    ...
}
```

For internal communications, we could use [Remote Procedure Calls](#).

Use case: User views the home timeline

- The **Client** posts a home timeline request to the **Web Server**
- The **Web Server** forwards the request to the **Read API** server
- The **Read API** server contacts the **Timeline Service**, which does the following:
 - Gets the timeline data stored in the **Memory Cache**, containing tweet ids and user ids - $O(1)$
 - Queries the **Tweet Info Service** with a [multiget](#) to obtain additional info about the tweet ids - $O(n)$
 - Queries the **User Info Service** with a multiget to obtain additional info about the user ids - $O(n)$

REST API:

```
$ curl https://twitter.com/api/v1/home_timeline?user_id=123
```

Response:

```
{
  "user_id": "456",
  "tweet_id": "123",
  "status": "foo"
},
{
  "user_id": "789",
  "tweet_id": "456",
  "status": "bar"
},
{
  "user_id": "789",
  "tweet_id": "579",
  "status": "baz"
},
```

Use case: User views the user timeline

- The **Client** posts a user timeline request to the **Web Server**
- The **Web Server** forwards the request to the **Read API** server
- The **Read API** retrieves the user timeline from the **SQL Database**

The REST API would be similar to the home timeline, except all tweets would come from the user as opposed to the people the user is following.

Use case: User searches keywords

- The **Client** sends a search request to the **Web Server**
- The **Web Server** forwards the request to the **Search API** server
- The **Search API** contacts the **Search Service**, which does the following:
 - Parses/tokenizes the input query, determining what needs to be searched
 - Removes markup
 - Breaks up the text into terms
 - Fixes typos
 - Normalizes capitalization
 - Converts the query to use boolean operations
 - Queries the **Search Cluster** (ie [Lucene](#)) for the results:
 - [Scatter gathers](#) each server in the cluster to determine if there are any results for the query
 - Merges, ranks, sorts, and returns the results

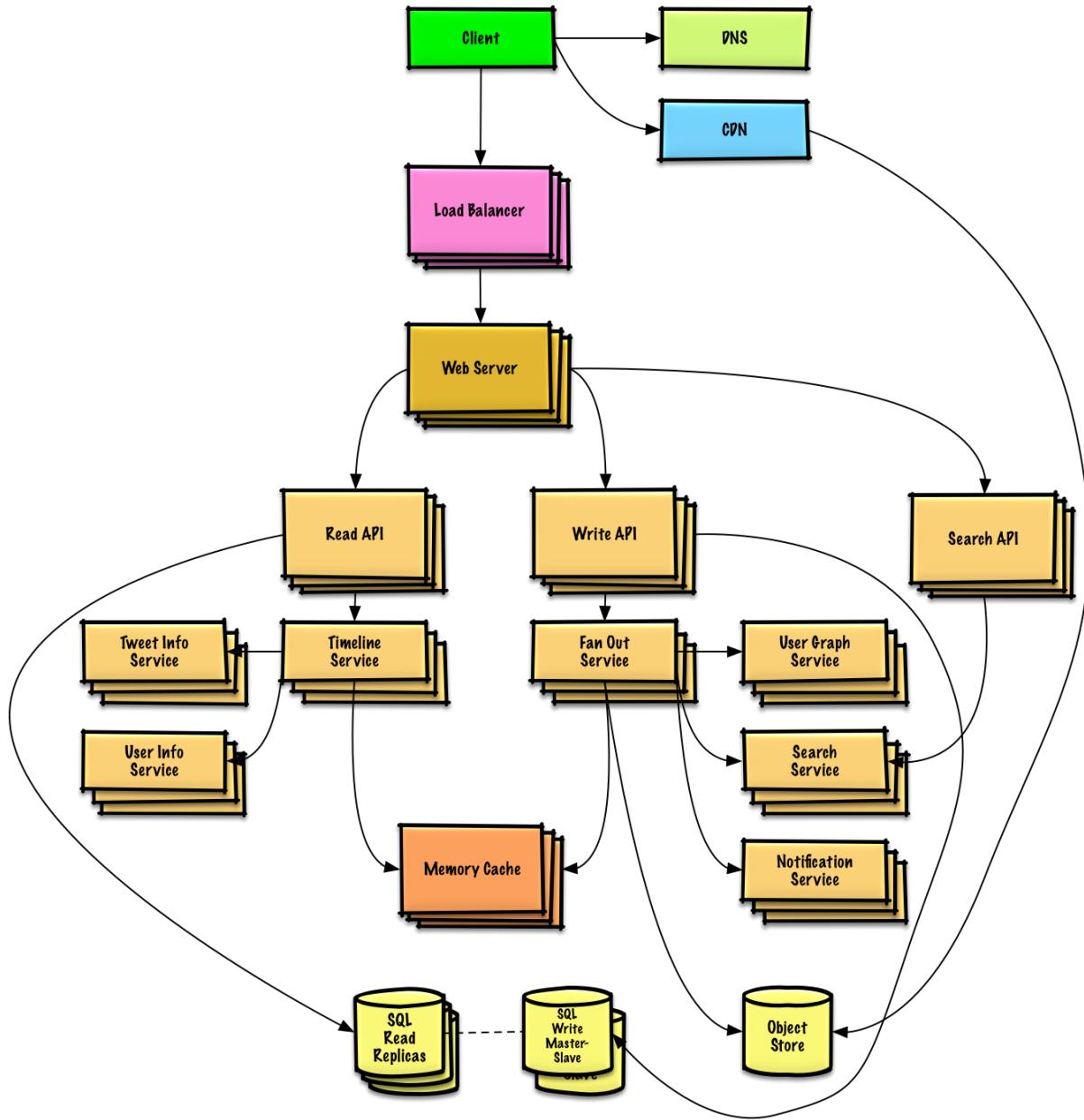
REST API:

```
$ curl https://twitter.com/api/v1/search?query=hello+world
```

The response would be similar to that of the home timeline, except for tweets matching the given query.

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.



Important: Do not simply jump right into the final design from the initial design!

State you would 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers**? **CDN**? **Master-Slave Replicas**? What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following [system design topics](#) for main talking points, tradeoffs, and alternatives:

- [DNS](#)
- [CDN](#)
- [Load balancer](#)
- [Horizontal scaling](#)
- [Web server \(reverse proxy\)](#)
- [API server \(application layer\)](#)
- [Cache](#)
- [Relational database management system \(RDBMS\)](#)
- [SQL write master-slave failover](#)
- [Master-slave replication](#)
- [Consistency patterns](#)
- [Availability patterns](#)

The **Fanout Service** is a potential bottleneck. Twitter users with millions of followers could take several minutes to have their tweets go through the fanout process. This could lead to race conditions with @replies to the tweet, which we could mitigate by re-ordering the tweets at serve time.

We could also avoid fanning out tweets from highly-followed users. Instead, we could search to find tweets for highly-followed users, merge the search results with the user's home timeline results, then re-order the tweets at serve time.

Additional optimizations include:

- Keep only several hundred tweets for each home timeline in the **Memory Cache**
- Keep only active users' home timeline info in the **Memory Cache**
 - If a user was not previously active in the past 30 days, we could rebuild the timeline from the **SQL Database**
 - Query the **User Graph Service** to determine who the user is following
 - Get the tweets from the **SQL Database** and add them to the **Memory Cache**
- Store only a month of tweets in the **Tweet Info Service**
- Store only active users in the **User Info Service**
- The **Search Cluster** would likely need to keep the tweets in memory to keep latency low

We'll also want to address the bottleneck with the **SQL Database**.

Although the **Memory Cache** should reduce the load on the database, it is unlikely the **SQL Read Replicas** alone would be enough to handle the cache misses. We'll probably need to employ additional SQL scaling patterns.

The high volume of writes would overwhelm a single **SQL Write Master-Slave**, also pointing to a need for additional scaling techniques.

- [Federation](#)

- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

We should also consider moving some data to a **NoSQL Database**.

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#)
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

14. Designing Ticketmaster

Solution 1:

Let's design an online E-ticketing system that sells movie tickets. A movie ticket booking system provides its customer the ability to purchase theatre seats online. They allow the customer to browse movies currently being played and to book available seats, anywhere anytime.

1. Requirements and System goals

Functional Requirements

- The service should list different cities where its affiliated cinemas are located.
- When user selects a city, the service should display movies released in that particular city.
- When user selects movie, the service should display the cinemas running the movie plus available show times.
- Users should be able to book a show at a cinema and book tickets.
- The service should be able to show the user the seating arrangement of the cinema hall. The user should be able to select multiple seats according to the preference.
- The user should be able to distinguish between available seats from booked ones.
- Users should be able to put a hold on the seat (for 5 minutes) while they make payments.
- Users should be able to wait if there is a chance that the seats might become available (When holds by other users expire).
- Waiting customers should be serviced in a fair, first come, first serve manner.

Non-Functional Requirements

- The service should be highly concurrent. There will be multiple booking requests for the same seat at any particular point in time.
- The system has financial transactions, meaning it should be secure and the DB should be ACID compliant.
- Assume traffic will spike on popular/much-awaited movie releases and the seats would fill up pretty fast, so the service should be highly scalable and highly available to keep up with the surge in traffic.

Design Considerations

1. Assume that our service doesn't require authentication.
2. No handling of partial ticket orders. Either users get all the tickets they want or they get nothing.
3. Fairness is mandatory.
4. To prevent system abuse, restrict users from booking more than 10 seats at a time.

2. Capacity Estimation

Traffic estimates: 3 billion monthly page views, sells 10 million tickets a month.

Storage estimates: 500 cities, on average each city has 10 cinemas, each with 300 seats, 3 shows daily.

Let's assume each seat booking needs 50 bytes (IDs, NumberOfSeats, ShowID, MovieID, SeatNumbers, SeatStatus, Timestamp, etc) to store in the DB. We need to store information about movies and cinemas; assume another 50 bytes.

So to store all data about all shows of all cinemas of all cities for a day

$$500 \text{ cities} * 10 \text{ cinemas} * 300 \text{ seats} * 3 \text{ shows} * (50 + 50) \text{ bytes} = 450 \text{ MB / day}$$

To store data for 5 years, we'd need around

$$450 \text{ MB/day} * 365 * 5 = 821.25 \text{ GB}$$

3. System APIs

Let's use REST APIs to expose the functionality of our service.

Searching movies

```
search_movies(
    api_dev_key: str,      # The API developer key. This will be used to throttle users
                           # based on their allocated quota.
    keyword: str,          # Keyword to search on.
    city: str,             # City to filter movies by.
    lat_long: str,         # Latitude and longitude to filter by.
    radius: int,           # Radius of the area in which we want to search for events.
    start_date: datetime,  # Filter with a starting datetime.
    end_date: datetime,    # Filter with an ending datetime.
```

```

postal_code: int,      # Filter movies by postal code / zipcode.
include_spell_check, # (Enum: yes or no)
result_per_page: int # number of results to return per page. Max = 30.
sorting_order: str   # Sorting order of the search result.
                      # Allowable values: 'name,asc', 'name,desc', 'date,asc',
                      # 'date, desc', 'distance,asc', 'name,date,asc', 'name,date,desc'
)

```

Returns: (JSON)

```
[
{
  "MovieID": 1,
  "ShowID": 1,
  "Title": "Klaus",
  "Description": "Christmas animation about the origin of Santa Claus",
  "Duration": 97,
  "Genre": "Animation/Comedy",
  "Language": "English",
  "ReleaseDate": "8th Nov. 2019",
  "Country": "USA",
  "StartTime": "14:00",
  "EndTime": "16:00",
  "Seats":
  [
    {
      "Type": "Regular",
      "Price": 14.99,
      "Status": "Almost Full"
    },
    {
      "Type": "Premium",
      "Price": 24.99,
      "Status": "Available"
    }
  ],
  "MovieID": 2,
  "ShowID": 2,
  "Title": "The Two Popes",
  "Description": "Biographical drama film",
  "Duration": 125,
  "Genre": "Drama/Comedy",
  "Language": "English",
  "ReleaseDate": "31st Aug. 2019",
  "Country": "USA",
  "StartTime": "19:00",
}
]
```

```

    "EndTime": "21:10",
    "Seats":
    [
        {
            "Type": "Regular",
            "Price": 14.99,
            "Status": "Full"
        },
        {
            "Type": "Premium",
            "Price": 24.99,
            "Status": "Almost Full"
        }
    ]
},
]

```

Reserving Seats

```

reserve_seats(
    api_dev_key: str, # API developer key.
    session_id: str, # User Session ID to track this reservation.
        # Once the reservation time of 5 minutes expires,
        # user's reservation on the server will be removed using this ID.
    movie_id: str, # Movie to reserve.
    show_id: str, # Show to reserve.
    seats_to_reserve: List(int) # An array containing seat IDs to reserve.
)

```

Returns: (JSON)

The status of the reservation, which would be one of the following:

1. Reservation Successful,
2. Reservation Failed - Show Full
3. Reservation Failed - Retry, as other users are holding reserved seats.

4. DB Design

1. Each **City** can have multiple **Cinemas**
2. Each **Cinema** can have multiple **Cinema_Halls**.
3. Each **Movie** will have **Shows** and each Show will have multiple **Bookings**.
4. A **User** can have multiple **Bookings**.

5. High Level Design

From a bird's eye view,

- Web servers handle user's sessions,
- Application servers handle all the ticket management and
- stored in the DB

- as well as work with cache servers to process reservations.

6. Detailed Component Design

Let's explore the workflow part where there are no seats available to reserve, but all seats haven't been booked yet, (some users are holding in the reservation pool and have not booked yet)

- the user is taken to a waiting page, waiting until the required seats get freed from the reservation pool. Options for the user at this point include:
- if the required number of seats become available, take the user to theatre page to choose seats
- While waiting, if all seats are booked, or there are fewer seats in the reservation pool than the user intends to book, then the user is shown the error message.
- User cancels the waiting and is taken back to the movie search page.
- At maximum, a user waits for an hour, after that the user's session expires and the user is taken back to the movie search page.

If seats are reserved successfully, the user has 5 minutes to pay for the reservation. After payment, booking is marked complete. If the user isn't able to pay within 5 minutes, all the reserved seats are freed from the reservation pool to become available to other users.

How we'll keep track of all active reservations that have not been booked yet, and keep track of waiting customers

We need two daemon services for this:

a. Active Reservation Service

This will keep track of all active reservations and remove expired ones from the system.

We can keep all the reservations of a show in memory in a [Linked Hashmap](#), in addition to also keeping data in the DB.

- We will need this doubly-linked data structure to jump to any reservation position to remove it when the booking is complete.
- The head of the HashMap will always points to the oldest record, since we will have expiry time associated with each reservation. The reservation can be expired when the timeout is reached.

To store every reservation for every show, we can have a HashTable where the key = ShowID and value = Linked HashMap containing BookingID and creation Timestamp.

In the DB,

- We store reservation in the Booking table.
- Expiry time will be in the Timestamp column.

- The Status field will have a value of Reserved(1) and, as soon as a booking is complete, update the status to Booked(2).
- After status is changed, remove the reservation record from Linked HashMap of the relevant show.
- When reservation expires, remove it from the Booking table or mark it Expired(3), and remove it from memory as well.

ActiveReservationService will work with the external Financial service to process user payments. When a booking is completed, or a reservation expires, WaitingUserService will get a signal so that any waiting customer can be served.

```
# The HashTable keeping track of all active reservations
hash_table = {
    # ShowID : # LinkedHashMap <BookingID, Timestamp>
    'showID1': {
        (1, 1575465935),
        (2, 1575465940),
        (2, 1575466950),
    },
    'showID2': { ... },
}
```

b. Waiting User Service

- This daemon service will keep track of waiting users in a Linked HashMap or TreeMap.
- To help us jump to any user in the list and remove them when they cancel the request.
- Since it's a first-come-first-served basis, the head of the Linked HashMap would always point to the longest waiting user, so that whenever seats become available, we can serve users in a fair manner.

We'll have a HashTable to store all waiting users for every show. Key = ShowID, value = Linked HashMap containing UserIDs and their start-time

Clients can use Long Polling to keep themselves updated for their reservation status. Whenever seats become available, the server can use this request to notify the user.

Reservation Expiration

On the server, the Active Reservation Service keeps track of expiry of active connections (based on reservation time).

On the client, we will show a timer (for expiration time), which could be a little out of sync with the server. We can add a buffer of 5 seconds on the server to prevent the client from ever timing out after the server, which, if left unchecked, could prevent successful purchase.

7. Concurrency

We need to handle concurrency, such that no 2 users are able to book the same seat.

We can use transactions in SQL databases, isolating each transaction by locking the rows before we can update them. If we read rows, we'll get a write lock on the them so that they can't be updated by anyone else.

Once the DB transaction is committed and successful, we can start tracking the reservation in the Active Reservation Service.

8. Fault Tolerance

If the two services crash, we can read all active reservations from the Booking table. Another option is to have a **master-slave configuration** so that, when the master crashes, the slave can take over. We are not storing the waiting users in the DB, so when Waiting User Service crashes, we don't have any means to recover the data unless we have a master-slave setup. We can also have the same master-slave setup for DBs to make them fault tolerant.

9. Data Partitioning

Partitioning by MovieID will result in all Shows of a Movie being in a single server. For a hot movie, this could cause a lot of load on that server. A better approach would be to partition based on ShowID; this way, the load gets distributed among different servers.

We can use Consistent Hashing to allocate application servers for both (ActiveReservationService and WaitingUserService) based on ShowID. This way, all waiting users of a particular show will be handled by a certain set of servers.

When a reservation expires, the server holding that reservation will:

1. Update DB to remove the expired Booking (or mark it Expired(3)) and update the seats status in Show_Seats table.
2. Remove the reservation from Linked HashMap.
3. Notify the user that their reservation expired.
4. Broadcast a message to WaitingUserService servers that are holding waiting users of that Show to find who the longest waiting user is. Consistent Hashing scheme will help tell what servers are holding these users.
5. Send a message to the WaitingUserService to go ahead and process the longest waiting user if required seats have become available.

When a reservation is successful:

1. The server holding that reservation will send a message to all servers holding waiting users of that Show.
2. These servers upon receiving the above message, will query the DB (or a DB cache) to find how many seats are available.
3. The servers can now expire all waiting users that want to reserve more seats than the available seats.

4. For this, the WaitingUserService has to iterate through the Linked HashMap of all the waiting users to remove them.

Solution 2:

The exact requirements of our system is that it should support multiple cities, and when a user selects a city they should see the available venues in that location. When a user selects a venue they can see the performances and dates for that venue. They can select a performance, date and a number of tickets to add to their cart.

The user can then use a third party supplier to pay for their tickets, after which they will receive an email and phone notification containing their receipt.

Our usual non-functional requirements stand. It must be reliable, scalable and available.

The Approach

We have a standard approach to system design which is explained more thoroughly in the article [here](#). However the steps are summarised below:

1. **Requirements clarification:** Making sure we have all the information before starting. This may include how many requests or users we are expecting.
2. **Back of the envelope estimation:** Doing some quick calculations to gauge the necessary system performance. For example, how much storage or bandwidth do we need?
3. **System interface design:** What will our system look like from the outside, how will people interact with it? Generally this is the API contract.
4. **Data model design:** What our data will look like when we store it. At this point we could be thinking about relational vs non-relational models.
5. **Logical design:** Fitting it together in a rough system! At this point I'm thinking at a level of 'how would I explain my idea to someone who knows nothing about tech?'
6. **Physical design:** Now we start worrying about servers, programming languages and the implementation details. We can superimpose these on top of the logical design.
7. **Identify and resolve bottlenecks:** At this stage we will have a working system! We now refine the design.

With that said, let's get stuck in!

Requirements Clarification

The questions I would be asking include how many cities are we expecting, how many venues per city, how many performances per venue per year, and the capacity of each venue!

Back of the envelope estimation

Let's say we have 100 cities, each with 10 venues, each holding 10 performances a day, with an average of 1000 tickets per venue. Let's also assume everything sells out (hopefully). This gives us:

$100 \text{ cities} * 10 \text{ venues} * 10 \text{ performances} * 1000 \text{ tickets} = 10,000,000 \text{ tickets transactions a day!}$

This corresponds to around 15 sales per second (spread evenly).

If we estimate a city/ venue/ performance/ ticket storage to take up roughly 50B per row, then our total (approximate) storage requirements are:

1. $100 * 50 = 5000\text{B} = 5\text{KB}$ for cities
2. $100 * 10 * 50 = 50,000\text{B} = 50\text{KB}$ for venues
3. $100 * 10 * 50 * 10 = 500,000\text{B} = 500\text{KB}$ for performances
4. $10,000,000 * 50 = 5,000,000,000\text{B} = 500\text{MB}$ for tickets

So the total becomes:

$$500\text{MB} + 500\text{KB} + 50\text{KB} + 5\text{KB} = 500.555\text{MB} \text{ (for the first day)}$$

If we save the ticket data each day then we will increase by 500MB a day, leaving us needing 182.5GB of storage by the end of the year.

To estimate traffic we would need the approximate number of times a person would access pages/ objects across the site. It's possible to do, but probably not worth covering for the exercise.

System interface design

Now we have the rough estimates of how much storage we would like to use, let's think about data access. Initially, we will need to access all the cities, venues, performances and tickets to select from.

We want to be slightly pragmatic about how we load data. We have a series of nested objects: cities contain venues contain performances contain tickets. We could potentially have a single /cities endpoint which returns all of the nested data. However, this seems like overkill — we will get all the tickets every time!

Instead we have a single endpoint per data object.

- **Cities:** /cities
- **Venues:** /cities/{id}/venues
- **Performances:** /cities/{id}/venues/{id}/performances
- **Tickets:** /cities/{id}/venues/{id}/performances/{id}/tickets

We receive a list of data objects per request, with a 200 response and the regular 4XX, 5XX codes.

The other thing we need to do is to be able to reserve and purchase tickets. Let's say we have a ticket object similar to the below.

```
{  
  "id": "<Id of the ticket object>","
```

```
"state": "<AVAILABLE/ RESERVED/ PURCHASED>"  
...  
}
```

To reserve a ticket we could send a POST request to /user/{id}/tickets. The ticket object would then be linked to the user, and the state changed to reserved. To purchase a ticket a PUT (or PATCH) request would be made to the same endpoint, updating the state to purchased.

This is extra useful as the ticket would remain linked to the performance, but the state would have changed, showing it as purchased or reserved.

Another approach we could try using is [GraphQL](#). Using traditional REST principles we can sometimes get embroiled in lots of different endpoints, multiple calls, and the unnecessary loading of data. GraphQL is a flexible query language for our APIs.

We initially define a schema for our cities:

```
type City {  
  id: ID!  
  name: String!  
  venues: [Venue!]!  
}
```

Each city has an Id (which is non-nullable, hence the exclamation mark), a name, and a list of venues. We can then query this using something of the format.

```
query  
{  
  city(id: 1) {  
    venues {  
      name  
    }  
  }  
}
```

Note, this is more of an aside. We'll stick with REST for the time being!

Data model design

Now we need to design our data model. From the description so far we can see a relational pattern emerging. The other thing we need is **transactions** with **ACID properties** to handle the purchasing of tickets.

Let's dive into the definition of a transaction. A transaction is a unit of work for a database. For example, purchasing a ticket may be a transaction. There are two core focuses:

1. To provide units of work that allow for consistent state and recovery in the event of failures midway through.
2. To provide isolation between programs accessing a database concurrently.

We can put these in the context of a ticket purchase. Let's say a ticket costs £10, and the user has £10 in their bank account. In the context of a purchase we would like to:

1. Take £10 from the user.
2. Mark the ticket as purchased.
3. Assign the ticket to the user.

If our database falls over (as they are wont to do), we don't want to get into a state where the user has had £10 deducted, but the ticket is still available and not assigned (point 1)!

Equally, if we have two users purchasing the same ticket, we don't want to deduct £10 from each of them, mark the ticket as purchased, then only be able to assign it to one of them (point 2)!

In Spring we use the [@Transactional annotation](#) to declare transactions. In the database layer itself we use a DBMS dependent syntax. For MySQL it is START TRANSACTION; and COMMIT;.

Another important concept to understand is ACID properties of transactions. ACID stands for:

1. **Atomicity:** The whole transaction comes as one unit of work. Either all of it completes, or none of it does.
2. **Consistency:** Our database will have a number of constraints on it (foreign keys, unique keys, etc.). This property guarantees that post transaction we will be left with a valid, constraint-abiding, set of data in our database.
3. **Isolation:** This is the ability to process multiple concurrent transactions such that they do not affect one another. If you and another person are both trying to buy a ticket simultaneously, one transaction must happen first.
4. **Durability:** Once a transaction is completed then the results are permanent. For example, if we buy a ticket, then there's a power cut, our ticket will still be purchased.

With all of that out the way, let's do our table design.

City

- id BIGINT PRIMARY KEY
- name VARCHAR

Venue

- id BIGINT PRIMARY KEY
- name VARCHAR
- city BIGINT FOREIGN KEY REFERENCES city(ID)

Performance

- id BIGINT PRIMARY KEY
- name VARCHAR
- venue BIGINT FOREIGN KEY REFERENCES venue(ID)

Ticket

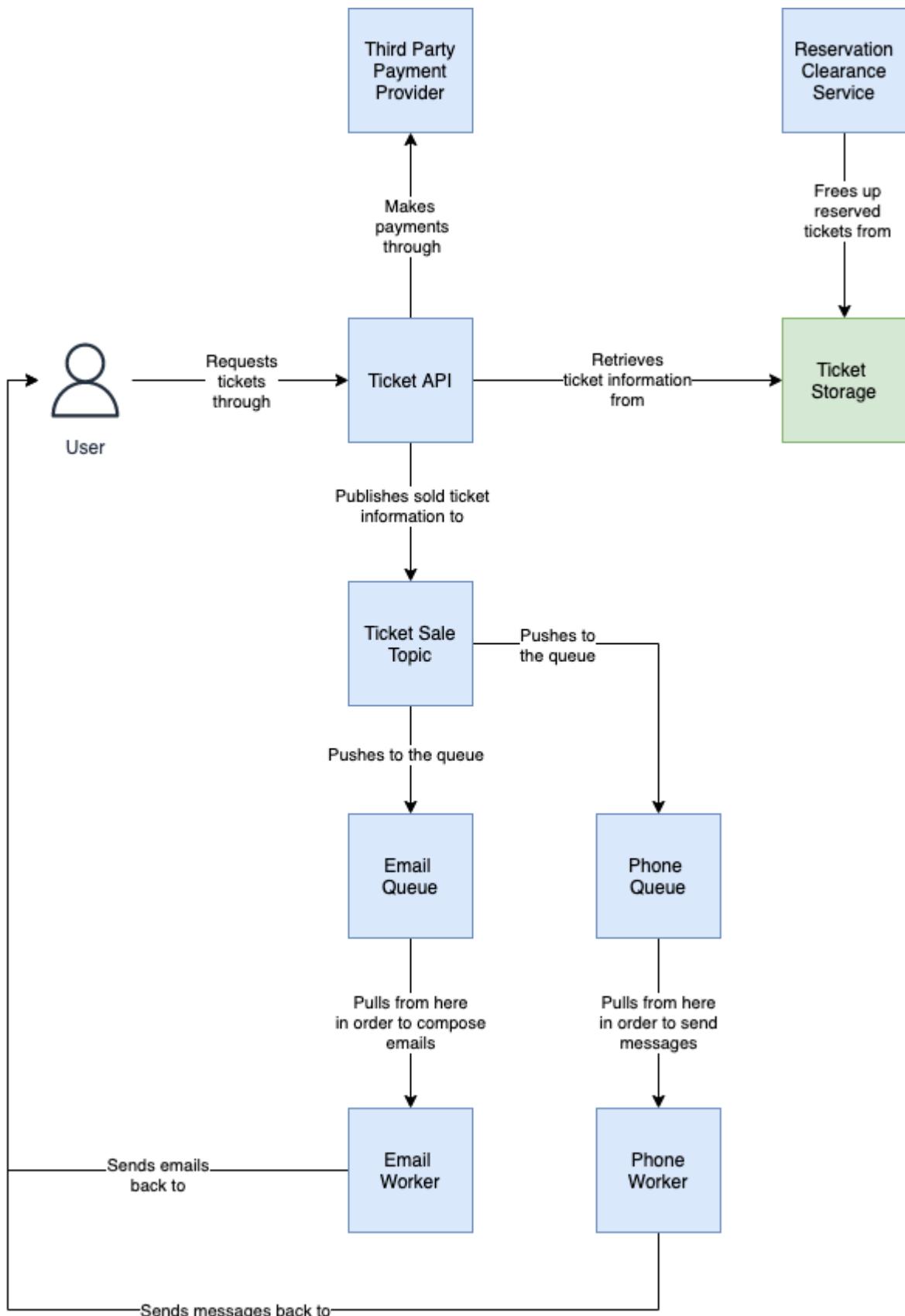
- id BIGINT PRIMARY KEY
- name VARCHAR
- performance BIGINT FOREIGN KEY REFERENCES performance(ID)
- user BIGINT FOREIGN KEY REFERENCES user(ID)
- reserved_until TIMESTAMP

User

- id BIGINT PRIMARY KEY
- name VARCHAR
- phone_number INT
- email_address VARCHAR

Logical design

The basic logical design is reasonably straightforward.



Basic Logical Design

Our user accesses ticket information through our ticket API. When a user adds a ticket to their basket, we reserve it, changing its state in the database. We also set a time x minutes in the future when the tickets will be removed from their basket.

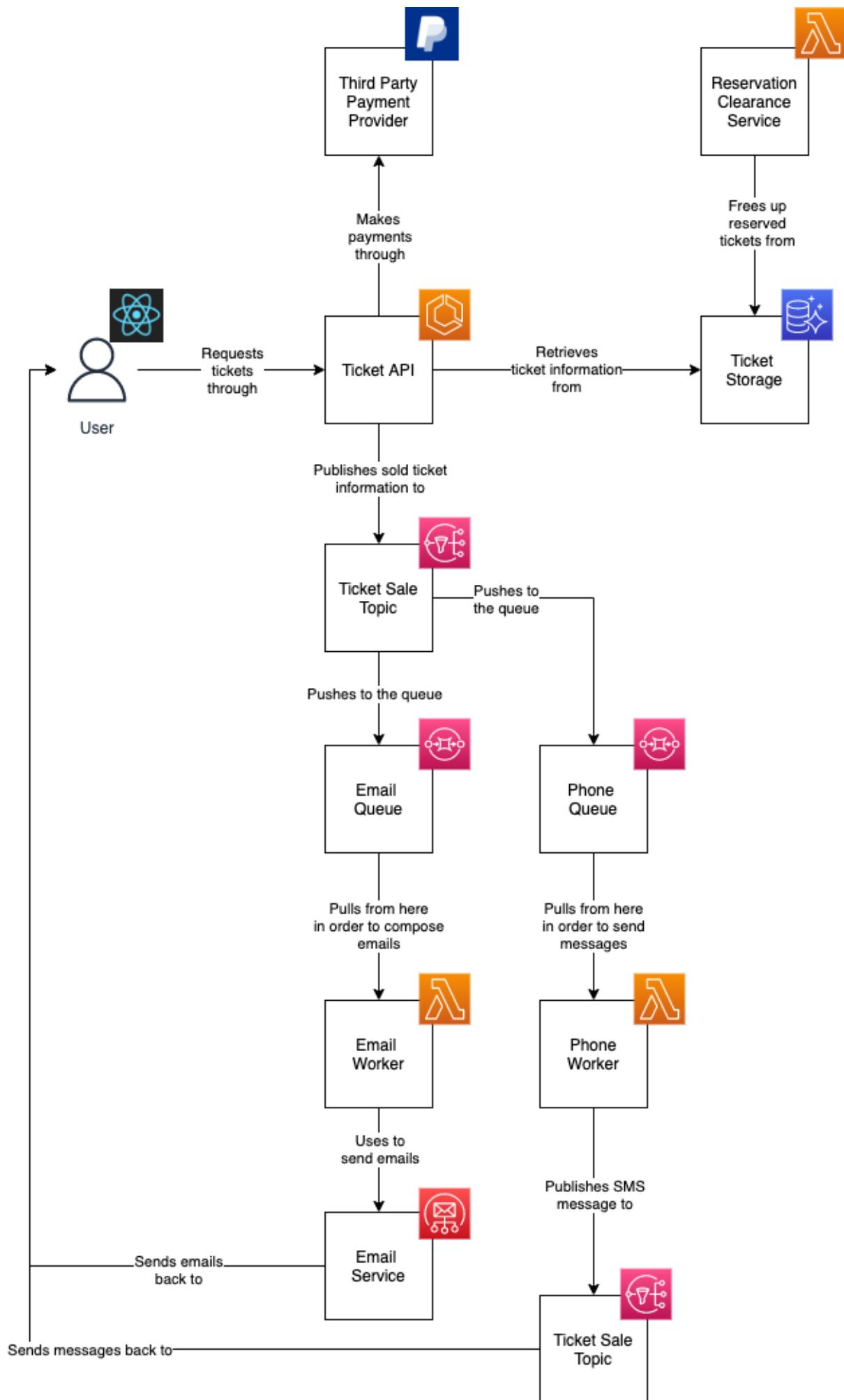
When a user purchases their ticket they will be redirected to a third party payment provider, and on success they will be redirected back to our ticket API to show a success page.

The ticket API will put a message on a sales topic, which is subscribed to by two queues. These two queues then push to two workers, who are responsible for sending emails and text messages.

The final part of the puzzle is how we free up tickets that have been reserved. This is the responsibility of another service, who polls the database every few minutes looking for expired reservations, freeing them up.

This is our very basic logical design, let's look at how we might build this.

Physical design



A basic physical design

Here is our basic physical design. Our client is in [React](#), which is provided by a Node server sitting on an [AWS ECS cluster](#). This backend service is responsible for communicating to the MySQL database sitting on [Aurora](#).

The reservation clearances are done by triggering a [Lambda](#) that runs on a timer, querying the DB and removing any outdated reservations.

Finally, on a successful sale we publish to an [SNS topic](#) subscribed to by [two SQS queues](#). Both workers are Lambdas that use either [SES](#) to send emails, or publish to a topic that handles SMS.

Another interesting thing to examine while we're here is how we might integrate our third party payment platform. We've used [PayPal](#) as an example (other platforms available).

A really good article is [here](#), and some demonstration client/ server code is [here](#). The idea is that you embed a button on your website with two methods: createOrder and onApprove.

The createOrder method is used to create an [order](#), telling PayPal what is being purchased. When you press the button we launch the PayPal checkout. On successfully completing the checkout we call the onApprove function, displaying a message to our user. The funds will safely be transferred to our PayPal account.

Identify and resolve bottlenecks

There are a number of bottlenecks in our design. Initially, we could introduce things like a CDN for a level of protection/ caching. We could also add a caching layer for data that rarely changes (the list of cities, venues and performances is a good one).

Another optimisation would be the separation of the backend APIs: one Node service for serving up the React app, and one as a backend service for dealing with the database/ third party payments.

The core optimisations we can do are around the database. To do this, let's define some terms.

Partitioning is the process of breaking up large tables into smaller chunks. By doing this we reduce query times as there is less data to query at once. There are two main types: horizontal and vertical.

Vertical scaling is where we break a table up dependent on columns. Perhaps we have some columns that contain large amounts of data. It makes sense to store them separately so we don't query them each time. This is similar to **normalisation**, where we break tables down to reduce dependency and redundancy.

Horizontal partitioning is where we break a table up dependent on rows. Each row will have the same number of columns, however there will now be multiple tables. Usually we horizontally partition based on a certain column.

Table

Table		
ID	Name	Large Column
1	Name 1	Large column 1
2	Name 2	Large column 2

Table

ID	Large Column
1	Large column 1
2	Large column 2

Table

Table		
ID	Name	Partition Column
1	Name 1	Partition 1
2	Name 2	Partition 1
3	Name 3	Partition 2
4	Name 4	Partition 2

Table

ID	Name	Partition Column
1	Name 1	Partition 1
2	Name 2	Partition 1

Table

ID	Name	Partition Column
3	Name 3	Partition 2
4	Name 4	Partition 2

Demonstrating vertical (top) and horizontal (bottom) partitioning

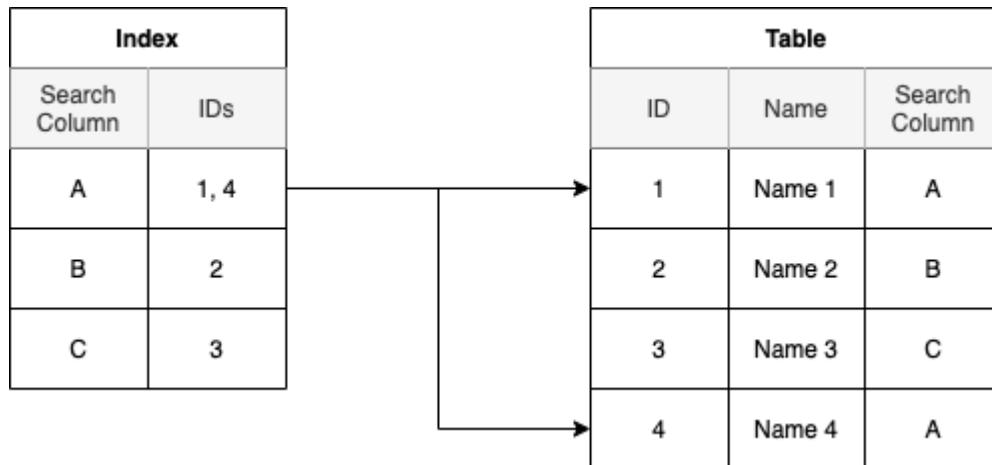
In the above we have vertically shared out the large column, and horizontally partitioned on the partition column.

Sharding is slightly different, and is a subset of horizontal partitioning. Horizontal partitioning is a logical separation of data, whereas sharding generally involves putting these logically separate blocks on different physical servers, identifying which server the data should sit on via a shard key.

But how does this relate to **indexing** I (perhaps) hear you cry? To answer this question, let's dig into what an index is.

When we store data for a table we store it in a block, with a pointer to the next block (think [linked list](#)). To find a particular block of data we need to search through all of them, one after another. As you can imagine, if what we need is right at the end of this list, this can be very slow.

To speed it up, we might have a data structure which maps the column we are searching for to the list of IDs with that value. We demonstrate below.



An index on our search column

This is a bit of a simplification, in reality they use a [B+ Tree](#) structure.

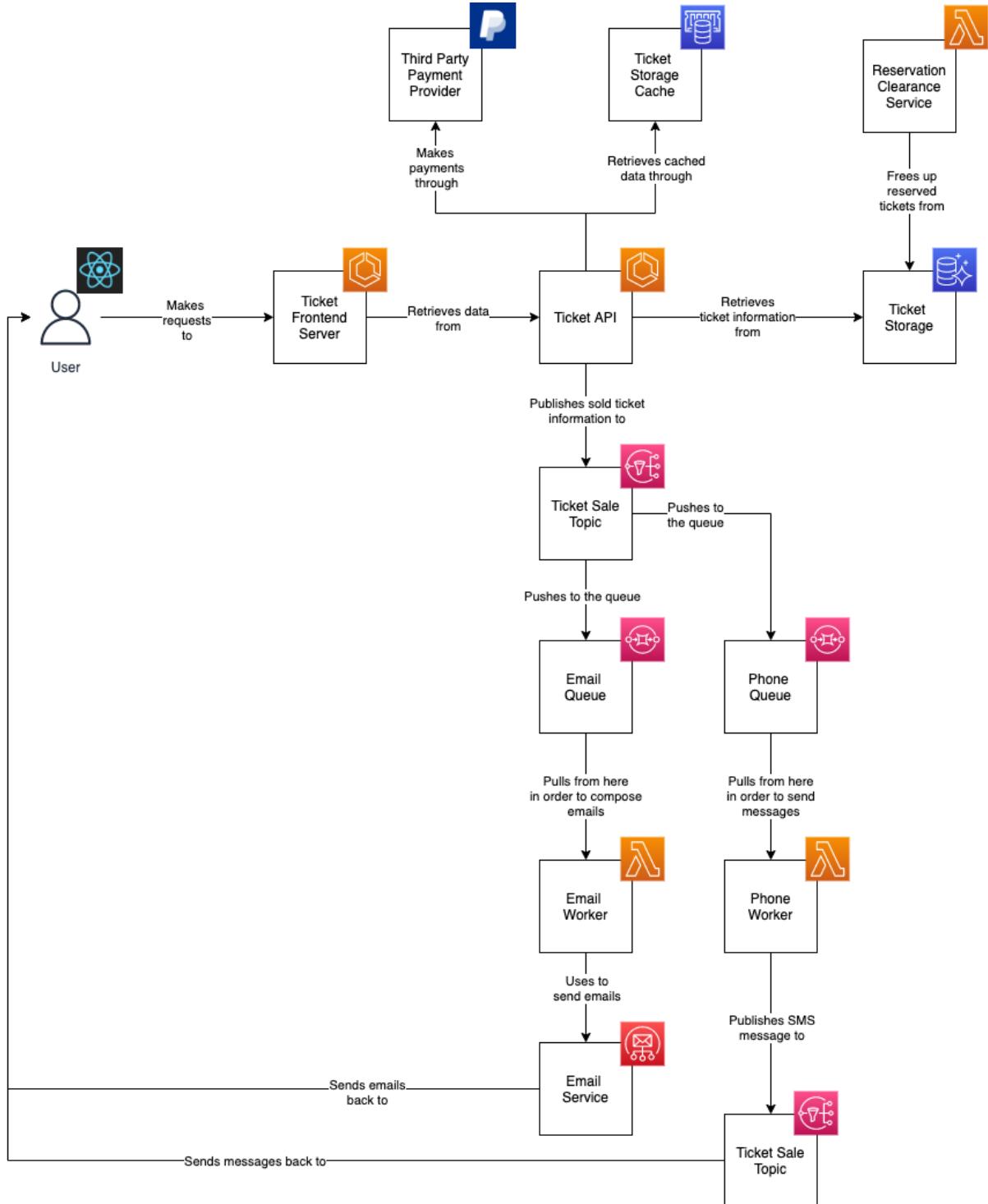
Another way to think about indexes is with a pack of cards. If I asked you to find the ace of spades, you would flick through the whole pack until you got it. However, if I separated them out into suits (equivalent to indexing on suits), it would be much easier to find!

So when include an index, and when include a partition? An index is good if you only want to access a small proportion of the data. Partitions are great if you want to access large portions of the data that you know will be grouped together.

Bringing that all together, we could partition/ shard by city! This makes sense as all venues, performances and tickets will be located per city.

The other issue we may want to address is resilience through replicas. We could have a replica per shard with a failover mechanism for if the main instance goes down.

Our final diagram is as below.



Final physical design

Solution 3:

1. Define Goals and Requirements

Tell your interviewer that you're going to support the below features. If the interviewer wants to add some more features he/she will mention that.

- The portal should list down the different cities where the theatres are located. (RDBMS)
- Once the user selects the city it should display the movies released in that particular city to that user.
- Once the user selects the movie, the portal should display the cinemas running that movie and the available shows.
- Users should be able to select the show at a particular theatre & book the tickets (third-party payment support).
- Send a copy of tickets via SMS notification or Email. (workers and GCM)
- Movies suggestions when login (Hadoop and spark streaming with ML to get recommendation engine), real-time notifications to the user about new movie releases & other stuff.
- The portal should display the seating arrangement of the cinema hall to the user.
- Users should be able to select multiple seats according to their choice.
- Users should be able to hold the seats for 5-10 minutes before he/she finalized the payment.
- The portal should serve the tickets in a First In First Out manner
- Comments and rating (Cassandra)
- The system should be highly concurrent because there will be multiple booking requests for the same seat at the same time.
- The core thing of the portal is ticket bookings which mean financial transactions. So the system should be secure and ACID compliant.
- Responsive design (ReactJS and Bootstrap) to run on devices of various sizes like mobile, tablet, desktop, etc.
- Movie information.

2. How does Bookmyshow Talk to Theatres?

When you visit any third-party application/movie tickets aggregator using the mobile app or website, you see the available and occupied seats for a movie show in that theatre. Now the question is how these third-party aggregators talk to the theatres, get the available seat information, and display it to the users. Definitely, the app needs to work with the theatre's server to get the seat allocation and give it to the users. There are mainly two strategies to allocate the seats to these aggregators.

- A specific number of seats will be dedicated to every aggregator and then these seats will be offered to the users. In this strategy, some seats are already reserved for these aggregators, so there is no need to keep updating the seat information from all the theatres.
- In the second strategy, the app can work along with the theatre and other aggregators to keep updating the seat availability information. Then the ticket will be offered to the users.

3. How to Get The Seat Availability Information?

There are mainly two ways to get this information...

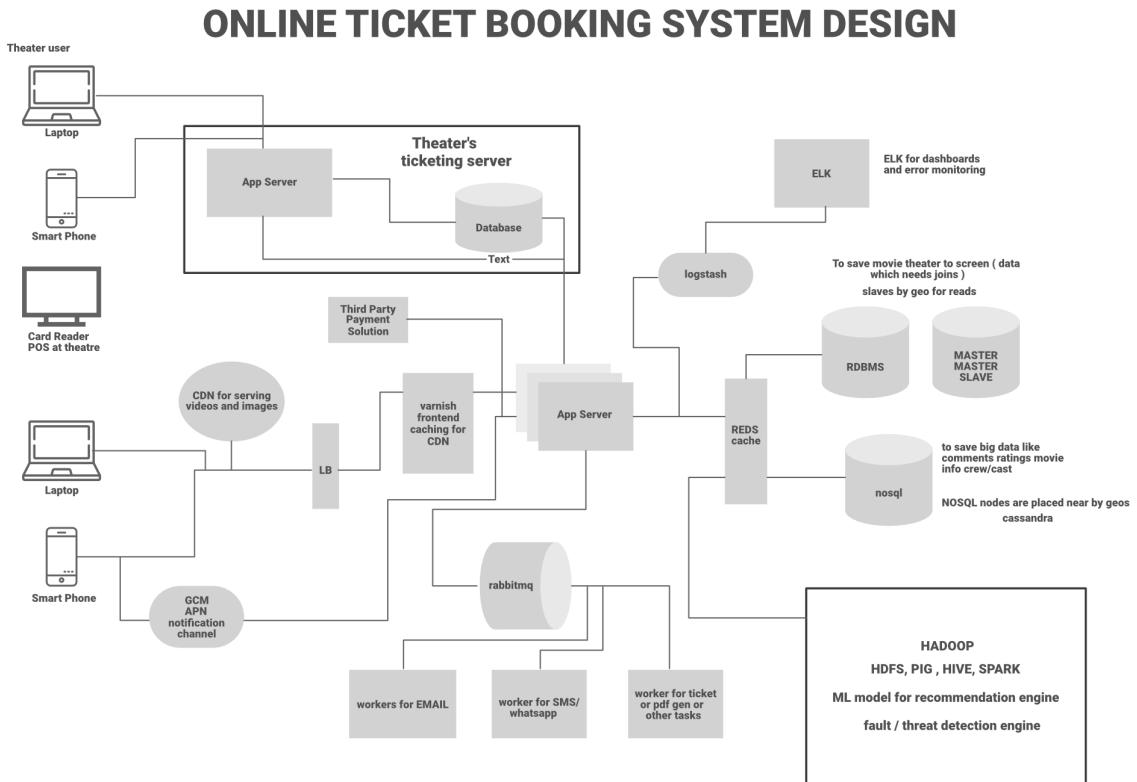
- The aggregators can connect to the theatre's DB directly and get the information from the database table. Then this information can be cached and displayed to the user.
- Use the theatre's server API to get the available seat information and book the tickets.

What will happen if multiple users will try to book the same ticket using different platforms? How to solve this problem?

The theatre's server needs to follow a timeout locking mechanism strategy where a seat will be locked temporarily for a user for a specific time session (for example, 5-10 minutes). If the user is not able to book the seat within that timeframe then release the seat for another user. This should be done on a first come first serve basis.

If you're using the theatres' server API then you will be making a lot of requests or IO blocking calls from your server to the theatre's server. To achieve better performance we should use async in python or Erlangs lightweight threads or Go Coroutines in Go.

High-Level Architecture



BookMyShow is built on **microservice architecture**. Let's look at the components individually.

Load Balancer

A load balancer is used to distribute the load on the server and to keep the system highly concurrent when we are scaling the app server horizontally. The load balancer can use multiple techniques to balance the load and these are...

1. Consistent Hashing
2. Round Robin
3. Weighted Round Robin
4. Least Connection

Frontend Caching and CDN

We do frontend caching using **Varnish** to reduce the load from the backend infrastructure. We can also use CDN Cloudflare to cache the pages, API, video, images, and other content.

App Servers

There will be multiple **app servers** and BookMyShow uses **Java, Spring Boot, Swagger, and Hibernate** for the app servers. We can also go with Python-based or NodeJS servers (Depending on requirements). We also need to scale these app servers horizontally to take the heavy load and to handle a lot of requests in parallel.

Elastic Search

Elastic search is used to support the search APIs on Bookmyshow (to search movies or shows). Elastic search is distributed and it has RESTful search APIs available in the system. It can be also used as an analytics engine that works as an App-level search engine to answer all the search queries from the front-end.

Caching

To save the information related to the movies, seat ordering, theatres, etc, we need to use **caching**. We can use **Memcache or Redis** for caching to save all this information in Bookmyshow. Redis is open-source and it can be also used for the locking mechanism to block the tickets temporarily for a user. It means when a user is trying to book the ticket Redis will block the tickets with a specific TTL.

Database

We need to use both RDBMS and NoSQL databases for different purposes. Let's analyze what we need in our system and which database is suitable for what kind of data...

- **RDBMS:** We have mentioned that we need ACID property in our system. Also, we have countries, cities, theatres in cities, multiple screens in these theatres, and multiple rows of seats on each screen. So here it is clear that we need a proper relationship representation. Also, we need to handle the transaction. RDBMS is fit for these cases. The portal will be read-heavy so we need to shard the data by Geo or we need to use master-master slave architecture. Slaves can be used for reading and master for writing.

- **NoSQL:** We also have a huge amount of data like movie information, actors, crew, comments, and reviews. RDBMS can not handle this much amount of data so we need to use the NoSQL database which can be distributed. Cassandra can be a good choice to handle this ton of information. We can save multiple copies of data in multiple nodes deployed in multiple regions. This ensures the high availability and durability of data (if a node goes down, we will have data available in other nodes).
- Use **HDFS** to run queries for analytics.

Async Workers

The main task of the Async worker is to execute the tasks such as generating the pdf or png of the images for booked tickets and sending the notification to the users. For push notifications, SMS notifications, or emails we need to call third-party APIs. These are network IO and it adds a lot of latency. Also, these time-consuming tasks can not be executed synchronously. To solve this problem, as soon as the app server confirms the booking of the tickets, it will send the message to the message queue, a free worker will pick up the task, execute it asynchronously and provide the SMS notification, other notifications, or email to the users. **RabbitMQ** or **Kafka** can be used for Message Queueing System and **Python celery** can be used for workers. For browser notification or phone Notification use **GCM/ APN**.

Business Intelligence and ML

For data analysis of business information, we need to have a **Hadoop** platform. All the logs, user activity, and information can be dumped into Hadoop, and on top of it, we can run **PIG/Hive** queries to extract information like user behavior or user graph. ML is used to understand the user's behavior and to generate movie recommendations etc. For real-time analysis, we can use **Spark** streaming. We can also figure out fraud detection and mitigation strategy using the Spark or **Storm** stream processing engine.

Log Management

ELK (ElasticSearch, Logstash, Kibana) stack is used for the logging system. All the logs are pushed into the Logstash. Logstash collects data from all the servers via Files/Syslog/socket/AMQP etc and based on a different set of filters it redirects the logs to Queue/File/Hipchat/Whatsapp/JIRA etc.

Step By Step Working

- The customer visits the portal and filters the location. To find the location we can use GPS (if it's a mobile phone) or ISP (if it's a laptop). Then the theatres and movies released in that theatres will be suggested to the user. Data will be provided from DB and ELK recommendation engine.
- Users select the movie and check the different timing for the movies in all the nearby theatres.
- Users select a particular date and time for a movie in his/her own choice of theatre. The available seat information will be fetched from the database and it will be displayed to the user.

- Once the user selects the available seat, BMS locks the seat temporarily for the next 10 minutes. BMS interacts with the theatres DB and blocks the seat for the user. The ticket will be booked temporarily for the current user and for all the other users using different aggregators or apps the seat will be unavailable for the next 10 minutes. If the user fails to book the ticket within that timeframe the seat will be released for the other aggregators.
- If the user continues with the booking he/she will check the invoice with the payment option and after the payment via the payment gateway, the app server will be notified about the successful payment.
- After successful payment, a unique ID will be generated by the theatre and it will be provided to the app server.
- Using that unique ID a ticket will be generated with a QR code and a copy of the ticket will be shown to the user. Also, a message to the queue will be added to send the invoice copy of the ticket to the user via SMS notification or email. The ticket will have all the details such as the movie, address of the theatre, timing, theatre number, etc.
- At movie time, when the customer visits the theatre the QR code will be scanned and the customer will be allowed to enter the theatre if the ID will be matched on both customer's ticket and the theatre's ticket.

APIs Needed

- GetListOfCities()
- GetListOfEventsByCity(CityId)
- GetLocationsByCity(CityId)
- GetLocationsByEventandCity(cityid, eventid)
- GetEventsByLocationandCity(CityId, LocationId)
- GetShowTiming(eventid, locationid)
- GetAvailableSeats(eventid, locationid, showtimeid)
- VerifyUserSelectedSeatsAvailable(eventid, locationid, showtimeid, seats)
- BlockUserSelectedSeats()
- BookUserSelectedSeat()
- GetTimeoutForUserSelectedSeats()

RDBMS Tables

- Place (To save the hierarchical data for any given theatre-like country, state, city, and street)
- Theatre
- Screen
- Tier (tier of seats)
- Seats
- Movie
- Offers
- Ticket
- User

Relationship Between RDBMS Tables:

- **One to many:** Place and theatre.
- **One to many:** Theatre and screen
- **One to many:** Screen and Tier
- **One to many:** Tier and seats
- **One to one:** Screen and Movie
- **One to many:** User and Tickets
- **One to many:** Tickets and Seats

NoSQL Tables

There will be no relationship between these tables.

- Comments
- Ratings
- Movie Information
- Trailers or Gallery
- Artists
- Cast and Crew
- Reviews
- Analytics Data

Technologies Used By Bookmyshow

- **User Interface:** ReactJS & BootStrapJS
- **Server language and Framework:** Java, Spring Boot, Swagger, Hibernate
- **Security:** Spring Security
- **Database:** MySQL
- **Server:** Tomcat
- **Caching:** In memory cache Hazelcast.
- **Notifications:** RabbitMQ. A Distributed message queue for push notifications.
- **Payment API:** Popular ones are Paypal, Stripe, Square
- **Deployment:** Docker & Ansible
- **Code repository:** Git
- **Logging:** Log4J
- **Log Management:** ELK Stack
- **Load balancer:** Nginx

A lot of candidates get afraid of the system design round more than the coding round. The reason is... they don't get the idea that what topics and tradeoffs they should cover within this limited timeframe. They need to keep in mind that the system design round is extremely open-ended and there's no such thing as a standard answer. For the same questions, the conversation with the different interviewers can be different. Your practical experience, your knowledge, your understanding of the modern software system, and how you express yourself clearly during your interview matter a lot to designing a system successfully.

15. Designing Yelp or Nearby Friends

Solution 1:

Let's design a Yelp like service, where users can search for nearby places like restaurants, theaters, or shopping malls, etc., and can also add/view reviews of places. Similar Services: Proximity server.

1. Why Yelp or Proximity Server?

Proximity servers are used to discover nearby attractions like places, events, etc. If you haven't used yelp.com before, please try it before proceeding (you can search for nearby restaurants, theaters, etc.) and spend some time understanding different options that the website offers. This will help you a lot in understanding this chapter better.

2. Requirements and Goals of the System

What do we wish to achieve from a Yelp like service? Our service will be storing information about different places so that users can perform a search on them. Upon querying, our service will return a list of places around the user.

Our Yelp-like service should meet the following requirements:

Functional Requirements:

1. Users should be able to add/delete/update Places.
2. Given their location (longitude/latitude), users should be able to find all nearby places within a given radius.
3. Users should be able to add feedback/review about a place. The feedback can have pictures, text, and a rating.

Non-functional Requirements:

1. Users should have a real-time search experience with minimum latency.
2. Our service should support a heavy search load. There will be a lot of search requests compared to adding a new place.

3. Scale Estimation

Let's build our system assuming that we have 500M places and 100K queries per second (QPS). Let's also assume a 20% growth in the number of places and QPS each year.

4. Database Schema

Each Place can have the following fields:

1. LocationID (8 bytes): Uniquely identifies a location.
2. Name (256 bytes)
3. Latitude (8 bytes)

4. Longitude (8 bytes)
5. Description (512 bytes)
6. Category (1 byte): E.g., coffee shop, restaurant, theater, etc.

Although a four bytes number can uniquely identify 500M locations, with future growth in mind, we will go with 8 bytes for LocationID.

Total size: $8 + 256 + 8 + 8 + 512 + 1 \Rightarrow 793$ bytes

We also need to store reviews, photos, and ratings of a Place. We can have a separate table to store reviews for Places:

1. LocationID (8 bytes)
2. ReviewID (4 bytes): Uniquely identifies a review, assuming any location will not have more than 2^{32} reviews.
3. ReviewText (512 bytes)
4. Rating (1 byte): how many stars a place gets out of ten.

Similarly, we can have a separate table to store photos for Places and Reviews.

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. The following could be the definition of the API for searching:

```
search(api_dev_key, search_terms, user_location, radius_filter, maximum_results_to_return,
category_filter, sort, page_token)
```

Parameters:

`api_dev_key` (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

`search_terms` (string): A string containing the search terms.

`user_location` (string): Location of the user performing the search.

`radius_filter` (number): Optional search radius in meters.

`maximum_results_to_return` (number): Number of business results to return.

`category_filter` (string): Optional category to filter search results, e.g., Restaurants, Shopping Centers, etc.

`sort` (number): Optional sort mode: Best matched (0 - default), Minimum distance (1), Highest rated (2).

`page_token` (string): This token will specify a page in the result set that should be returned.

Returns: (JSON)

A JSON containing information about a list of businesses matching the search query. Each result entry will have the business name, address, category, rating, and thumbnail.

6. Basic System Design and Algorithm

At a high level, we need to store and index each dataset described above (places, reviews, etc.). For users to query this massive database, the indexing should be read efficient, since while searching for the nearby places users expect to see the results in real-time.

Given that the location of a place doesn't change that often, we don't need to worry about frequent updates of the data. As a contrast, if we intend to build a service where objects do change their location frequently, e.g., people or taxis, then we might come up with a very different design.

Let's see what are different ways to store this data and also find out which method will suit best for our use cases:

a. SQL solution

One simple solution could be to store all the data in a database like MySQL. Each place will be stored in a separate row, uniquely identified by LocationID. Each place will have its longitude and latitude stored separately in two different columns, and to perform a fast search; we should have indexes on both these fields.

To find all the nearby places of a given location (X, Y) within a radius 'D', we can query like this:

```
Select * from Places where Latitude between X-D and X+D and Longitude between Y-D and Y+D
```

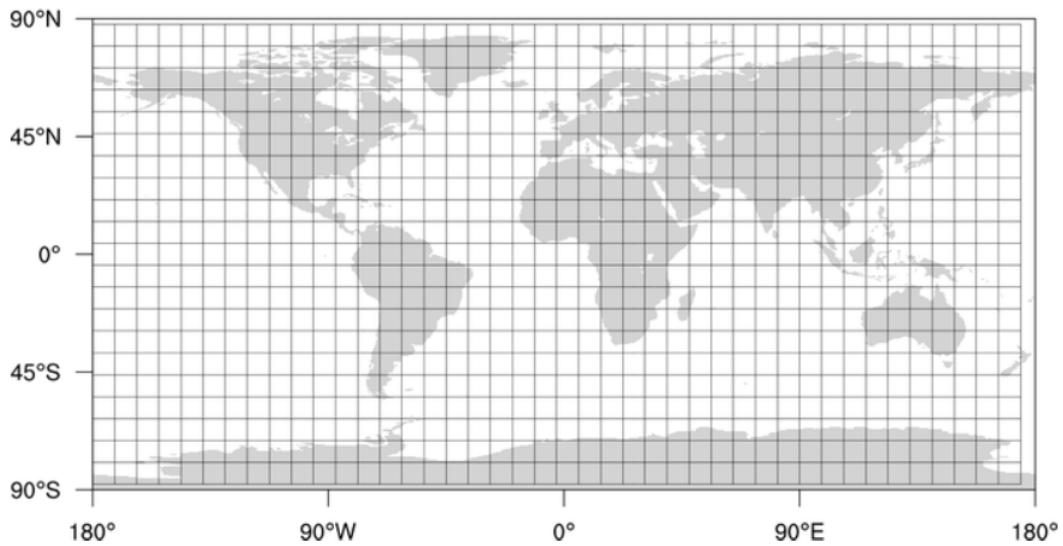
The above query is not completely accurate, as we know that to find the distance between two points we have to use the distance formula (Pythagorean theorem), but for simplicity let's take this.

How efficient would this query be? We have estimated 500M places to be stored in our service. Since we have two separate indexes, each index can return a huge list of places and performing an intersection on those two lists won't be efficient. Another way to look at this problem is that there could be too many locations between 'X-D' and 'X+D', and similarly between 'Y-D' and 'Y+D'. If we can somehow shorten these lists, it can improve the performance of our query.

b. Grids

We can divide the whole map into smaller grids to group locations into smaller sets. Each grid will store all the Places residing within a specific range of longitude and latitude. This scheme would enable us to query only a few grids to find nearby places. Based on a given location and radius, we can find all the neighboring grids and then query these grids to find nearby places.

Grid of two dimensional data



Let's assume that GridID (a four bytes number) would uniquely identify grids in our system.

What could be a reasonable grid size? Grid size could be equal to the distance we would like to query since we also want to reduce the number of grids. If the grid size is equal to the distance we want to query, then we only need to search within the grid which contains the given location and neighboring eight grids. Since our grids would be statically defined (from the fixed grid size), we can easily find the grid number of any location (lat, long) and its neighboring grids.

In the database, we can store the GridID with each location and have an index on it, too, for faster searching. Now, our query will look like:

```
Select * from Places where Latitude between X-D and X+D and Longitude between Y-D and Y+D and GridID in (GridID, GridID1, GridID2, ..., GridID8)
```

This will undoubtedly improve the runtime of our query.

Should we keep our index in memory? Maintaining the index in memory will improve the performance of our service. We can keep our index in a hash table where 'key' is the grid number and 'value' is the list of places contained in that grid.

How much memory will we need to store the index? Let's assume our search radius is 10 miles; given that the total area of the earth is around 200 million square miles, we will have 20 million grids. We would need a four bytes number to uniquely identify each grid and, since LocationID is 8 bytes, we would need 4GB of memory (ignoring hash table overhead) to store the index.

$$(4 * 20M) + (8 * 500M) \approx 4 \text{ GB}$$

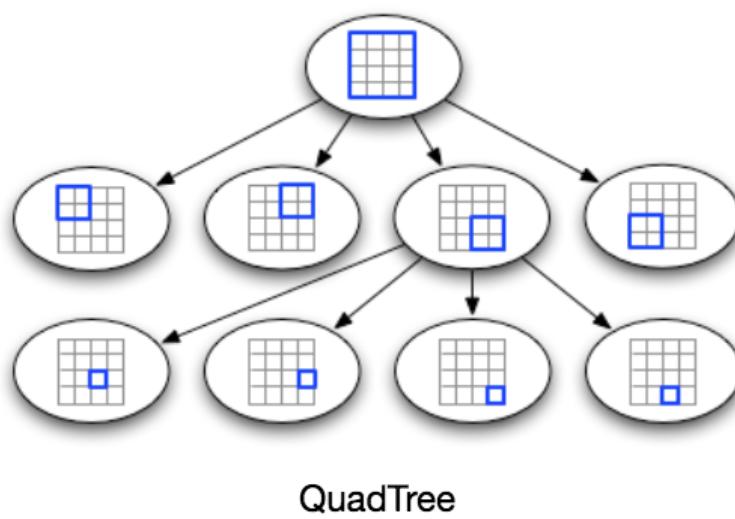
This solution can still run slow for those grids that have a lot of places since our places are not uniformly distributed among grids. We can have a thickly dense area with a lot of places, and on the other hand, we can have areas which are sparsely populated.

This problem can be solved if we can dynamically adjust our grid size such that whenever we have a grid with a lot of places we break it down to create smaller grids. A couple of challenges with this approach could be: 1) how to map these grids to locations and 2) how to find all the neighboring grids of a grid.

c. Dynamic size grids

Let's assume we don't want to have more than 500 places in a grid so that we can have a faster searching. So, whenever a grid reaches this limit, we break it down into four grids of equal size and distribute places among them. This means thickly populated areas like downtown San Francisco will have a lot of grids, and sparsely populated area like the Pacific Ocean will have large grids with places only around the coastal lines.

What data-structure can hold this information? A tree in which each node has four children can serve our purpose. Each node will represent a grid and will contain information about all the places in that grid. If a node reaches our limit of 500 places, we will break it down to create four child nodes under it and distribute places among them. In this way, all the leaf nodes will represent the grids that cannot be further broken down. So leaf nodes will keep a list of places with them. This tree structure in which each node can have four children is called a [QuadTree](#)



How will we build a QuadTree? We will start with one node that will represent the whole world in one grid. Since it will have more than 500 locations, we will break it down into four nodes and distribute locations among them. We will keep repeating this process with each child node until there are no nodes left with more than 500 locations.

How will we find the grid for a given location? We will start with the root node and search downward to find our required node/grid. At each step, we will see if the current node we are visiting has children. If it has, we will move to the child node that contains our desired location and repeat this process. If the node does not have any children, then that is our desired node.

How will we find neighboring grids of a given grid? Since only leaf nodes contain a list of locations, we can connect all leaf nodes with a doubly linked list. This way we can iterate forward or backward among the neighboring leaf nodes to find out our desired locations. Another approach for finding adjacent grids would be through parent nodes. We can keep a pointer in each node to access its parent, and since each parent node has pointers to all of its children, we can easily find siblings of a node. We can keep expanding our search for neighboring grids by going up through the parent pointers.

Once we have nearby LocationIDs, we can query the backend database to find details about those places.

What will be the search workflow? We will first find the node that contains the user's location. If that node has enough desired places, we can return them to the user. If not, we will keep expanding to the neighboring nodes (either through the parent pointers or doubly linked list) until either we find the required number of places or exhaust our search based on the maximum radius.

How much memory will be needed to store the QuadTree? For each Place, if we cache only LocationID and Lat/Long, we would need 12GB to store all places.

$$24 * 500M \Rightarrow 12 \text{ GB}$$

Since each grid can have a maximum of 500 places, and we have 500M locations, how many total grids we will have?

$$500M / 500 \Rightarrow 1M \text{ grids}$$

Which means we will have 1M leaf nodes and they will be holding 12GB of location data. A QuadTree with 1M leaf nodes will have approximately 1/3rd internal nodes, and each internal node will have 4 pointers (for its children). If each pointer is 8 bytes, then the memory we need to store all internal nodes would be:

$$1M * 1/3 * 4 * 8 = 10 \text{ MB}$$

So, total memory required to hold the whole QuadTree would be 12.01GB. This can easily fit into a modern-day server.

How would we insert a new Place into our system? Whenever a new Place is added by a user, we need to insert it into the databases as well as in the QuadTree. If our tree resides on one server, it is easy to add a new Place, but if the QuadTree is distributed among different servers, first we need to find the grid/server of the new Place and then add it there (discussed in the next section).

7. Data Partitioning

What if we have a huge number of places such that our index does not fit into a single machine's memory? With 20% growth each year we will reach the memory limit of the server in the future. Also, what if one server cannot serve the desired read traffic? To resolve these issues, we must partition our QuadTree!

We will explore two solutions here (both of these partitioning schemes can be applied to databases, too):

a. Sharding based on regions: We can divide our places into regions (like zip codes), such that all places belonging to a region will be stored on a fixed node. To store a place we will find the server through its region and, similarly, while querying for nearby places we will ask the region server that contains user's location. This approach has a couple of issues:

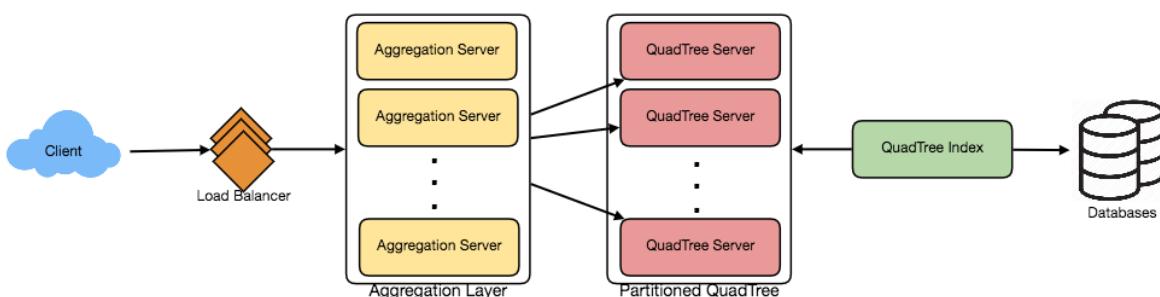
1. What if a region becomes hot? There would be a lot of queries on the server holding that region, making it perform slow. This will affect the performance of our service.
 2. Over time, some regions can end up storing a lot of places compared to others. Hence, maintaining a uniform distribution of places, while regions are growing is quite difficult.

To recover from these situations, either we have to repartition our data or use consistent hashing.

b. Sharding based on LocationID: Our hash function will map each LocationID to a server where we will store that place. While building our QuadTree, we will iterate through all the places and calculate the hash of each LocationID to find a server where it would be stored. To find places near a location, we have to query all servers and each server will return a set of nearby places. A centralized server will aggregate these results to return them to the user.

Will we have different QuadTree structure on different partitions? Yes, this can happen since it is not guaranteed that we will have an equal number of places in any given grid on all partitions. However, we do make sure that all servers have approximately an equal number of Places. This different tree structure on different servers will not cause any issue though, as we will be searching all the neighboring grids within the given radius on all partitions.

The remaining part of this chapter assumes that we have partitioned our data based on LocationID.



8. Replication and Fault Tolerance

Having replicas of QuadTree servers can provide an alternate to data partitioning. To distribute read traffic, we can have replicas of each QuadTree server. We can have a

master-slave configuration where replicas (slaves) will only serve read traffic; all write traffic will first go to the master and then applied to slaves. Slaves might not have some recently inserted places (a few milliseconds delay will be there), but this could be acceptable.

What will happen when a QuadTree server dies? We can have a secondary replica of each server and, if primary dies, it can take control after the failover. Both primary and secondary servers will have the same QuadTree structure.

What if both primary and secondary servers die at the same time? We have to allocate a new server and rebuild the same QuadTree on it. How can we do that, since we don't know what places were kept on this server? The brute-force solution would be to iterate through the whole database and filter LocationIDs using our hash function to figure out all the required places that will be stored on this server. This would be inefficient and slow; also, during the time when the server is being rebuilt, we will not be able to serve any query from it, thus missing some places that should have been seen by users.

How can we efficiently retrieve a mapping between Places and QuadTree server? We have to build a reverse index that will map all the Places to their QuadTree server. We can have a separate QuadTree Index server that will hold this information. We will need to build a HashMap where the 'key' is the QuadTree server number and the 'value' is a HashSet containing all the Places being kept on that QuadTree server. We need to store LocationID and Lat/Long with each place because information servers can build their QuadTrees through this. Notice that we are keeping Places' data in a HashSet, this will enable us to add/remove Places from our index quickly. So now, whenever a QuadTree server needs to rebuild itself, it can simply ask the QuadTree Index server for all the Places it needs to store. This approach will surely be quite fast. We should also have a replica of the QuadTree Index server for fault tolerance. If a QuadTree Index server dies, it can always rebuild its index from iterating through the database.

9. Cache

To deal with hot Places, we can introduce a cache in front of our database. We can use an off-the-shelf solution like Memcache, which can store all data about hot places. Application servers, before hitting the backend database, can quickly check if the cache has that Place. Based on clients' usage pattern, we can adjust how many cache servers we need. For cache eviction policy, Least Recently Used (LRU) seems suitable for our system.

10. Load Balancing (LB)

We can add LB layer at two places in our system 1) Between Clients and Application servers and 2) Between Application servers and Backend server. Initially, a simple Round Robin approach can be adopted; that will distribute all incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is if a server is dead the load balancer will take it out of the rotation and will stop sending any traffic to it.

A problem with Round Robin LB is, it won't take server load into consideration. If a server is overloaded or slow, the load balancer will not stop sending new requests to that server. To

handle this, a more intelligent LB solution would be needed that periodically queries backend server about their load and adjusts traffic based on that.

11. Ranking

How about if we want to rank the search results not just by proximity but also by popularity or relevance?

How can we return most popular places within a given radius? Let's assume we keep track of the overall popularity of each place. An aggregated number can represent this popularity in our system, e.g., how many stars a place gets out of ten (this would be an average of different rankings given by users)? We will store this number in the database as well as in the QuadTree. While searching for the top 100 places within a given radius, we can ask each partition of the QuadTree to return the top 100 places with maximum popularity. Then the aggregator server can determine the top 100 places among all the places returned by different partitions.

Remember that we didn't build our system to update place's data frequently. With this design, how can we modify the popularity of a place in our QuadTree? Although we can search a place and update its popularity in the QuadTree, it would take a lot of resources and can affect search requests and system throughput. Assuming the popularity of a place is not expected to reflect in the system within a few hours, we can decide to update it once or twice a day, especially when the load on the system is minimum.

Solution 2:

The System to Design

In all honesty I've never used Yelp or Nearby friends, I tend to use Google Maps! However, the idea is the same. You are represented by a point on a map, and you can search in your immediate vicinity for things like pubs, restaurants, pubs, theatres, pubs, cinemas, pubs, train stations, or even pubs! If you've not used one of these services before an example using Google Maps is [here](#).

Our system requires the following:

- We need to be able to search for locations by name and location.
- We need to be able to comment/ rate and upload photos of locations.

Our usual non-functional requirements stand. It must be reliable, scalable and available.

The Approach

We have a standard approach to system design which is explained more thoroughly in the article [here](#). However the steps are summarised below:

1. **Requirements clarification:** Making sure we have all the information before starting. This may include how many requests or users we are expecting.

2. **Back of the envelope estimation:** Doing some quick calculations to gauge the necessary system performance. For example, how much storage or bandwidth do we need?
3. **System interface design:** What will our system look like from the outside, how will people interact with it? Generally this is the API contract.
4. **Data model design:** What our data will look like when we store it. At this point we could be thinking about relational vs non-relational models.
5. **Logical design:** Fitting it together in a rough system! At this point I'm thinking at a level of 'how would I explain my idea to someone who knows nothing about tech?'
6. **Physical design:** Now we start worrying about servers, programming languages and the implementation details. We can superimpose these on top of the logical design.
7. **Identify and resolve bottlenecks:** At this stage we will have a working system! We now refine the design.

With that said, let's get stuck in!

Requirements Clarification

The first thing I would be wondering would be how many users we would be expecting and how frequently they will be using the system. I would also want a rough estimate of the number of locations, the memory requirements of a location, the average number of locations returned per search and the average image size.

Let's say we have 2 million locations. We have 1 million users, each of which are reading twice a day with 10 results, and writing once a week. A location takes up around 1MB of storage, whereas an image takes about 100KB (these estimates aren't that realistic).

Back of the envelope estimation

We can turn the above requirements into a back of the envelope estimation. $1,000,000 * 2 * 10 = 20,000,000$ reads a day = 240 reads a second. Equally, $1,000,000 = 1.5$ writes a second. Initially we see we have a read heavy system. We're reading 240MB/s (not including image requests) and writing around 150KB/s (assuming reviews and comments are around the same size as an image).

In terms of storage, we're going to need around $2,000,000 * 1\text{MB} = 2\text{TB}$ of storage, not including adding more locations!

System interface design

Having decided we've got a fairly big system on our hands, let's look at how we would like to interact with it. Initially we would like to be able to search by name or location. We can do this by using a GET request to `/search?query=<query text>` or `/search?longitude=<longitude>&latitude=<latitude>` endpoints, which return a list of locations. We could add more parameters (radius, maximum results etc.), but we will standardise this for now.

A location object may look similar to the below:

Note, we've assumed that the rating system is essentially a liking system, where the count is the number of likes.

From this endpoint we can then return a 200 for a successful response, with the usual 4XX, 5XX errors if anything goes wrong.

Commenting, rating and photos follow a similar pattern. We can use a POST request to /location/{id}/comments, /location/{id}/ratings or /location/{id}/images. The body of the post to comments can use the comment object included in the location.json. A rating wouldn't need much in the body, it would be enough just to say someone has liked it.

The image would be slightly more complex. We might think of using a form and multipart-form-data. A more thorough coverage of this is in the article [here](#).

Each of these endpoints would return a 201 for created, then the regular 4XX and 5XX error codes.

Note, we're not bothering with tracking users in this example, but in the real world we would need some sort of user management. We're also assuming users can't submit locations, only read them.

Data model design

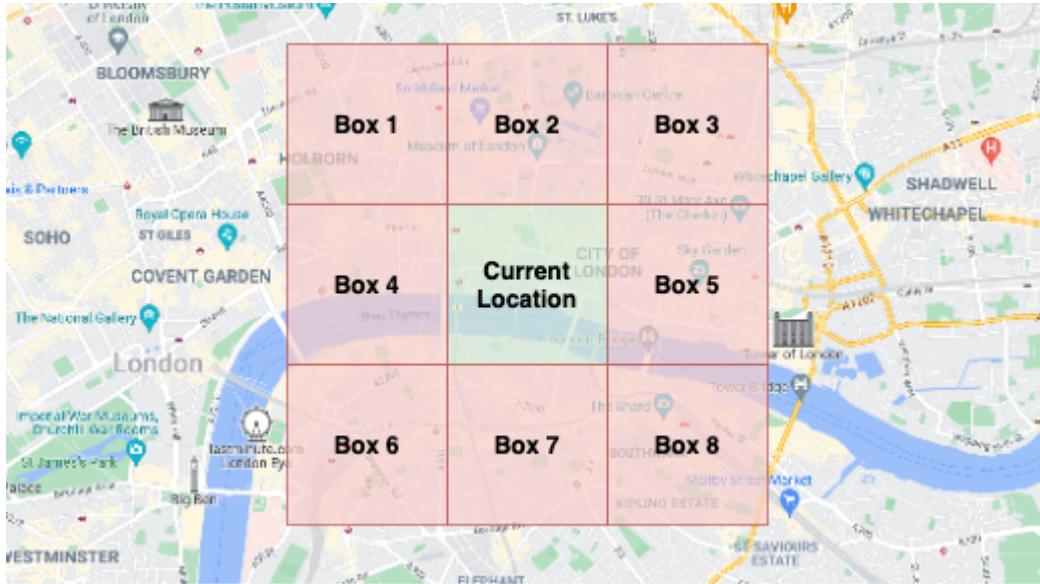
The next thing we need to do is design our data model. We have a very solid location object, however we need to understand how we might map that to persistent storage.

There are two options, SQL and NoSQL. SQL offers us consistency of data, which is good in some respects. However, in this case consistency doesn't bother us too much. If a user doesn't see a location or review for a few moments, that's not too big a worry. We can also leverage the scalability of NoSQL to support the demand on our service. Let's go for NoSQL.

The other thing we need to understand is how our data model will affect our search capabilities. Let's pretend we're always searching for locations within a 10 mile radius of our current location.

One naive approach would be to have a massive SQL table with longitude and latitude columns, where we search for all locations with both fields within a 10 mile radius. However, we can immediately see this would become ineffective the more locations we get.

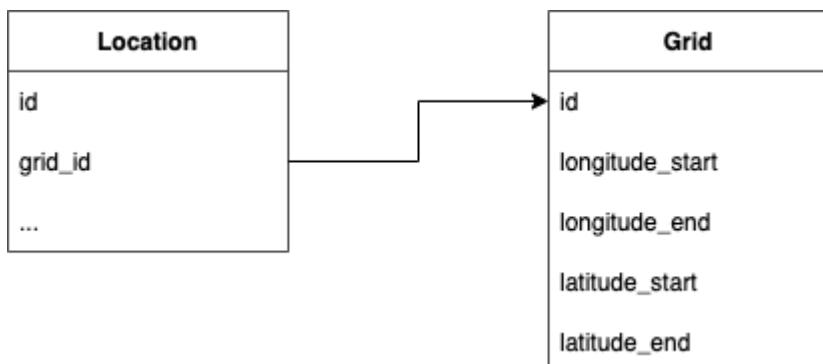
A better approach would be to divide our map into grids.



Overlaying a grid onto Google Maps

In this design we have split London up into small squares, each of which contains a number of locations. To calculate the locations in a radius we can search for the current and adjoining grids.

We originally had two million locations. Let's say the earth is one million kilometres squared (highly unlikely, but roll with it for the sake of example). We break the world map down into squares of 10km squared. This makes $1,000,000 / 10 = 100,000$ squares. Suddenly we've moved from searching two million rows, to searching 100,000!

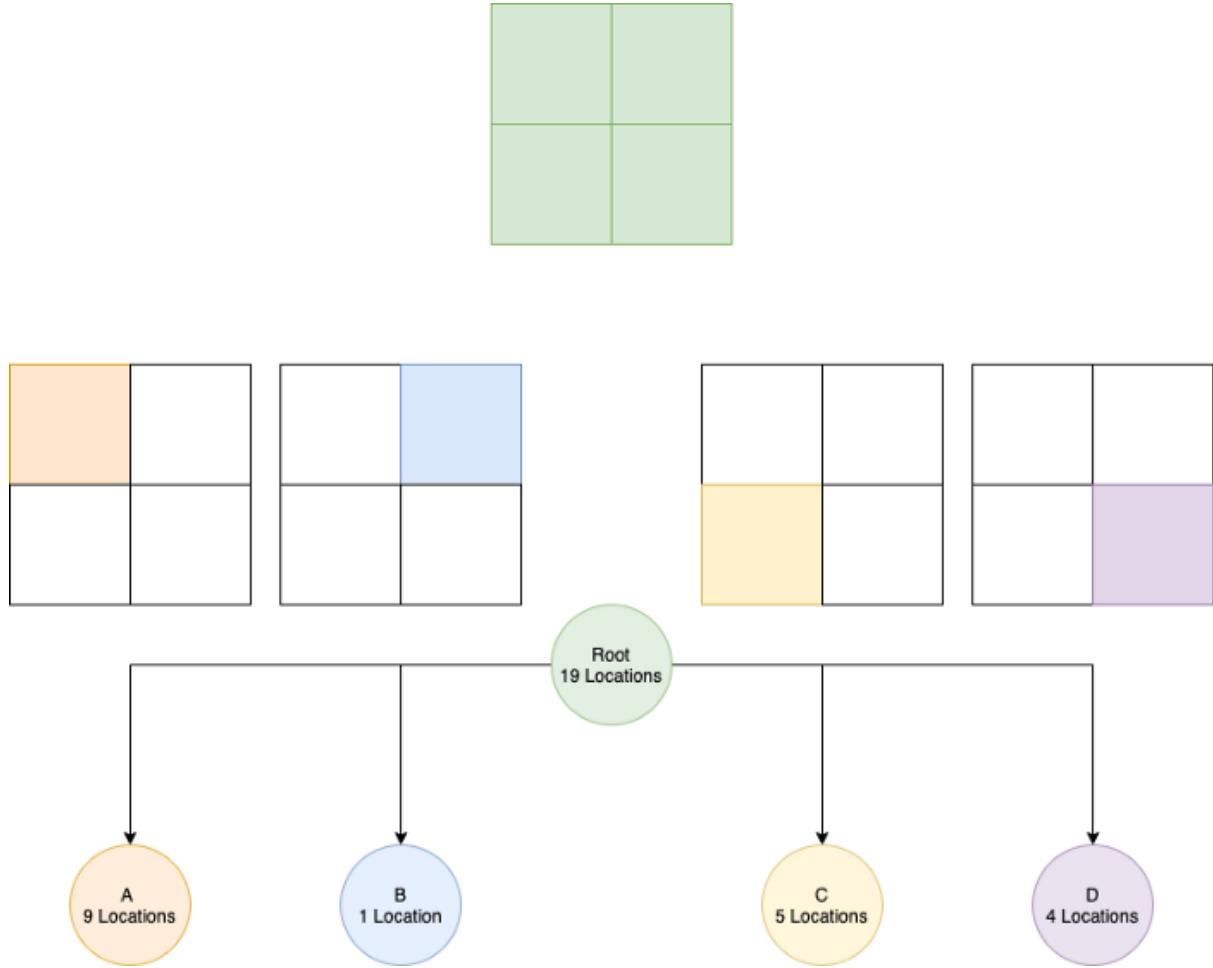


An example of a potential new grid data model and how we map locations to grids.

There may be further issues with this approach. There's a lot of things to do in London, but less so in the middle of the Pacific Ocean! Some of our grids may be empty, and some of them may be far too full.

Enter, the [QuadTree](#). At its core this is a tree structure where each node has four children. For those of you less familiar with trees, I'd recommend reading my article [here](#), and my article on [tries](#) [here](#). The tries article is especially relevant as it demonstrates the process of setting a threshold, after which we break down a node into other nodes.

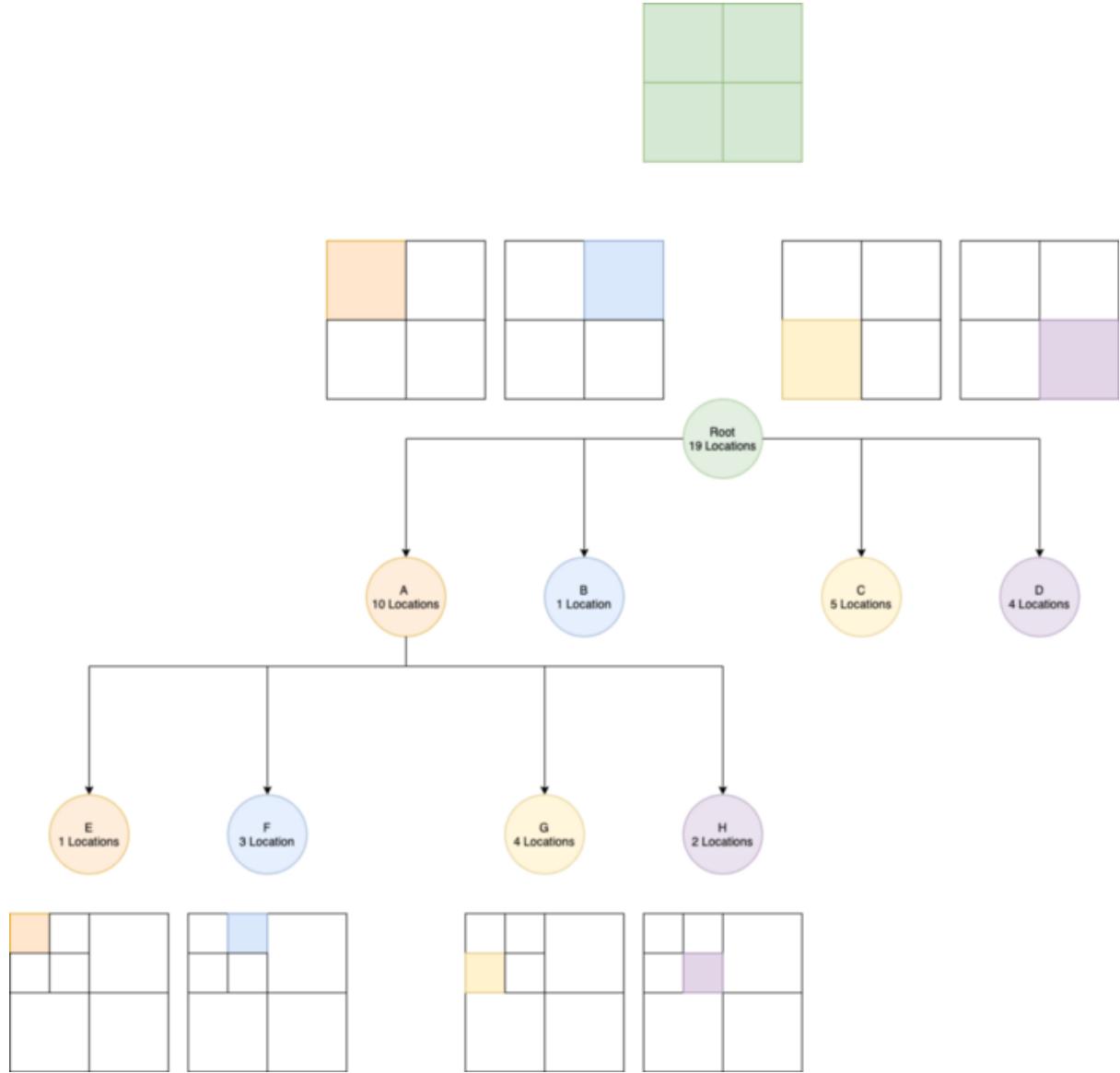
Imagine we have a tree design as below:



A starting example of our quadtree

Initially, our root node represents our entire map (hence all the squares are green). We can split this area down further into four quadrants, represented by A, B, C and D, each containing their relevant location information. The trigger we have for doing this splitting is a threshold, which has been set at 10. What this means is every time a node has 10 locations we split it down further.

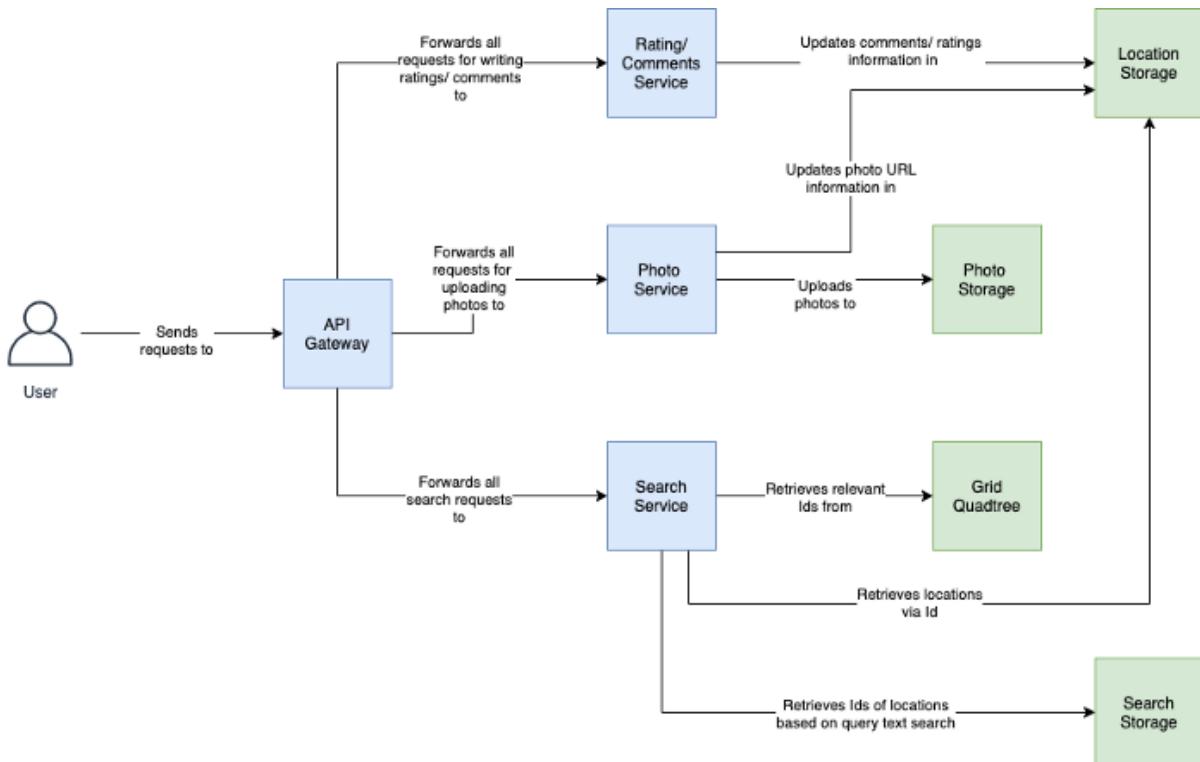
We can add another location to A to demonstrate.



Splitting up a node once it has reached the threshold

To search we then recurse down the tree! This is our index of areas to location Ids.

Logical design



A basic logical design is contained above

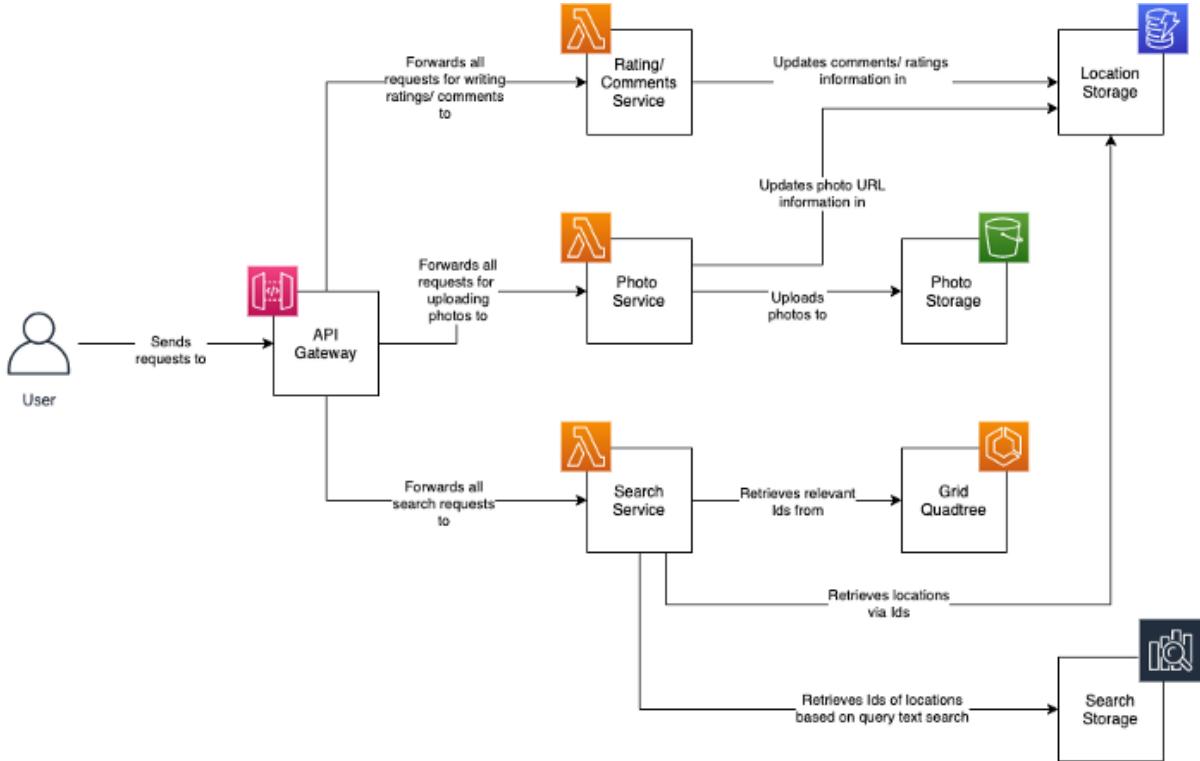
A user sends their requests to an API Gateway, which forwards requests for writing ratings and comments to one service, requests for uploading photos to another service, and search requests to another.

The ratings/ comments service is responsible for updating the persisted locations with ratings/ comments information. The photo service receives photo upload requests and is responsible for saving them to some form of storage. Note, we're ignoring encoding for this example, as otherwise the system will get too complex to be helpful.

Finally, the search service interacts with the grid quadtree for retrieving location IDs dependent on longitude and latitude, then using these to query the location storage. It can also look up any query text using the inverted index.

Although this seems fairly straightforward, when we move onto the later section we'll see there's a lot of optimisation to be done!

Physical design



A basic physical design

Here is a proposed initial physical design. All requests are sent to an [AWS API Gateway](#), and there are [three Lambdas](#) subscribed to it, each representing one of the consuming services. Our location storage is done through [DynamoDb](#), our image storage is done in [S3](#).

We're hosting our grid quadtree on a cluster of ECS instances. Notice, this is different to distributing the quadtree. We will have several copies of the tree, but the whole tree will reside on each node. Our inverted index will be done using AWS [OpenSearch](#).

An interesting point to note is some people don't separate out their NoSQL storage and their index. This is fine, you can [use ElasticSearch as your primary data storage](#). However, I probably wouldn't recommend it, unless your use case definitely doesn't depend on any of the properties mentioned in that article.

Identify and resolve bottlenecks

The first step we can make to optimise our design is the introduction of a CDN. Let's use this as an opportunity to dive into how they work.

CDNs

When we host our site, we are normally hosting it from a set region of the world. Let's say it's hosted in Europe. If a user wants to visit it from San Francisco then the information must travel all the way between continents! This can be quite slow.

It would be useful for us to be able to store copies of parts of the website all over the world. For example, if we know a page won't change frequently, we can store a copy in L.A., improving the San Francisco load times.

This is the role of CDNs. These geographically distributed servers help cache content including HTML, Javascript, stylesheets, video, and images. It can also improve security, reduce costs, and increase availability. Pretty nifty, eh?

Improving security is done in a couple of ways. Initially, as we will discuss, users can connect directly to a CDN. [This means that we can use the CDN's certificate and connection for TLS/SSL](#) (for a recap on TLS/SSL you can see my article [here](#)).

CDNs will prioritise making their connections as secure as possible, and will handle certificate rotation, which takes a load off our backs.

They also provide [protection against DDoS attacks](#). They can detect unusual traffic patterns, respond by dropping suspicious packets, route remaining valid traffic, and adapt for the future by remembering previously offensive parties.

So how do we set up a CDN for our service?

In a cloud-driven world we might use something like [AWS Cloudfront](#), which we set up in front of our web-facing service. Requests are no longer sent to our service, but instead all proxied through the CDN. In our responses we may express [caching headers](#) to tell our CDN more about how long it should keep hold of this response, or alternatively we may update the CDN manually with new files.

Allowing the CDN to request content before caching it is known as the ‘pull’ method, whereas putting our own files up is known as ‘push’.

The other thing we can do to improve our performance is partitioning. As our system grows we need to be able to scale, and at some juncture we will no longer be able to store our all of our information on single servers.

There are two main places we can look at partitioning our data. At the quadtree level and at the location level.

Sharding the quadtree we can do by region. To clarify, imagine we want to split our tree in half. All we need to do is split the tree, put one half on one server, and the other on another. We then introduce a gateway component in front of the servers, forwarding requests to the required servers.

There is a potential to introduce an index between the quadtree server number and the places stored on it, however it seems a bit unnecessary.

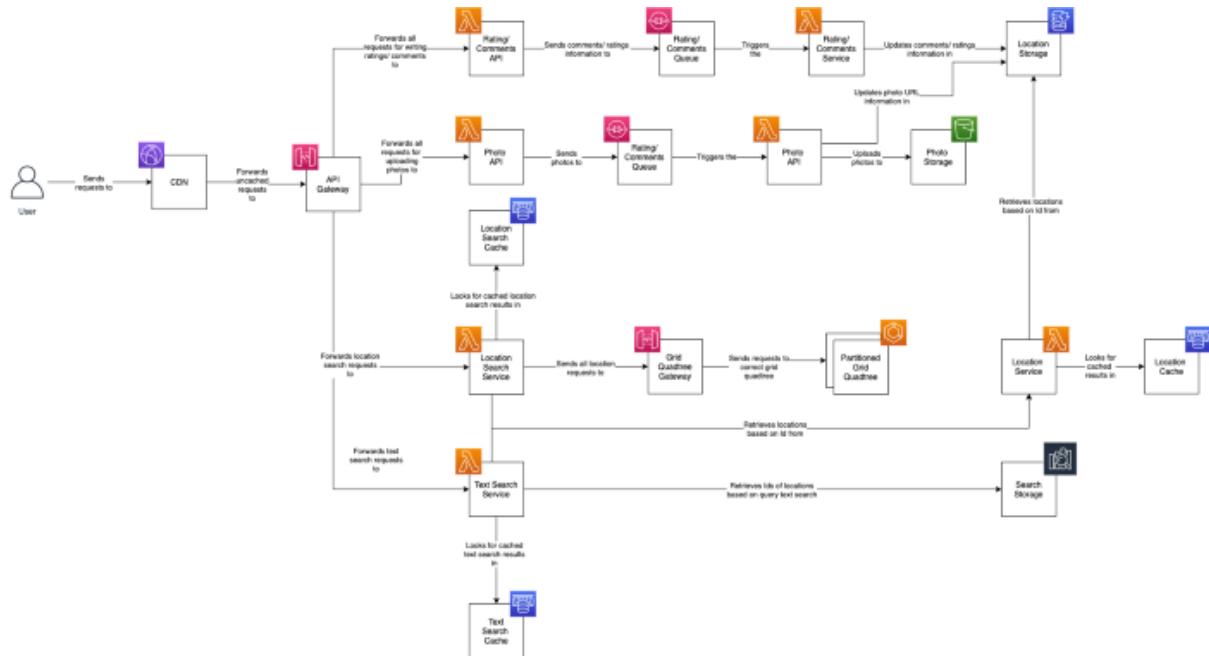
We can shard the locations by their Id. When one of our search services returns a location Id, a component in front of our fleet of servers will be responsible for directing the request to the appropriate place.

We would also introduce replicas of our data, and allow failovers in case of emergency.

Another optimisation we could do would be to separate out our search services, one in front of the quadtree, one in front of OpenSearch, and then a third one serving up requests for locations based on grid Id. We can also add a cache to our search services.

Finally, we can add resiliency by putting queues in between the write services and changing the response codes to 201 (accepted). This means if downstream systems are unavailable we don't lose information.

Our finished design would look similar to the below.



A potential final, physical design

Solution 3:

- 1) Users can add/delete/update Places
- 2) Given their location (long/lat) users should be able to find all nearby places within a given radius
- 3) Users should be able to add feedback/review about a place. Feedback can have pics, text & rating

Functional Requirements

- 1) Users can add/delete/update Places
- 2) Given their location (long/lat) users should be able to find all nearby places within a given radius
- 3) Users should be able to add feedback/review about a place. Feedback can have pics, text & rating

Non-functional requirements

- 1) Real-time user experience with **minimal latency**
- 2) Heavy search load - more search requests vs adding new place - **heavy read**

Scale estimation

- 500M places
- 100k queries per second
- Assume 20% growth in no. of places & QPS each year

DB schema

Each **place** can have following fields:

- Location ID (8 bytes)
- Name (256 bytes)
- Latitude (8 bytes)
- Longitude (8 bytes)
- Description (512 bytes)
- Category (1 byte)

Total size is 793 bytes per place

Also need to store **reviews**, **photos** and **ratings** of a place. Have separate table store each.

System APIs for Yelp

search (api_dev_key, search_terms, user_location, radius_filter, maximum_results_to_return, category_filter, sort, page_token)

Basic system design & algorithm

Need to store & index each dataset. Indexing should be READ efficient since users need real time results. Given location of user doesn't change often so don't need to worry about updating

SQL Solution of storing data for Yelp (MySQL)

Each place stored in separate row, uniquely identified by Location ID. Each place will have long & lat stored separately in different columns (can have indexed).

- To find all nearby places for given location (X,Y) within radius D - can use pythagorus
- **Efficiency:** Not very since 500M places with 2 separate indexes so each index can return large list of places & perform intersection on 2 lists. Ideally need to shorten lists.

Grid solution of storing data for Yelp

Divide map into smaller grids to group locations into smaller sets (within long & lat). Scheme enables query only a few grids to find nearby places. Use Grid ID (4 bytes) to uniquely identify grids in system

Reasonable grid size?

- equal to distance we want to query so only need to search within grid. Statically defined so easily find grid no. of any location & neighbouring grids.
- In DB we can store GridID with each location
- Can maintain index in memory (key our index in hash table, key = grid number & value = list of places in grid) improving performance

How much memory will we need to store index?

- Assume search radius is 10 miles
- Total area of earth is 200M sq miles so 20M grids
- Need 4 bytes number to uniquely identify each grid & location ID is 8 bytes
- $(4 \times 20M) + (8 \times 500M) = 4GB$ memory ignoring hashtable overhead

Issues with Grids storage

Still can run slowly in densely populated areas since not uniformly distributed. Challenges with dynamic approach: 1) How to map grids to location 2) how to find all neighbouring grids of grid

Dynamic size grid solution for Yelp

- Assume don't want >500 places in a grid for faster searching so when grid reaches limit, we can break it down into **4 grids of equal size** + distribute places among them
- **Tree** where **each node has 4 children (QuadTree)** can be data structure. Each node represents a grid and contain info about all places in that grid. All leaf nodes represent grids that cannot be further broken down.

How do we build QuadTree?

How will we find grid for given location?

- Start with 1 node = world
- >500 locations, break down into four nodes and distribute locations among them. Keep repeating with each child node until no nodes left with more than 500 locations.
- Start with root node and search downward to find required node/grid. At each step, check current node we are visiting has children.
- If has, move to child node containing desired location and repeat.
- If node doesn't, then that is our desired node.

How will we find neighboring grids of a given grid?

Since only leaf nodes contain list of locations, can **connect all leaf nodes with doubly linked list** so can iterate forward/backward among neighboring leaf nodes to find desired locations.

Or go through parent nodes. Keep a **pointer in each node to access its parent**, and since each parent node has pointers to all of its children, we can find siblings of node. Can keep expanding search for neighboring grids by going up through **parent pointers**.

Once we have nearby LocationIDs, can query backend DB to find details about places

What is the search workflow?

- 1) Find the node that contains the user's location
- 2) If that node has enough desired places, can return them to the user.
- If not, keep expanding to neighboring nodes (either through parent pointers or doubly linked list) until either we find the required number of places or exhaust search based on MAX radius

How much memory needed to store the QuadTree?

- If we cache only LocationID & Lat/Long, (24*500M) need 12GB
- Will have 1M grids since 500M locations/500 places per grid.
- Have 1M leaf nodes holding 12GB of location data.
- QuadTree with 1M leaf nodes will have approx 1/3 rd internal nodes and each internal node will have 4 pointers (for its children). If each pointer is 8 bytes, then memory we need to store all internal nodes would be:
 $1M \times 1/3 \times 4 \text{ pointers} \times 8 \text{ bytes} = 10\text{MB}$
- Total memory for Quad Tree is 12.01GB (can fit into server)

How would we insert a new Place into our system?

- need to insert into DBs & the QuadTree.
- If tree resides on 1 server, easy to add a new Place, but if QuadTree distributed among different servers, need to find the grid/server of the new Place and then add it there
- Sharding based on regions
 - Divide places into regions (zip codes), so all places belonging to a region will be stored on a fixed node.
 - To store a place we will find the server through its region and, similarly, while querying for nearby places we will ask the region server that contains user's location.
- Issues with sharding based on regions
 - For hot region, there would be lots of queries on server holding that region, slowing performance, affecting service.
 - Over time, some regions can end up storing lots of places vs others. So maintaining uniform distribution of places, while regions are growing is quite difficult.

Solutions: Repartition data or use consistent hashing

Sharding based on LocationID

- Hash function will map each LocationID to a server.
- While building QuadTree, will iterate through all places and calculate hash of each LocationID to find server where it would be stored.
- To find places near location, have to query all servers and each server will return a set of nearby places
- Centralized server aggregates results to return them to user

Will we have different QuadTree structure on different partitions?

- Yes, this can happen since not guaranteed that we will have an equal number of places in any given grid on all partitions.
 - BUT make sure all servers have approx equal no of Places.
 - Diff tree structure on diff servers will not cause issue, as will be searching all neighboring grids within given radius on all partitions
- High level system design for Yelp
- Replication & Fault Toleration
- Replicas of QuadTree servers can provide alt to data partitioning

- To distribute read traffic can have replicas of QuadTree server. Have **master-slave config where slaves will only serve read traffic**; all **write traffic will go to master** & then applied to slaves. Slaves might not have recently inserted places but acceptable

- If QuadTree server dies, can have 2ndary replica of each server, which takes control after failover

What if both primary and secondary QuadTree servers die at the same time?

- Allocate new server + rebuild same QuadTree on it.
- Can iterate through whole DB and filter LocationIDs using hash function to determine all required places that will be stored on server.
- Inefficient and slow + when server being rebuilt, will not be able to serve any query, thus missing some places that should have been seen by users.

How can we efficiently retrieve a mapping between Places and QuadTree server?

- Build reverse index that will map all Places to their QuadTree server. Separate QuadTree Index server that holds info.
- Need to build HashMap: 'key' = QuadTree server no. and 'value' = HashSet containing all Places kept on QuadTree server.
- Need to store LocationID and Lat/Long with each place because info servers can build QuadTrees.
- Keeping Places' data in HashSet, enabling us to add/remove Places from our index quickly. So whenever QuadTree server needs to rebuild itself, can simply ask QuadTree Index server for all Places needs to store.
- Fast approach and should have replica of QuadTree Index server for fault tolerance. If QuadTree Index server dies, can always rebuild index from iterating through DB

How can we return most popular places within a given radius?

- Keep track of overall popularity of each place (aggregated no. e.g. stars out of 10).
- Store number in DB in QuadTree
- While searching within given radius, can ask partitions of QuadTree to return top 100 places with max popularity
- Aggregator server can determine top 100 places among all places returned by partitions
- Can update popularity of place in QuadTree once or twice a day when load on system less

Solution 4:

Design a proximity server like NearBy or Yelp Part — 1

What are Proximity Servers?

Proximity servers are applications like NearBy or Yelp which are helpful in discovering attractions like restaurants, theaters, temples, or recreational places that are adjacent to your location.

Functional Requirements

1. Search functionality is the core of a proximity server. Users should be able to search all the proximate places within a given radius for any location (latitude and longitude).
2. Users can add new places or edit the old ones with the basic details like images, brief description, etc.
3. Users can give ratings(1 to 5 stars) and reviews(text and images) to places.

Non-Functional Requirements

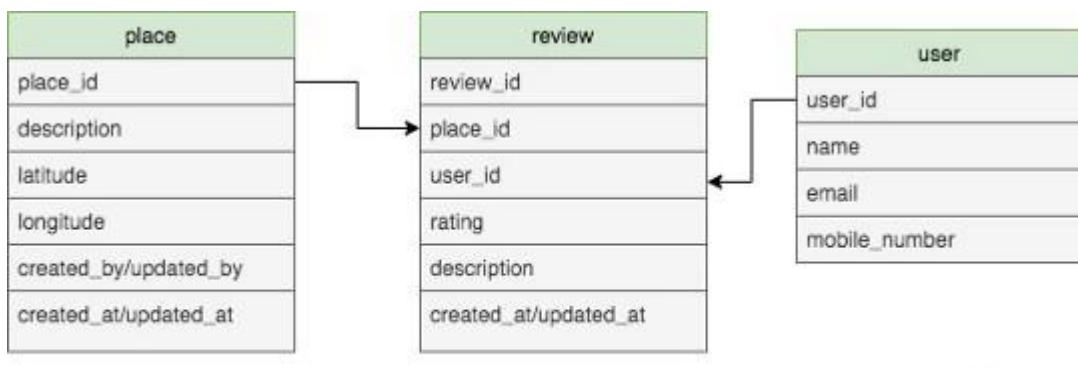
1. The system should be highly available with real-time search experience and minimum latency.
2. We can expect this application to be read-heavy as there will be a myriad of search requests compared to the frequent addition of new places.

Scale Estimation

We can estimate the total number of places in the system to be **200 Million** with **100K requests per second**. Considering the future scale for **5 years** with **20% growth per year** we should build our system for at least a scale of **5 years**.

- Our system should be ready to manage the mammoth scale of **400 Million**.
- Our system should be able to handle the load of **200K requests per second**.

Basic Database Schema



APIs Design:

1. Search API:

```
GET /searchRequest: query_param = {  
    search_terms: "Pizza",  
    radius: 10,           (in kilometres)  
    category_filter: "Italian Restaurants", (optional field)  
    filter: "5 star rated restaurants", (optional field)  
    maximun_no_of_results: 20      (optional field)
```

}Response: A JSON containing information about a list of popular places matching the search query. Each result entry will have the place name, address, category, rating, and image.

2. Add places:

POST /placesRequest: body = {

```
place_name: "Julia's Kitchen",
latitude: 85,
longitude: 10,
category: "Restaurant",
description: "Best Italian restaurant",
photos: ["url1", "url2", "url3"]      (Array of photos)
```

{Response}: Response of the newly created place with place id

3. Add reviews:

POST /reviewsRequest: body = {

```
user_id: "uid121",
place_id: "place1",
rating: 4,                      (integer between 1 to 5)
review: "Restaurant was good and clean",
photos: ["url1", "url2", "url3"]   (Array of photos)
```

Response: Response of the newly created review with it's id.

1. Simple SQL based storage

Places, reviews, User details can be stored in SQL DB easily and latitude and longitude can be indexed for search optimization.

Basic Search Flow:

To find all the places nearby for a given location with latitude and longitude (100, 85) within a radius 10Km, the query will be :

Select * from Places where latitude between 100-10 and 100+10 and longitude between 85-10 and 85+10);

Place Id	Latitude	Longitude
10001	98	87
10002	101	82
10003	104	85
10004	93	89

Considering our scale **400 Million** places, this query will not be efficient for such a mammoth load as we have two separate indexes and each index can return a huge list of places and performing an intersection on these two lists will be slow.

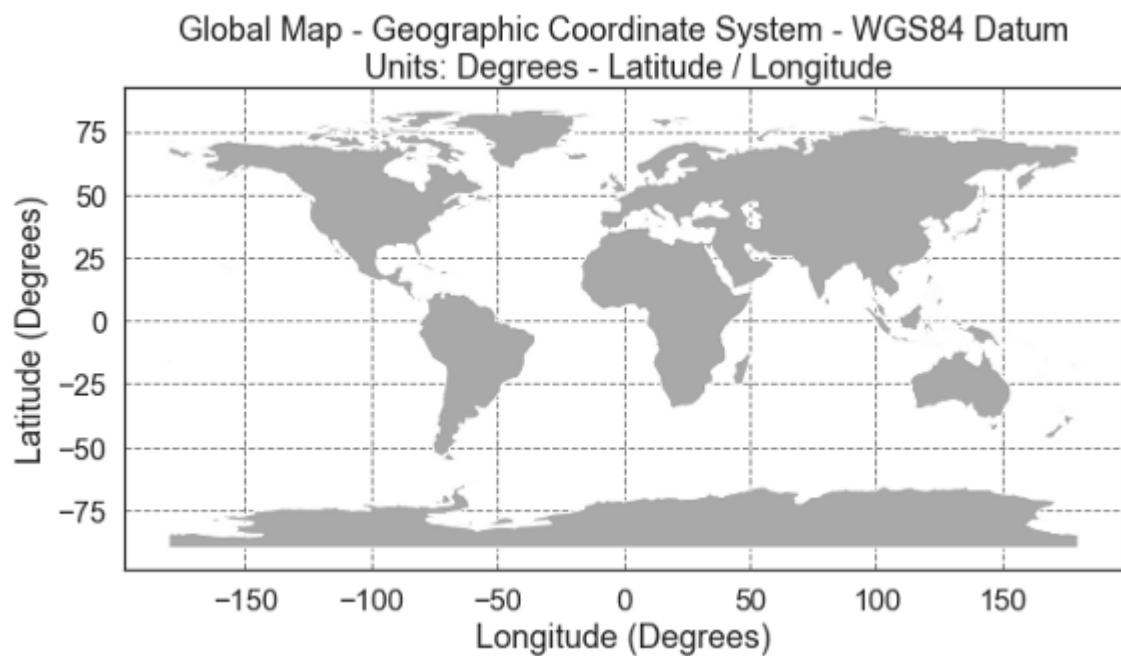
If we can reduce the size of these lists then the query performance could increase. Another approach to solving this problem more efficiently is by using the concept of 2D grids.

2. Grids

We can divide the entire world map into small squares. Considering the area of the earth is 500 Million square km, and having a fixed search radius is 10km. We will have **500M/10 = 50 Million squares** with a fixed grid size of 10km. With fixing the grid size to the query radius, we will only need to search within the grid and its neighboring 8 grids.

place	Grid Structure
place_id	
description	
latitude	
longitude	
created_by/updated_by	
created_at/updated_at	
grid_id	

Every place with location (latitude and longitude) will belong to a specific grid. Each grid will have a unique id which can be indexed and stored in the places table.



Source:

<https://www.earthdatascience.org/courses/use-data-open-source-python/intro-vector-data-python/spatial-data-vector-shapefiles/geographic-vs-projected-coordinate-reference-systems-python/>

Basic Search Flow:

As our grids are statically defined with search radius being equivalent to the grid size, therefore, we can find the grid id for any location and its neighboring 8 grids. To find all the places nearby for a given location with latitude and longitude(100, 85) within a radius of 10Km.

1. Get the **main grid id** and its **8 neighboring grid ids** :

Left: ([x1-10,x2-10], [y1,y2])
 Right: ([x1+10,x2+10], [y1,y2])
 Top: ([x1,x2], [y1-10,y2-10])
 Bottom: ([x1,x2], [y1+10,y2+10])
 Top-Left: ([x1-10,x2-10], [y1-10,y2-10])
 Top-Right: ([x1+10,x2+10], [y1-10,y2-10])
 Bottom-Left: ([x1-10,x2-10], [y1+10,y2+10])
 Bottom-Right: ([x1+10,x2+10], [y1+10,y2+10])

Execute the following query:

Select * from Places where Latitude between 100–10 and 100+10 and Longitude between 85–10 and 85+10 and GridID in (mainGridID, Left, Right, Top, Bottom, Top-Left, Top-Right, Bottom-Left, Bottom-Right)

Place Id	Latitude	Longitude
10001	98	87
10002	101	82
10003	104	85
10004	93	89

This will definitely reduce and improve the overall runtime of the query as the scope of the search query execution got reduced to just 9 grids.

Data Caching

We can make it faster by storing the grid's number and the list of its places in memory. As discussed above, we will be having **50 Million** grids and we can assume that each grid id will be of **6 bytes** and as with the mammoth scale of **400 Million** places, we can assume the **place_id** to be of **8 bytes**. Therefore the total memory required to cache **grid ids** and **place ids** would be 4 GB.

$$(50M * 6) + (400M * 8) \approx 4GB$$

Problems with this approach

1. This approach will execute slow for popular places like **Dam Square(Amsterdam)** or **Sentosa (Singapore)** and this can cause an imbalance in the grids where some grids will be densely populated and others will be sparsely populated with places like coastal regions or islands.

2. If at all we can dynamically adjust the grid sizes by keeping a threshold of a maximum number of places in a grid but this approach will not be a cakewalk and will furthermore add more implementation complexities.

Design a proximity server like Yelp Part — 2

2D Grid Approach

Considering the area of the earth is 500 Million square km, and having a fixed search radius is 10km. We will have **500M/10 = 50 Million squares** with a fixed grid size of 10km.

This approach had a complex problem of managing popular places like **Dam Square(Amsterdam)** or **Las Vegas(USA)** and can furthermore cause an imbalance in the grids where some grids will be densely populated and others will be sparsely populated with places like coastal regions or islands. We have also talked about the implementation complexities involved in creating dynamically adjusting grids.

Overview

In this part, we will be discussing how we can solve the problem of grid imbalance along with the final system design and handling some important core design requirements like Data Partitioning and Data Replication.

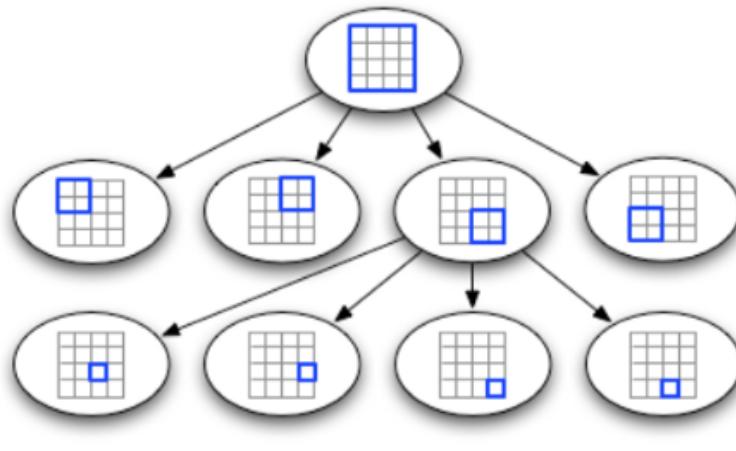
How can we solve the problem of grid imbalance?

This is the common challenge that **Tinder** and **Uber** also faced when they started growing and receiving a whale of traffic.

QuadTrees

A [quadtree](#) is a [tree data structure](#) in which each node has exactly zero or four children. Quadtree's peculiarity is the way it efficiently dividing a flat **2-Dimensional** space and storing data of places in its nodes.

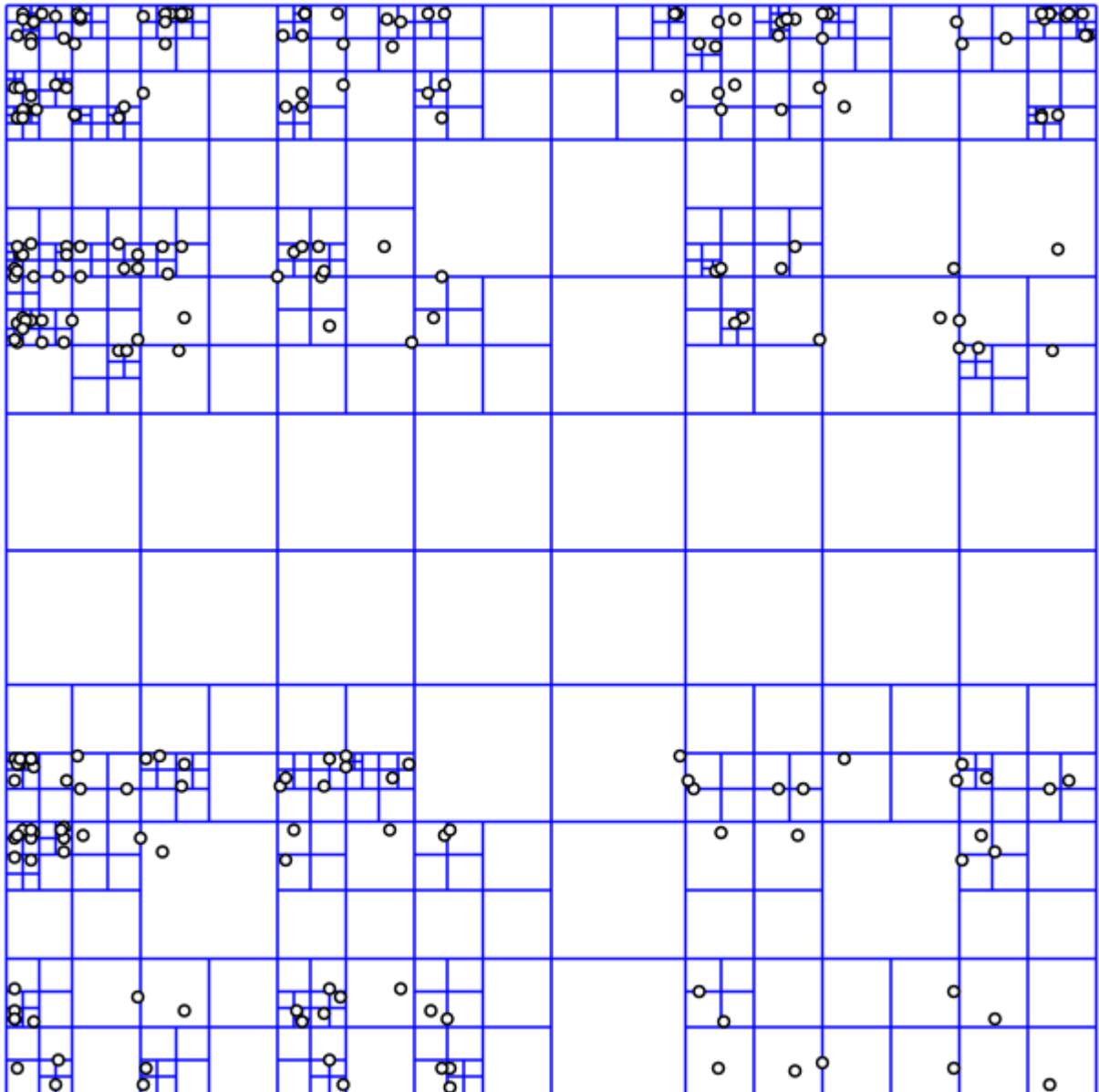
Considering our case, each node can represent a grid and can contain details about the places in that grid. If a node reaches the bucket size of **X**, then that node will be broken down into 4 child nodes and their data will be distributed into those nodes recursively.



QuadTree

Initialization

We will firstly keep all the places into one root node and as our 5 years scale is **400 Million**, the root node will not be able to hold all of them. The root node will then be recursively broken down into **4 child nodes** until no nodes are left with more than **500 locations**. Now we have our **QuadTree** constructed for 400 Million places with leaf nodes containing all the locations.



A quadtree representation with a bucket size of 1. Source: [Wikipedia](#)

Search Flow

We will start our search from the root node, then search downward till we find the required node. The required node will always be the leaf node as discussed above that places will be stored only in the leaf nodes.

Our Quadtrees construction algorithm always ensures that neighboring nodes will always contain geographically close places. Thus, for finding the nearby locations we will also be taking the neighboring nodes in the consideration.

To find all the places nearby for a given location with latitude and longitude(100, 85) within a radius of 10Km.

```
List<Places> places = getNearByPlaces(root, 100, 85, 10);
```

Data Caching

We can make it faster by caching the QuadTree details. As discussed above, we will be having **400M/500 = 0.8 Million nodes** in total. We can assume that the **node_id** will take **6 bytes** of size and each node can have 4 children pointers except the leaf ones. Apart from that, we also have **location_id**, **latitude**, and **longitude** of **8 bytes** each. Thus for storing everything, we will require:

$$(8+8+8) * 400 M + 0.8M*4 = 10 \text{ GB}$$

Data Partitioning

Considering the scale, we cannot rely on just one server for serving all the traffic as it can be a single point of failure and can adversely hamper the requirement of availability which is not acceptable these days when you have enormous power of distributed systems. QuadTrees should be partitioned.

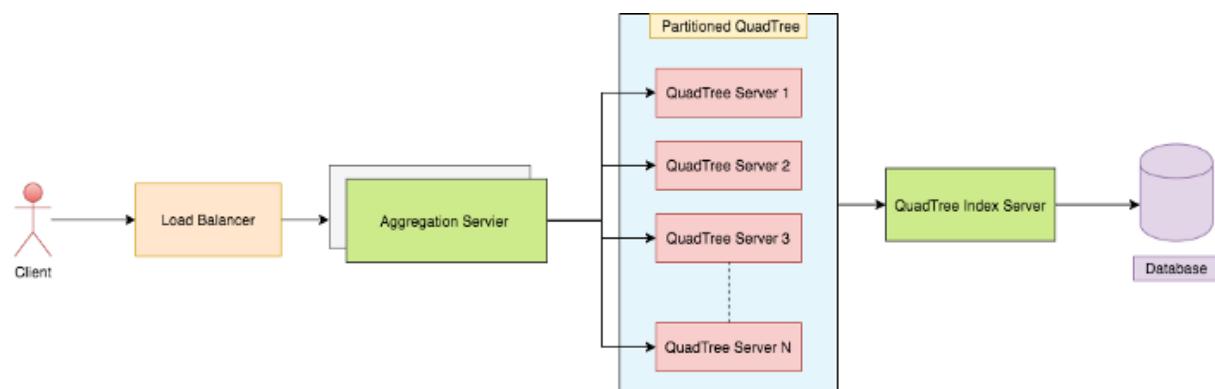
We can have multiple ways and algorithms to partition the data

1. **Regional based sharding:** The data can be partitioned on the basis of regions, but as discussed above this approach will result in a non-uniform data distribution because some locations are densely populated whereas others are sparsely populated. Therefore our uniform data distribution problem will not be solved.
2. **Sharding based on Place ID:** The data can be sharded on the basis of **place_id** by either hashing or consistent hashing. While constructing the Quadtree, we will iterate through all the places and calculate the hash of each **place_id** through our hash function. Our hash function will map each place id to a server where we will store details of that particular place.

The second approach looks simpler. Though we will end up having multiple QuadTrees which wouldn't be of that much concern as the uniform distribution of places is guaranteed.

Final System Design

To find nearby places, we have to query all servers and each server will return a set of nearby places. A centralized aggregation server will collaborate these results, sort them, and return to the user.



Data Replication

One of the most important core requirements of any mammoth scale distributed system is reliability which means the system should be able to work correctly even in the face of adversity. Therefore, in order to serve this, we cannot rely simply on just one machine and make that a single point of failure.

Master-Slave

The **master-Slave** architecture will suit our use case the most. Writes will happen only through masters and reads will happen through the slaves. Whenever a master server goes down, any one of the slave servers can take its position and become master and serve the writes. With this approach, a small delay of few milliseconds can be there in showing recently updated data causing eventual consistency but that should be fine as it's not a banking application.

What will happen if everything replica is down?

It's an extremely rare fault, but our system should be smart enough to deal with such cases. We can solve this by building an inverted index strategy based on **QuadTree Index Server** that can store the **QuadTree Server number** as the **key** and a **HashSet of place ids** that are present in the corresponding server as its values.

This approach will be less time-consuming. We should also have a replica of the QuadTree Index server for fault tolerance. If any QuadTree server dies, then that server can always rebuild by searching the **QuadTree index server** instead of querying and increasing database load.

Conclusion

We learned about how to design a proximity server like NearBy or Yelp, we discussed both the functional and non-functional requirements, calculated the user scale, figured out the mandatory APIs followed by creating database tables.

After that, we tried to design the actual system and incrementally solved the challenging problems attached to storing place details with their location:

1. Firstly, we tried to store place details along with its latitude and longitude in an SQL based storage system and also created an index on the location parameters. We carefully analyzed the downsides of this approach including inefficiency in the search query for a mammoth scale of 400 Million.
2. Secondly, we tried to solve the above problem by dividing the entire world map into a fixed-sized 2D grids and also found out that this approach is more efficient than the above. But the downside of this approach is the grid imbalance between densely and sparsely populated areas.
3. Finally, we solved the above problems by introducing a Quadtree data structure where each node has exactly zero or four children. Quadtrees efficiently divide the 2 Dimensional space into its structure and stores all the places detail in its nodes. Quadtrees are the best fit for our use case as they impeccably solve the problem of

grid imbalance and scalability. Quadtrees are heavily used by **Tinder** and **Uber** for solving their complex **geo-sharding and spatial problems**.

Along with the final system design, we also discussed some of the interesting problems related to Data Partitioning and Data replication.

16. Designing Facebook's Newsfeed

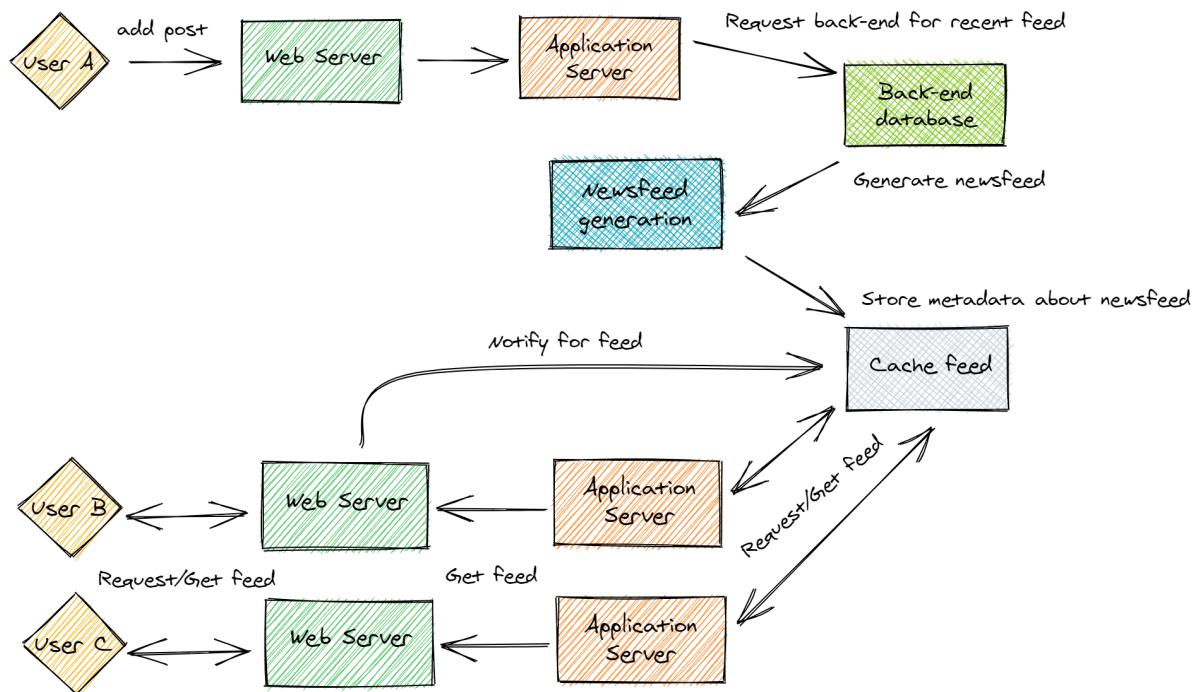
Solution 1:

The newsfeed updates Facebook users about all the activities and stories regarding friends, family, businesses, and news sources that you're connected to on Facebook. A key feature of the newsfeed is its ability to show content to the user according to **its relevance** towards each particular user.

We see that this newsfeed system is not particular to Facebook, but also to YouTube, Instagram, and Twitter-- all these applications are based on a similar newsfeed dashboard. Social media sites aim to engage users by displaying content and updates from other users, and the newsfeed system works very well for this purpose.

Overview of the System

Let's dig into the newsfeed architecture. It uses a ranking algorithm to determine what content to show. If we describe the system at a higher level, the chain of actions starts when the user adds or updates a post on Facebook.



This post is then received by the web server, which then sends it to Facebook application servers. These application servers coordinate with the back-end data of users to **generate a newsfeed**. The generated newsfeed is then sent for storage in a cache feed.

Now, whenever the user asks for a more recent feed, a feed notification is sent to servers, then the feed is retrieved by the servers and eventually sent to users that requested it.

System Design

The system design of a newsfeed system is unexpectedly quite simple. At the database level, the most important entity in the design of this system is the user, who will be assigned a unique ID along with all the necessary information that is required to create an account (such as your birthday, email, etc).

The other important entity in newsfeed design is the feed item. This feed item will also be assigned a unique feed (and feed_id), along with content and metadata attributes that should support images and videos. Each media item will also be an entity on its own.

There will be two main relationships modeled in this system, that would be a user - user relation and feed item - media relation. This is because users can be friends with or follow different users, and each feed item will correspond to different media sources.

Let's test your knowledge. Fill in the missing part by typing it in.

Newsfeed system can be represented as a _____ database.

Write the missing line below.

Let's now break down how a newsfeed is actually generated, published, and viewed by Facebook users.

Feed Generation

To generate feed for each specific user, the feed database is queried to fetch feed items of the user's friends and followers. These feed items are then sorted according to recency and relevance for the user, by being ranked using a ranking algorithm.

This feed generation process would however take quite a lot of time, especially when dealing with users with a large number of followers. Generating at the time of request (when a user gets on Facebook and scrolls) is a non-starter, as it would be too slow.

To improve this process, the feed can be pre-generated and stored in a cache memory. This pre-generated feed allows for the faster retrieval of feed items. It also allows for the generation of the feed for offline users or users with poor internet connection.

Feed Publishing

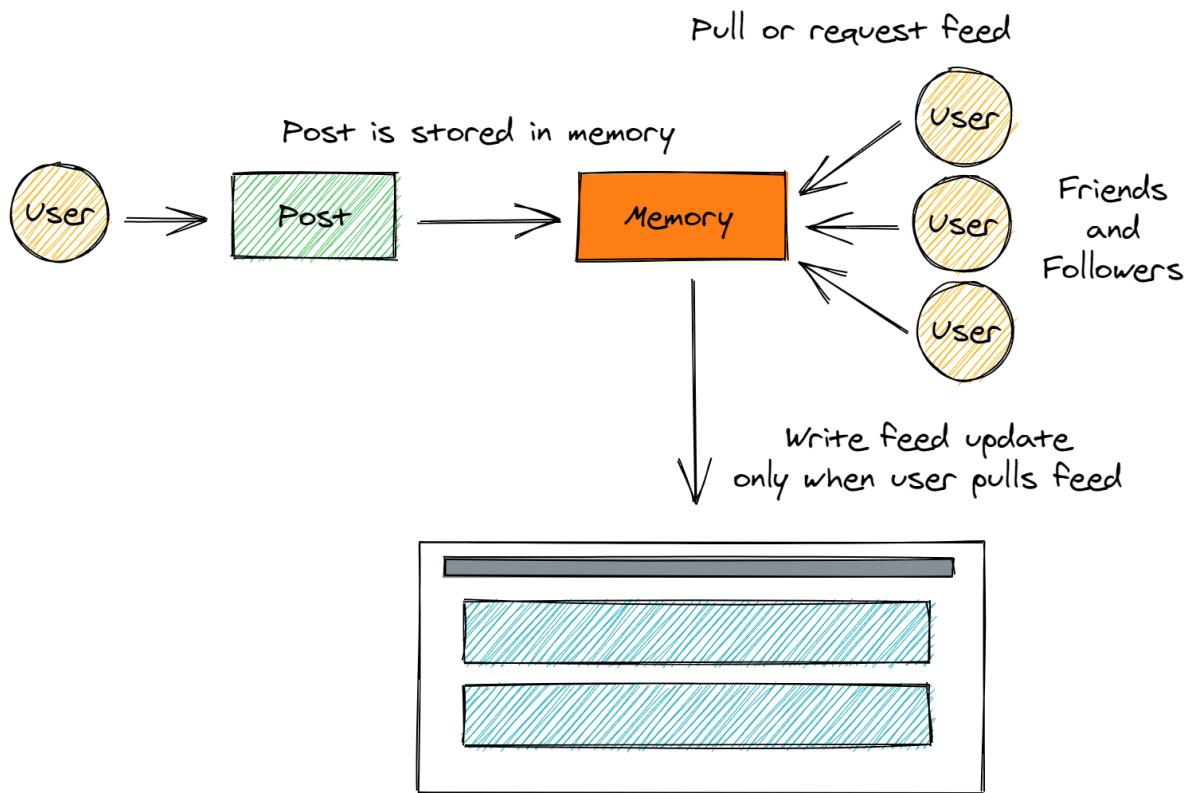
Feed publishing is the step where feed data is displayed according to each specific user. This can be a costly action, as the user may have a large number of friends and followers. To deal with this, feed publishing has three approaches:

1. push
2. pull
3. hybrid model

Pull Model or Fan-out-on-load

When a user creates a post, and a friend reloads their timeline, the feed is created and the pull model stores the generated feed in memory. The most recent feed is only loaded when the user requests a recent feed (that is, when they reload their newsfeed)-- that is, the computation of who this goes out to happens at the time of post creation. This approach reduces the amount of write operations from the system database.

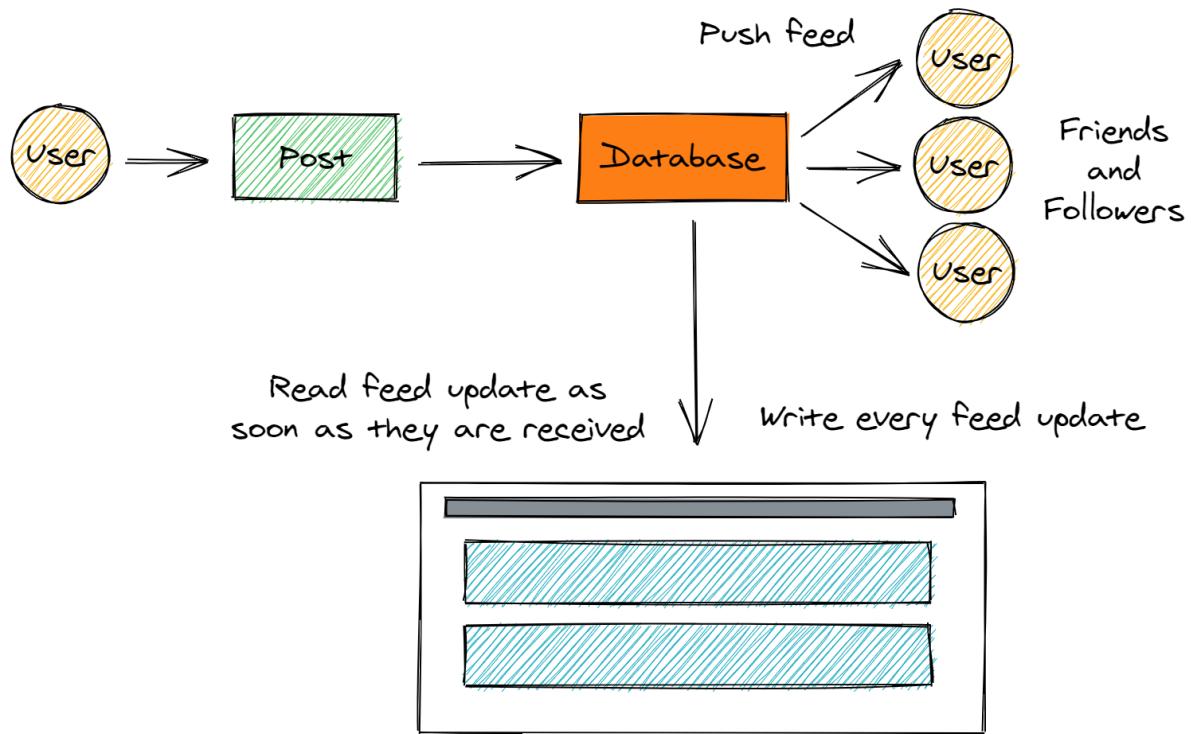
The downside of this approach is that the users will not be able to view recent feeds unless they issue a request to the server. Another problem could be the increase in read or load operations from the server, which may fail to load a user's newsfeed.



Push Model or Fan-out-on-write

Once a user creates a post, the push model pushes or sends this post to all the followers immediately. The work is all done at the time of write to figure out who will get this post. This prevents the system from having to go through a user's entire friends and follower list to check for updates on their posts published. Thus, the read operations by the system database is significantly reduced.

However, a prominent failure case of this approach occurs when a user has a large number of friends, as that would result in an increase in the number of write operations from the database.



Hybrid Model

A third approach is a hybrid approach between the pull and push model. It combines the beneficial features of the above two approaches and tries to provide a balanced approach between the two.

One method for achieving that would be by allowing only users with a lesser number of followers to use the push model. For users with a higher number of followers, a pull model will be used. This can result in saving a huge number of resources.

Ranking Newsfeed

After the feed is generated, each feed item is ranked according to the relevance for each specific user. Facebook utilizes an edge rank algorithm for ranking all the feed items in the newsfeed for a particular user. The edge in edge rank algorithm refers to every small activity on Facebook, such as posts, likes, shares, etc. The algorithm utilizes this feature of the network and ranks each edge connected to the user according to relevance. Edges with higher ranks will usually be displayed on top of the feed for the user.

The rank for each feed item in Facebook's edge rank algorithm is described by,

$$\text{Rank} = \text{Affinity} \times \text{Weight} \times \text{Decay}$$

Affinity is the "closeness" of the user to the creator of the edge. If a user frequently likes, comments, or messages the edge creator, then the value of affinity will be higher resulting in a higher rank for the post.

Weight is the value assigned according to each edge. Content with heavier weight would increase the rank. An example could be, a comment having higher weightage than likes, and thus a post with more comments is more likely to get a higher rank.

Decay is the measure of the creation of the edge. The older the edge, the lesser will be the value of decay and eventually the rank.

These factors combine to give a suitable rank to stories for each specific user. Once the posts are ranked, they are sent to memory or directly retrieved from servers to display on the newsfeed when the user requests it through the feed publishing process.

Try this exercise. Is this statement true or false?

Fan-out-on-load increases write operation when publishing newsfeed.

Press true if you believe the statement is correct, or false otherwise.

Solution 2:

All the social media sites have some sort of news feed system, like those in Facebook, Twitter, Instagram, Quora, and Medium.

News feed is a list of posts with text, image, or video generated by other entities in the system tailored for you to read. It's constantly updating while other entities creating new posts.

Requirement

Functional requirement

News feed is generated using the posts from other entities in the system that the user followed or the user might be interested in.

Posts might have text, image and video.

1. The new posts generated by others should be appended to the news feed of the user.

Non-functional requirement

News feed generation: should happen near-real-time. The latency seen by the end user should be just 1~2 seconds.

1. **Appending new post:** After a new post is sent to the system, it shouldn't take more than 5 seconds to be able to show up in a news feed request.

Capacity Estimation

During the fourth quarter of 2019, *Facebook* reported almost 1.66 billion daily active users (DAU). Overall, daily active users accounted for 66 percent of monthly active users. With

over 2.5 billion monthly active users, *Facebook* is the most popular social network worldwide.

The world population is just 7.8 billion as of March 2020. It means that 21% of world population are Facebook DAU and 32% are MAU. That's incredible.

To simplify computation, let's assume the system we are building have 1 billion DAU. Assume on average a user follows 500 users or entities on Facebook. An entity can be a group or a page.

Traffic Estimates

Assume on average one user fetches news feed 10 times per day. So it's $1e10$ requests per day and about 116K QPS.

Storage Estimates

Assume on average we keep 500 posts of each user's news feed in memory for fast fetch, and each post is 1KB in size. So 500 KB per user, 500 TB for all DAUs, and 5000 machines if each of them has 100GB memory.

System APIs

`getNewsFeed(userId, options?)`

`userId` (GUID): the id of the user who is fetching the news feed

The optional `options` parameter can contain the following fields:

`afterPostId` (GUID): fetch the news feed from after this post. If unspecified, fetch the newest posts.

`count` (number): the maximum number of posts returned for each request. If unspecified, some default maximum number is set by the backend.

`excludeReplies` (boolean): used to prevent the news feed from containing the replies.

The return values is a JSON containing a list of news feed items.

Database Design

Entities

User

Entity (page, group, etc): `entityId`, name, description, timestamp

Post: `postId`, title, text, `authorId`, timestamp

1. Media: `mediaId`, url, timestamp

Relationships

Follower-Followee: A User can follow other Users or Entities. (m:n)

Author-Post: Users and Entities can generate Posts. For simplicity, assume only Users can generate Posts. (1:n; we can embed the `authorId`)

1. Post-Media: Each Post has some associated Medias. (1:n)

High-level Design

Workflows

Feed generation

When Alice ask for her news feed, the system will:

Get followees: retrieve the IDs of all the users/entities that Alice follows

Aggregate posts: retrieve latest, most popular and relevant posts for those IDs.

Rank posts: rank the posts based on relevance and time.

Cache: cache the feeds generated and return the top 20 posts to Alice

1. Waterfall flow: When Alice reaches the end of those first 20 posts, another request is sent to fetch the next 20 posts.

Feed publishing (Live updates)

Assume Alice follows Bob, and Bob sends a new post. The system will need to update Alice's news feed:

Get followers: retrieve the IDs of Bob's followers

Add posts: Add the post Bob created to the news feed pool of those follower IDs.

Rank posts: rank the posts based on relevance and time.

Update Cache: update the ranked post into cache

Notify followers: let the follower know that there are new posts. (See

1.)

Components

Web servers: maintains connections with the users.

Application server: executes the workflows mentioned above.

Database and cache:

User/Entity: relational database

Post: relational database

Media (image/video): blob storage

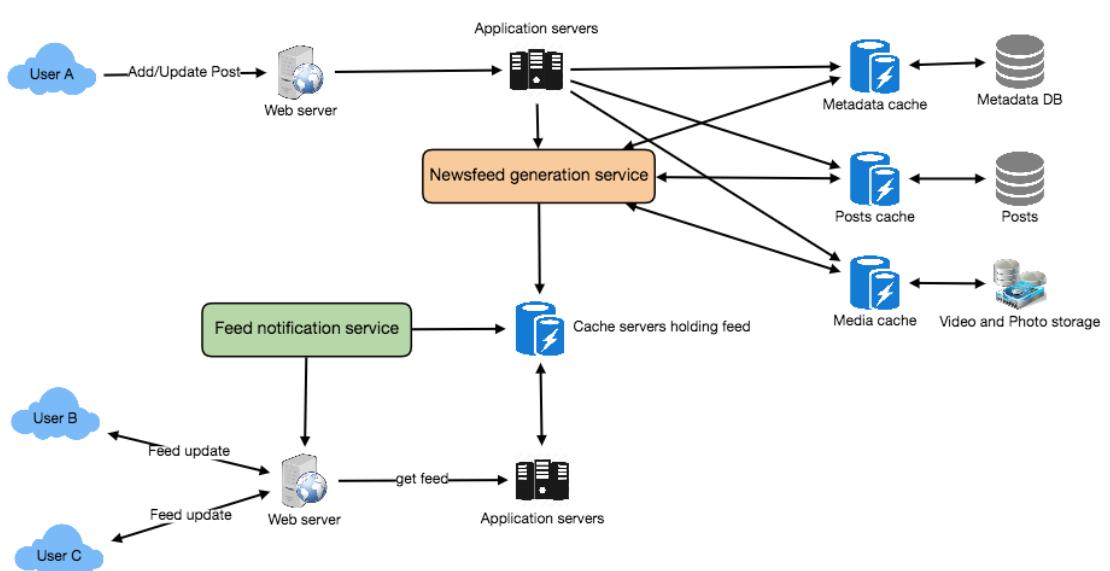
1. Metadata: relational database

Dedicated services:

Feed generation

1. Feed notification

Architecture



Detailed Design

Feed generation

Naive implementation (Fan-out read)

```
SELECT FeedItemID FROM FeedItem WHERE SourceID in
( SELECT EntityOrFriendID FROM UserFollow WHERE UserID = <current_user_id> )
ORDER BY CreationDate DESC
LIMIT 100
```

Issues of this naive implementation:

Super slow for users with a lot of friends/follows as we have to perform sorting/merging/ranking of a huge number of posts

We generate the timeline when a user loads their page. This could be quite slow and have high latency.

1. For live updates, each status update will result in feed updates for all followers. This could result in high backlogs in our Newsfeed Generation Service.

To improve the efficiency, we can pre-generate the timeline and store it in memory.

Offline Generation (Fan-out write)

We can have dedicated servers that are continuously generating users' newsfeed and storing them in memory. Whenever a user requests for the news feed, we can simply serve it from the pre-generated, stored location.

How many feed items should we store in memory for a user's feed?

Adjust on usage pattern.

Should we generate (and keep in memory) newsfeed for all users?

For users that don't login frequently.

Simple solution: LRU based cache.

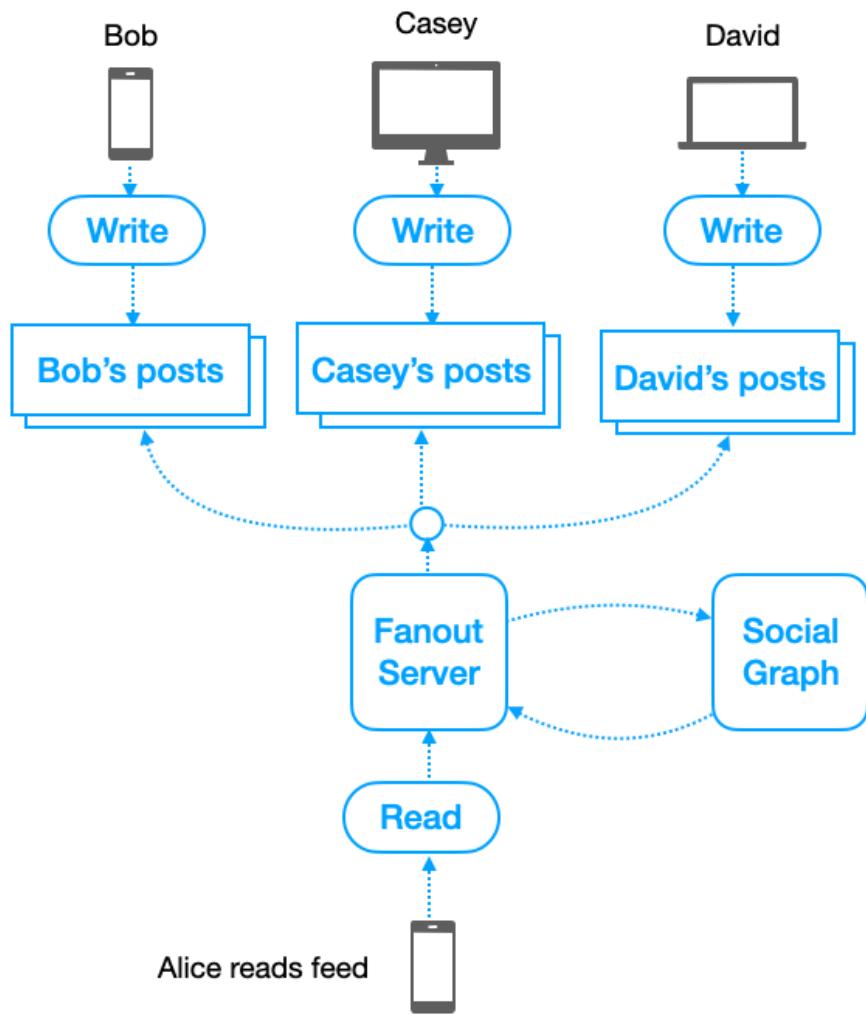
Smarter solution: learn the login pattern of users. What time? Which days of week?

Feed publishing

The process of pushing a post to all the followers is called a **fanout**.

fanout read (pull)

When you request for news feed, you creates a read request to the system. With fanout read, the read request is fanned out to all your followees to read their posts.



Fanout read

Pro:

The cost of write operation is low.

1. Easier to do different aggregation strategies when reading the data.

Con:

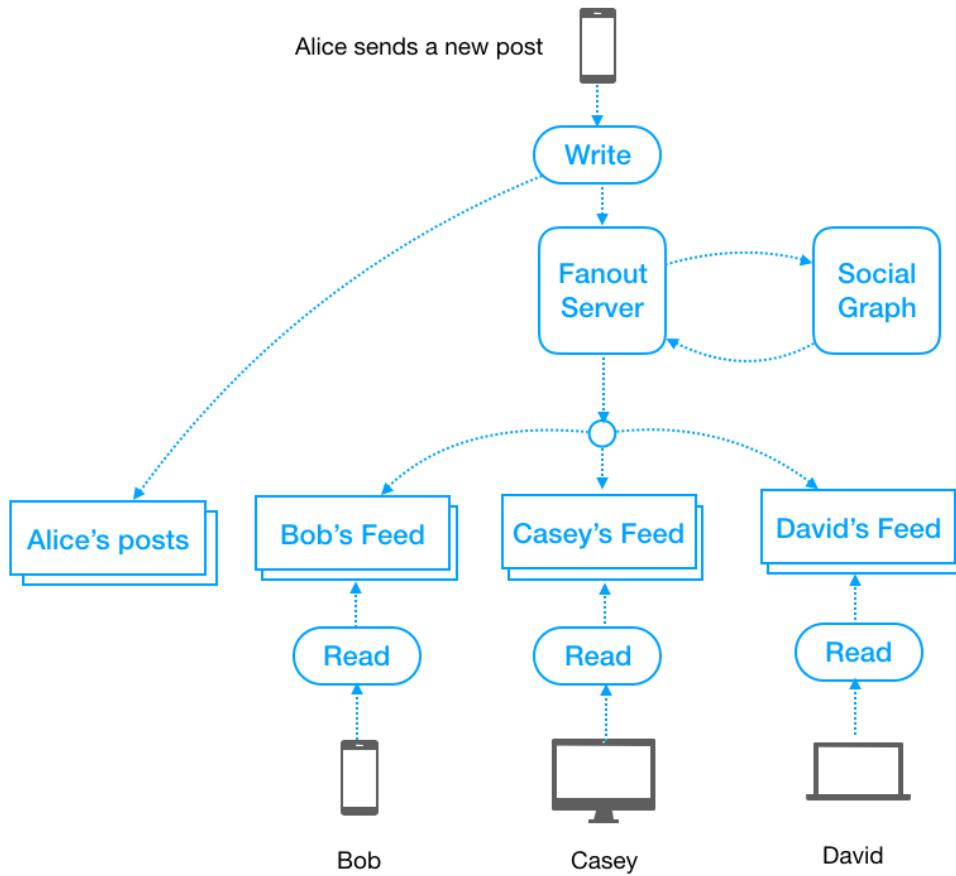
The read operation is super costly for a user who has lots of followees.
new data can't be shown to the users until they pull.

1. If we periodically pull to fetch latest posts, it's hard to find the right pull cadence and most of the pull requests will result in an empty response, causing waste of resources.

This architecture is better for write-intensive application.

fanout write (push)

When you send a new post, you creates a write request to the system. With fanout write, the write request is fanned out to all your followers to update their newsfeed.



Fanout write

Pro:

1. The cost of read operation is low.

Con:

1. The write operation is super costly for a user who has millions of followers.

This architecture is better for read-intensive application. Take twitter as example, its $\text{readRate} \gg \text{writeRate}$.

For systems have less latency requirement, we can use this approach as well. For example, WeChat Public Accounts do fanout write and all their followers get notified after some latency ranging from seconds to minutes.

Hybrid

Idea 1. For users who has lots of followers, stop fanout write for their new posts. Instead, the followers fanout read the celebrities' updates.

Idea 2. When users send new posts, limit the fanout write to only their online followers.

How many feed items can we return to the client in each request?

The backend should have some maximum limit. But it should be configurable by the client so that different client (mobile vs desktop) can have different limits.

Should we always notify users if there are new posts available for their newsfeed?

For mobile devices where data usage is relatively expensive, "Live Update" should be configurable by the client.

Feed Ranking

Instead of simply ranking the feeds chronologically, today's ranking algorithms also try to ensure that posts of higher relevance are ranked higher.

Select features that can determine the relevance of a feed item, e.g. number of likes, comments, shares, time of update, whether the post has images/videos, etc.

Compute the score based on the features.

- Rank the posts using the score.

Set up metrics like user stickiness, retention, ads revenue, etc. to determine whether our ranking algorithm are good.

Data Partitioning

Sharding posts and metadata

As we have more users and posts, we need to scale our system by distributing our data onto multiple machines such that we can read/write efficiently.

Sharding feed data

Since we only store a limited number of feeds in memory, we shouldn't distribute the feed data of one user onto multiple servers.

We can partition the user feed data based on userId. We hash the userId and map the hash to a cache server. We would need to use .

Solution 3:

Timeline/newsfeed is something which you see as soon as you open an app may it be Facebook, Twitter, Instagram, Quora, etc. If you are asked to design any one of these platforms, the interviewer might discuss newsfeed/timeline generation also alongside other features of the platform asked to design. Facebook's newsfeed is a scrollable version of life stories from your friends, connections, or pages which you follow. Facebook launched newsfeed in 2010 when it had 10 million users. Your **News Feed** is made up of stories from your friends, Pages you've chosen to follow, and groups you've joined. Facebook users spend up to 40% of their time in the News Feed.

Let's design the same.

Step 1: Use Case Discussion:

- **Interviewee:** Newsfeed will be made up of posts from user connections. Will posts contain media like photos and videos?
- **Interviewer:** Yes
- **Interviewee:** Do we also have to show the number of likes and comments for each post?
- **Interviewer:** Yes
- **Interviewee:** On what basis should we rank the posts?
- **Interviewer:** Recent posts must appear up. Newsfeed must be shown such that relevant or posts from close friends must appear up. I.e the newsfeed must be unique for each user.

- **Interviewee:** What is the scale of our application?
- **Interviewer:** Let's consider the current user base of Facebook (2020).

From the above discussion, we have jot down the following feature requirements-

- The news feed consists of posts from connections.
- Posts can contain photos and videos.
- Newsfeed must contain likes and comments for each post.
- Newsfeed must be shown such that relevant or posts from close friends must appear up. I.e the newsfeed must be unique for each user.
- Currently, as of 2020, the total Facebook user base is 1.5 billion. Let's say out of 1.5B users 500M are daily active users.

Tip: The point is that at the interview if you need to come up with such numbers it's good to have some background knowledge. Sometimes the interviewer might give the estimates, sometimes won't.

Step 2: Design Expectations

- We will be handling a large amount of data, hence partition tolerance is necessary. **Partition tolerance** means that the cluster must continue to work despite any network failure between system components/ nodes in the system. Users must not face any performance failure.
- Our service must be highly **available**.
- It is okay if we don't have strong consistency. Owing to the CAP theorem, as availability and partition tolerance is our priority, **eventual consistency** is fine. It is if we don't see the most recent and correct number of likes on a post for a while.
- The number of posts to be shown must be dependent on the screen size of the user
- Newsfeed must not end i.e it must be generated more and more if required.
- Our system must be resilient and reliable.

Step 3: System APIS:

- **get_timeline(user_id, user_screen_size, count, get_from)**
 - **user_id:** Unique user id of the person who is fetching the timeline.
 - **user_screen_size:** Attribute which defines the screen size which varies depending on the device on which newsfeed is being shown for example- mobile, tablet, laptop, desktop screen, etc. Thus, the number of posts to be fetched at once and the resolution with which it has to be fetched is determined based on screen size.
 - **count:** Number of posts to be fetched at once.
 - **get_from:** The id from and after which the next feed has to be obtained. For example: If the first 20 posts are fetched, for the next get_timeline() request, get_from will be 21, and if we are fetching 20 posts each time our count will be 20.

This API request will get a JSON response from the server which will be rendered to the client's screen.

Step 4: Traffic and Storage Estimations

Total users: 1.5bn

Daily active users: 500mn

- **Traffic estimations:**
- Assuming each daily active user opens app 4 times a day,
Total get_timeline() requests= $500M * 4 = 2B$ request per day= 24k requests per sec
Storage estimations:

Generating newsfeed at the time get_timeline() request is made will be a very expensive procedure. It is better if we precompute newsfeed and just render it whenever a request is received. Let's say we always keep the top 250 posts precomputed for each user. Though our total number of users is 1.5 B our DAU count is 500M. Generating newsfeed for all nonactive users would be a waste in storage. Hence we will keep pre-generated newsfeed for daily active users and if a nonactive user requests for newsfeed, it can be generated at that time request is made.

Let's assume we precompute 250 posts for each individual's newsfeed. Let's assume that of the total 250 posts, 1/5th contains videos and the rest contain photos.

Assuming the average photo size to be 200kb and video size to be 2mb,

Storage for one user= $250/5*2mb+250*4/5*200kb=100mb+40mb= 140mb$

Total storage= $500M * 120mb = 60PB$

Step 5: High-Level Design:

We are receiving thousands of requests per sec. Obviously one application server won't be able to serve all requests. Hence we need a load balancer in front of them. Media content like photos and videos will be obtained from object storage, and the metadata like user connections, posts, etc will be obtained from metadata databases. Alongside other microservices, as discussed in step 5 of [Facebook System Design](#) article, we need a Newsfeed Generation service.

What algorithm would you use for the load balancer to distribute the load?

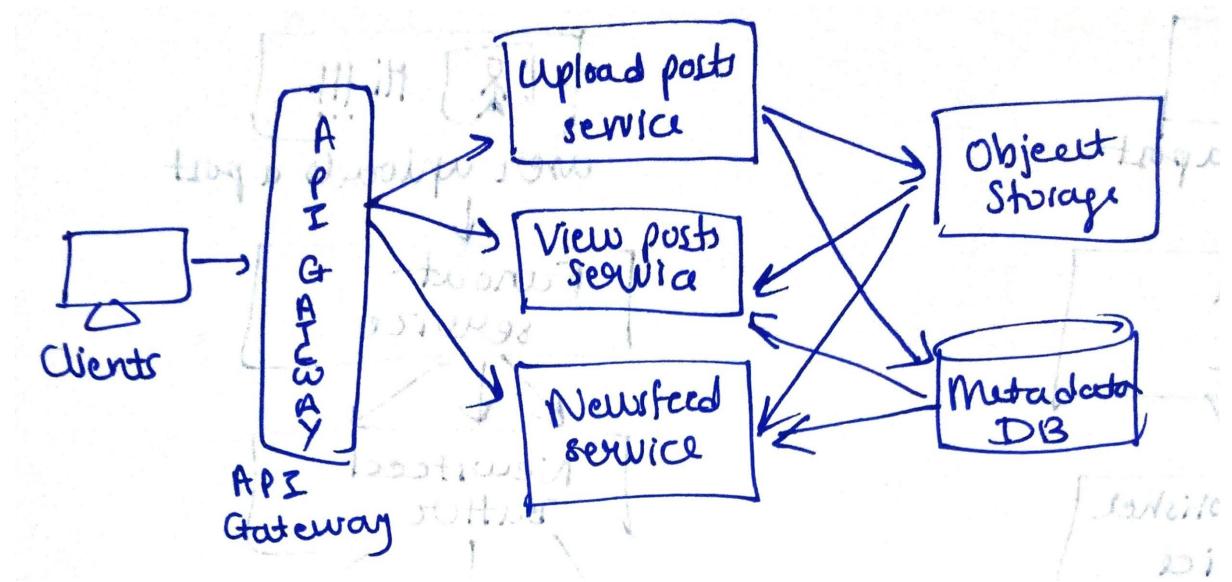
Consistent hashing is a perfect solution as it would take into account failing servers or the addition of new servers.

We add an API gateway as an entry point for the clients to forward requests to appropriate microservices. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

We would use SOAP/REST API to make API requests as discussed in system APIs.

Go through [Scalability basics](#) for more information.

Our High-level design looks like-

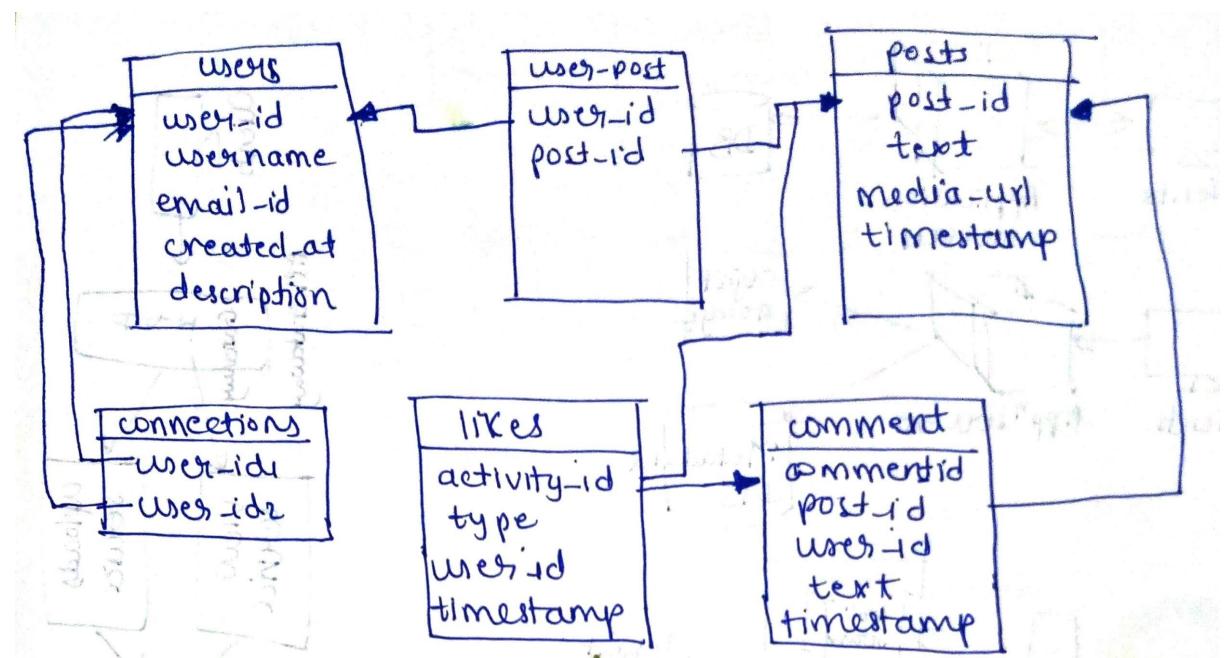


Step 6: ER diagram

We need to model two relations: user-feed relation and friend relation. The former is pretty straightforward. We can create a user-feed table that stores userID and corresponding postID. For a single user, it can contain multiple entries if the has published many feeds.

For friend relation, the adjacency list is one of the most common approaches. If we see all the users as nodes in a giant graph, edges that connect nodes denote friend relation. We can use the connections table that contains two userIDs in each entry to model the edge (friend relation). By doing this, most operations are quite convenient like fetch all friends of a user, check if two people are friends.

The ER diagram will look like-



In the design above, let's see what happens when we fetch feeds from all friends of a user.

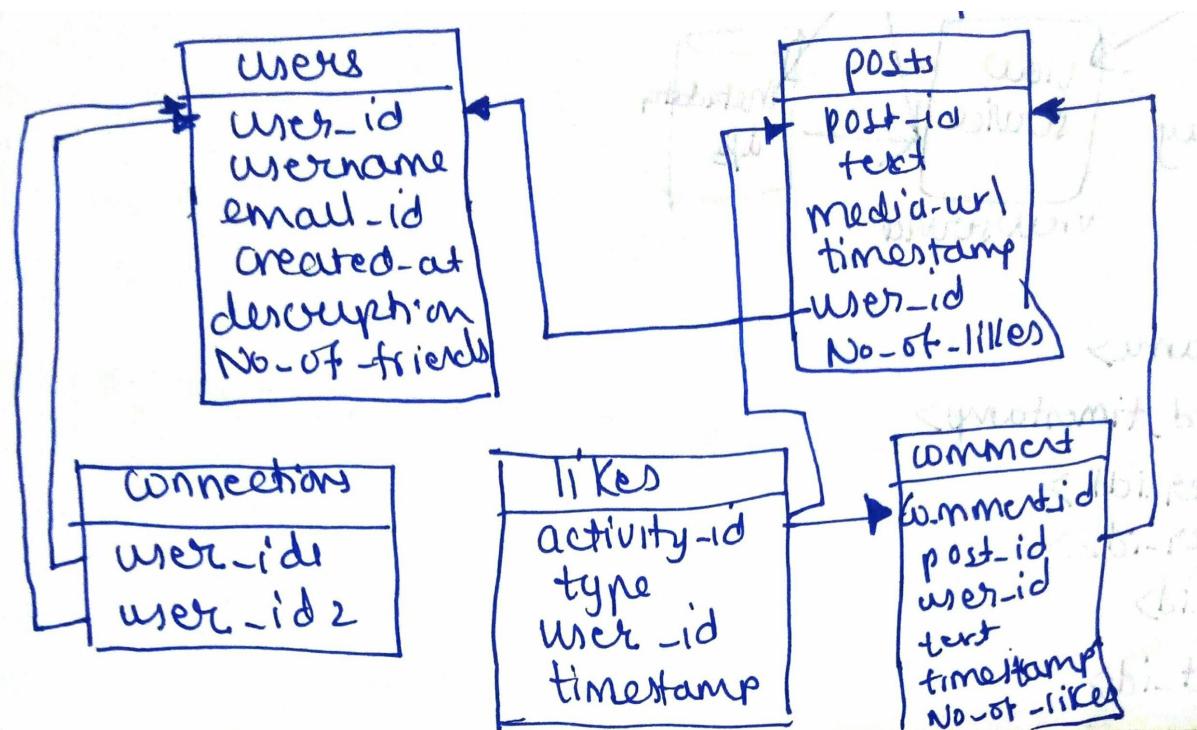
- The system will first get all userIDs of friends from the connections table.
- Then it fetches all post_IDs for each friend from the user_posts table.
- Feed content is fetched based on post_ID from the posts table.
- Finally it will rank it and then resent it to the user.

This process is a very long process and thus has high latency and may lose customers' interest in our size. If we see there are 3 joints involved.

Step 1: Denormalize Database

We can denormalize the tables such that the posts table also contains the user ids of the ones who posted it. This way it does introduce redundancy to the data but then we can obtain data faster.

So now our ER diagram looks like-



Step 2: Create index

users : <username>
posts : <user_id, timestamp>
connections : <user_id1>
 <user_id2>
likes : <activity_id>
comment : <post_id>

For detailed information on ER diagrams for Facebook, read step 6 of [Facebook System Design](#) article.

Step 7: SQL or NoSQL

Read step 7 of [Facebook System Design](#) article to get more information.

Let's say we use Cassandra in our system. As we are using Cassandra, the indexes which we already discussed, would be used as keys for consistent hashing.

Step 8: Low-level design:

- **Feed Generation**
- We are still generating news feeds for users when they request it. Instead of this what if we precompute newsfeed for each daily active user?

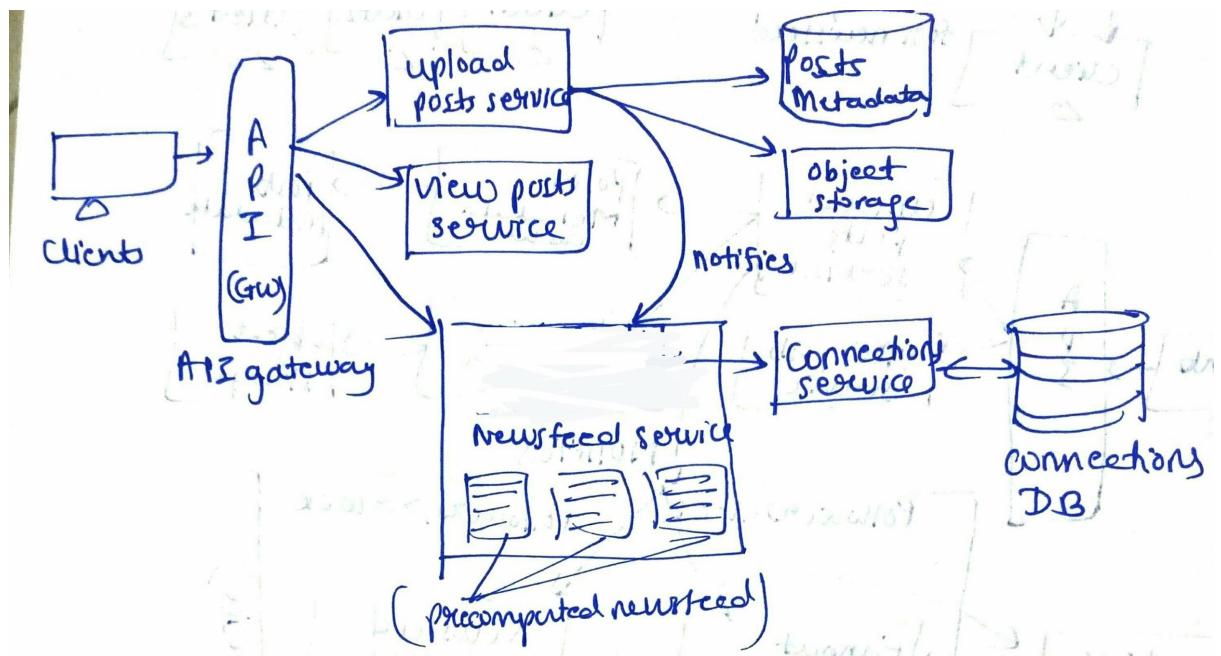
Let's say we want to compute a newsfeed for you. You would like to see the posts made by people who you follow.

Solution: Precompute the newsfeed

We can have dedicated servers that are continuously generating users' newsfeed even though the user is offline. So even though you have not opened your Facebook application, Facebook has already precomputed your newsfeed beforehand. If your friend uploads a post, your newsfeed will be updated with the addition of the post.

Let's call this service **Newsfeed service**. **Newsfeed service** will always be busy keeping your newsfeed up-to-date irrespective of whether you are online or offline. But if you are not an active user, this service won't be busy working for you :D

Also, create a service called **Posts service** which notifies the **Newsfeed service** if a new post is updated. We also add a **Connections** service which obtains the connections of the user who uploaded the post.



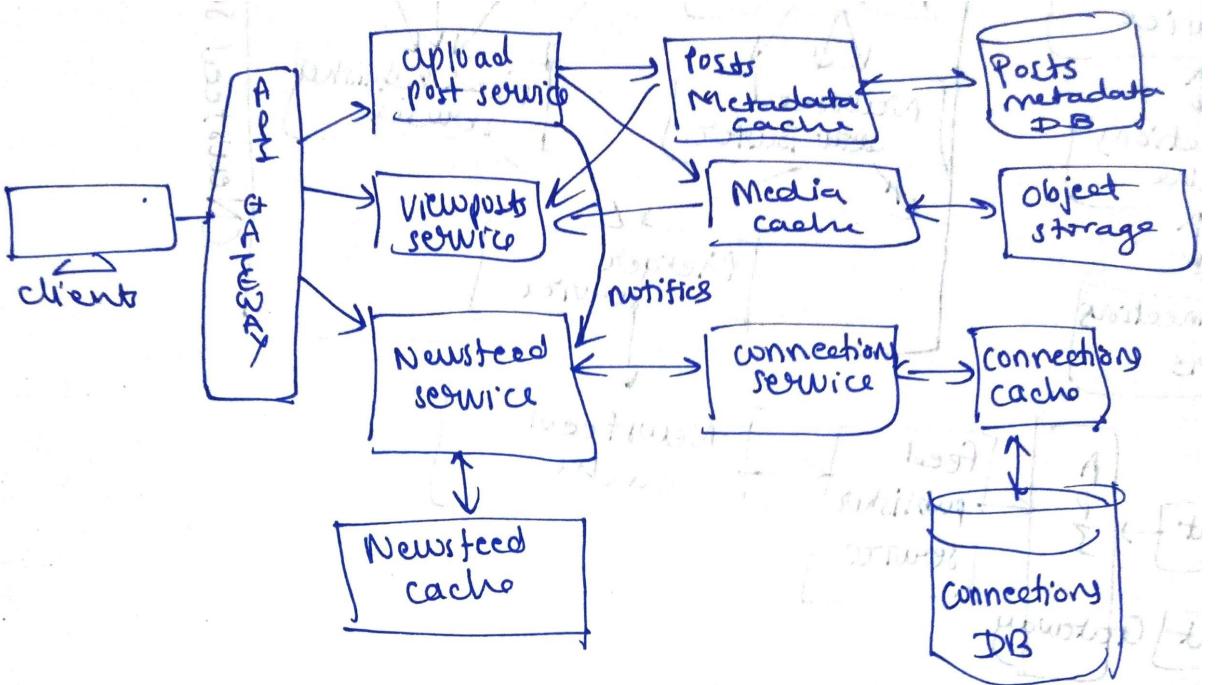
Thus whenever there is a new post, the following steps are followed:

- The **Posts service** updates its database with the post contents. i.e. the posts metadata is stored in the **posts-metadata database** and the media contents are stored in the object storage.
- **Posts service** notifies about the new post to **Newsfeed service**.
- **Newsfeed service** sends a request to the **Connections service** to obtain followers of the user who made the post.
- **Connections service** fetches all followers of the user_id and serves the result to the Newsfeed service.
- The **Newsfeed service** updates the newsfeed of the followers of the user who had uploaded the post.
- After precomputing the newsfeed, how are we planning to store the newsfeed? In the database or cache?

Solution: Use cache to have in-memory storage for newsfeed

Storing precomputed newsfeed in-memory would optimize the time required to serve `get_timeline()` request. Thus, we introduce an in-memory **Newsfeed cache** which will be updated by the **Newsfeed service** on the addition of new posts. LRU mechanism can be used to invalidate data in the cache. Thus, the newsfeed of users who don't access newsfeed frequently will be removed from the cache and thus won't occupy space unnecessarily. We will also add cache in front of metadata databases and object storage in order to avoid firing queries to the databases every time.

Now our system design looks like-



Redis is read-optimized and stores all data in memory. Hence Redis can be used here as in-memory cache.

Keep different types of caches-

- **Short term cache** . A timeline of recent activity is frequently invalidated because it is changing all the time as you perform actions through your life.
- **Long term cache** . A query cache is kept in Memcached. The results of large queries, like the ranking of all your activities in 2010, can be efficiently cached since they will rarely be invalidated.
- **Which data structure would you use to store newsfeed in the cache?** Linked hashmap would be a good idea. You can have detailed information about linked hashmap [here](#).

● **Publishing Newsfeed:**

- Let's add another service called **Feedpublisher service** to serve precomputed newsfeed when requested by a user. **Feedpublisher service** will work on fetching the precomputed newsfeed from the **Newsfeed cache** and make it available to the client as a result.

Whenever a user uploads a new post, we need to update the newsfeed of every connection this user has. As we know, **Newsfeed service** works on updating the newsfeed of every connection of the user uploading the post and stores the updated newsfeed in the **Newsfeed cache** , Lets see how this service works internally.

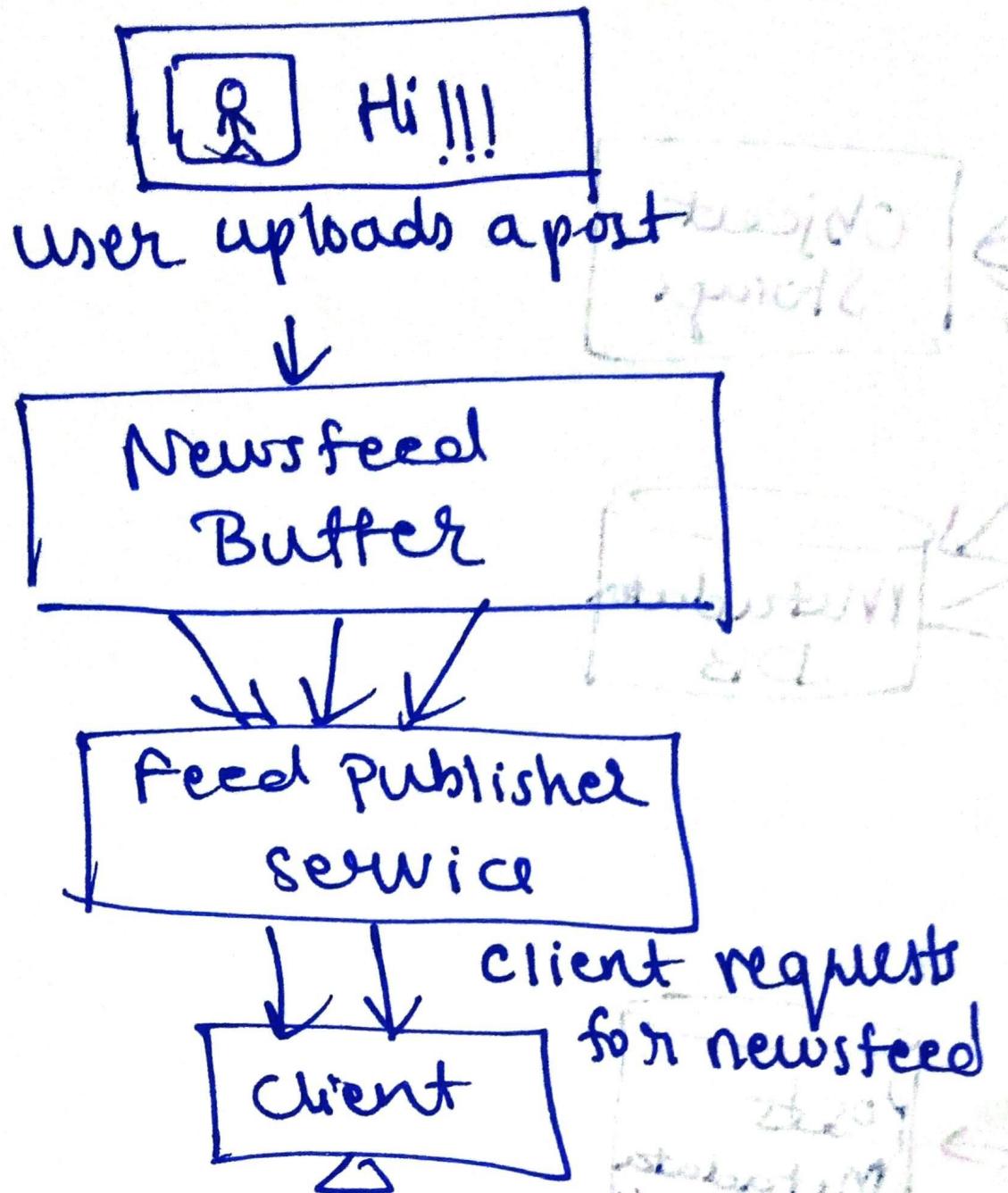
Let's say A uploads a new post and B is A's friend. I.e connection. So we have a Connections service that has this connection stored in its database.

Option 1: **Pull model / Fan out on load -**

New posts are added into the newsfeed when the connection requests for the newsfeed.

Here, when A uploads a new post, the newsfeed of every connection of A is **NOT** updated with this new post until they request for it. Let's say B is a connection of A. A's new post will be added to B's timeline, when B requests for its newsfeed. Until

then a list of all posts which have to be added in the B's old cached newsfeed is maintained. Let's call this **Newsfeed buffer**.



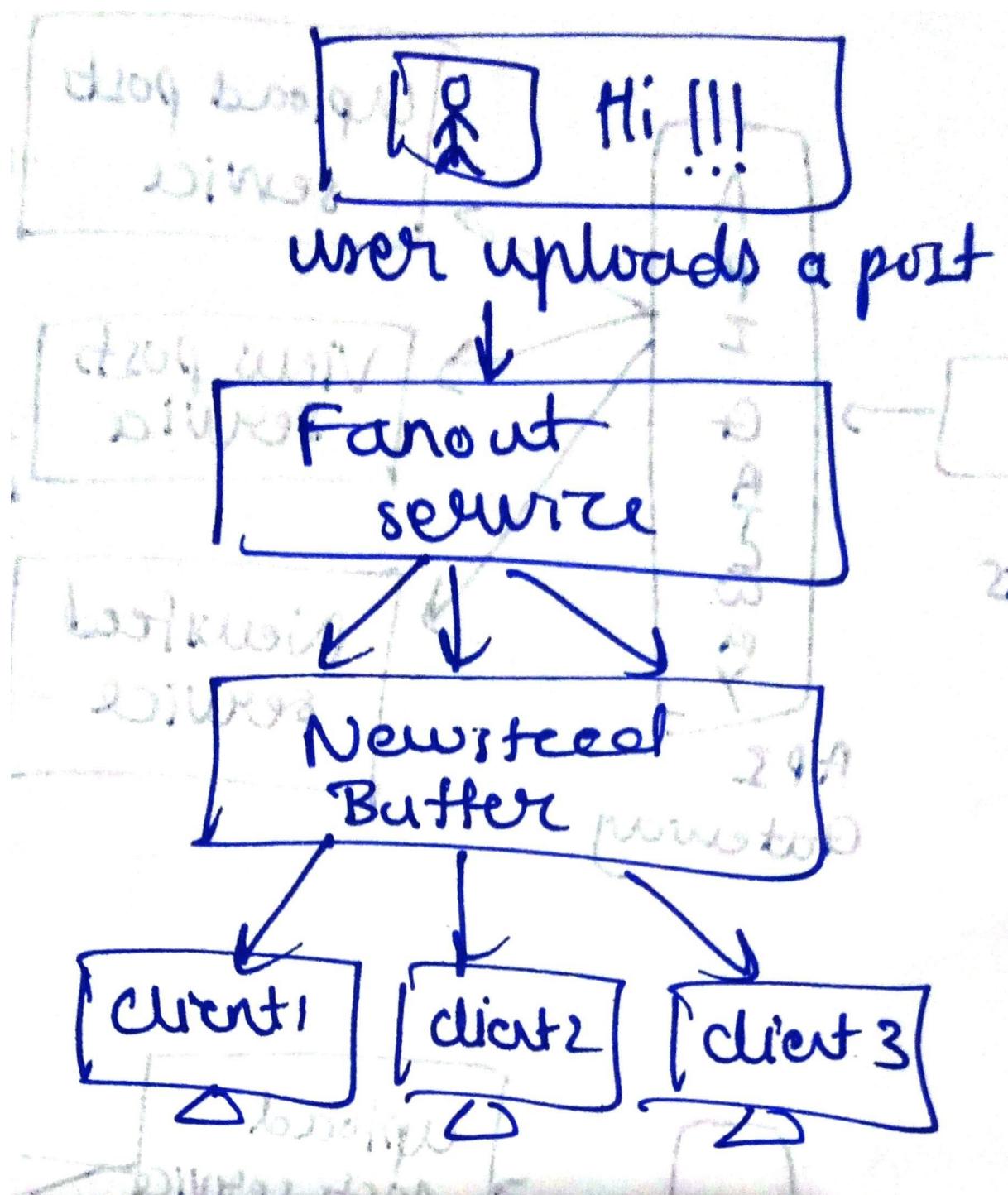
Problem:

- New data might not be available until user requests for it.
- Many times, there would be no new update in the newsfeed and an empty response will be served by our servers. This way our resources will be used just to send an empty response.

- Option 2: Push model / Fan out on write :

This way the server pushes the newly uploaded post to every connection's newsfeed and doesn't wait for users(connections) to request for it. Thus the precomputed newsfeed is always kept updated. Hence if a new post is uploaded, it is incorporated in the newsfeed of every connection. Let's say we use a **Fanout service** to fan out the post to every connection once it is uploaded.

Here, when A uploads a new post, the **Fanout service** will push this post to the newsfeed of every A' connection.

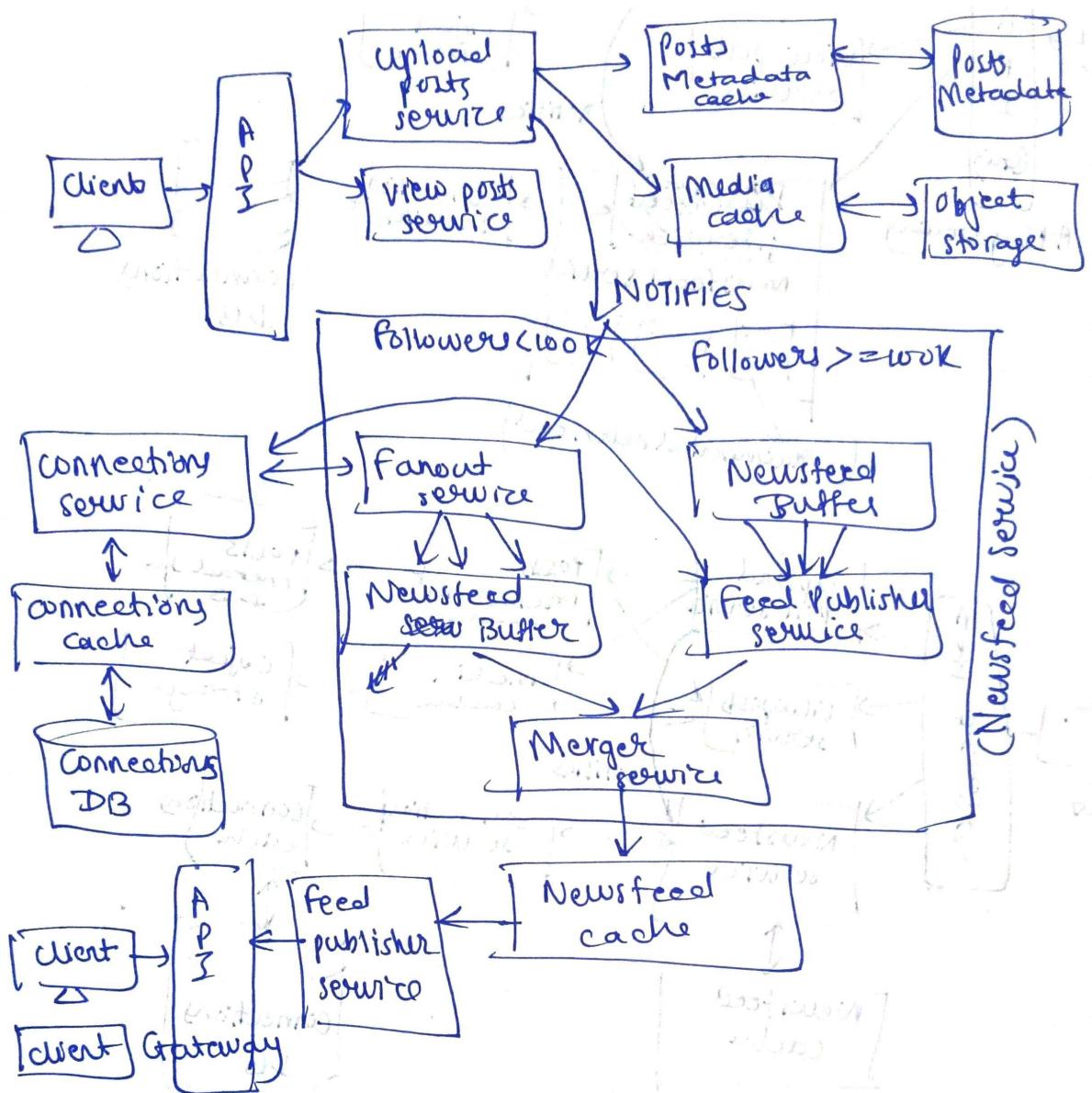


Problem:

Some users have millions of followers. Pushing to every one of these millions of followers will keep the server busy to a great extent.

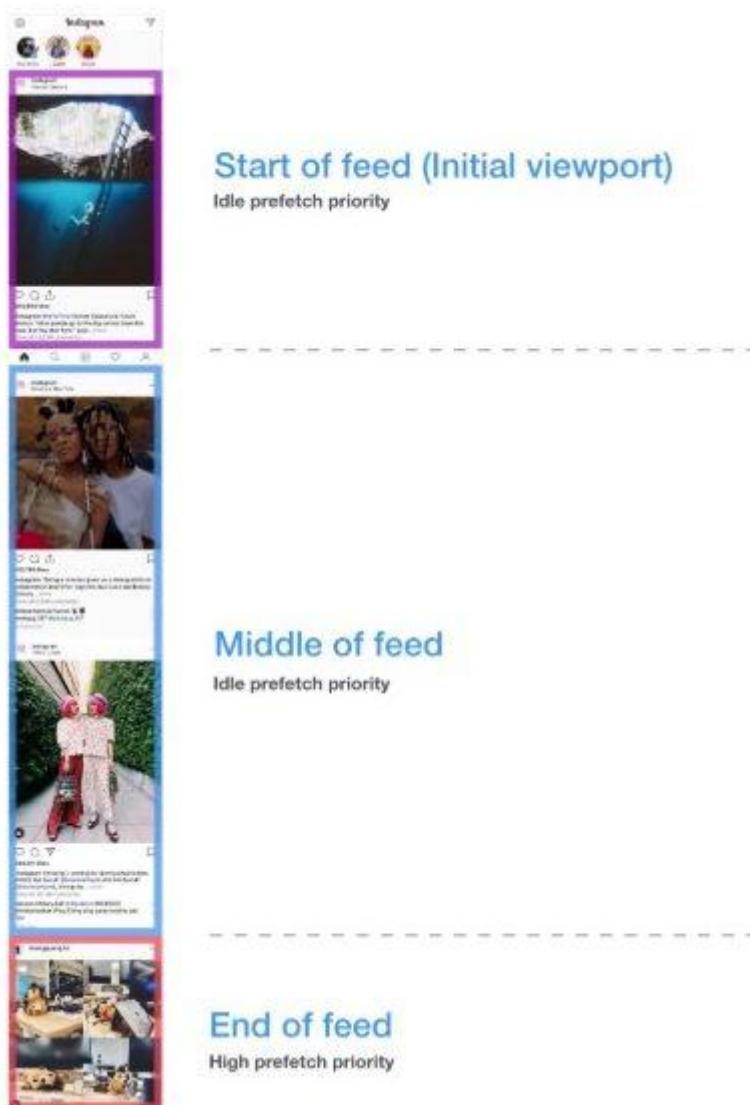
Eureka!!! let's combine these two models:

Option 3: Hybrid Model : If users have comparatively less number of followers, we can use the push model. Pushing to less number of followers is not a problem. While if a given user with millions of followers uploads a new post, we can use a pull model for them. That is, followers of these users will be updated on the new post whenever the followers requests for the newsfeed. This way the server won't be busy sending the new post to all the millions of followers. Pushing to users with less number of followers is affordable.



How do we provide an infinite scrolling feed of posts?

- One of the main surfaces on social networking platforms is the Feed consisting of an infinite scrolling feed of posts. We can implement this by loading an initial batch of posts and then loading additional batches as the user scrolls down the feed. However, we don't want the user to wait every time they get to the bottom of the feed (while we load a new batch of posts), so it's very important for the user experience that we load in new batches before the user hits the end of their current feed. Thus not all pre-generated 250 posts are sent from the cache at once. Let's say the `get_newsfeed()` function first sends the top 20 posts from a user's newsfeed. If the user reaches the end of these first 20 posts, let's say the user is on the 15th post, the next slot of the next 20 posts is fetched from the cache. This way in the background the next posts are kept ready. This way the user won't experience high latency while receiving a continuous news feed.



Problems with prefetch- (deep concepts)

This is quite tricky to do in practice for a few reasons:

- **Problem 1** - We don't want off-screen batch feed loading to take CPU and bandwidth priority away from parts of the feed the user is currently viewing.

- **Problem 2** - We don't want to waste user bandwidth by being over-eager with preloading posts the user might not even bother scrolling down to see, but on the other hand if we don't preload enough, the user will frequently hit the end of the feed.
- **Problem 3** - Our website must be designed to work on a variety of screen sizes and devices, so we display feed images using the `user_screen_size` attribute (which lets the browser decide which resolution of the images and videos in the posts to use based on the user's screen size). This means it's not easy to determine which image resolution we should preload & risks preloading images the browser will never use.
-

Solution for problem 1 and 2: Prioritize prefetch when required

Thus preloading of the next batch of posts is an asynchronous task. The pre-generated news feed is stored in the form of queues in the cache. A queue for every user's news feed. The queue consists of the metadata of posts to be displayed in JSON format.

We can build a prioritized task that handles queueing of asynchronous work of prefetching the next batch offered posts. Thus this task of prefetching won't begin if the browser is busy with some other work.

However, if the user scrolls close enough to the end of the current feed, the priority of this prefetch task is increased or made 'high' by canceling and thus firing off the prefetch immediately.

Once the JSON data for the next batch of posts arrives, we can queue a sequential background prefetch of all the images in that preloaded batch of posts. The prefetch is preferred to be done sequentially in the order the posts are displayed in the feed rather than in parallel so that we can prioritize the download and display of images in posts closest to the user's viewport.

Solution for problem 3: Using `user_screen_size` attribute

In order to prefetch the correct number of posts according to the user's screen size, we have a `user_screen_size` attribute.

Thus combined effect of-

- denormalizing database
- precomputing the newsfeed
- storing in the cache
- fetching posts asynchronously by altering the priority of prefetching whenever the user reaches end of the newsfeed
- Using screen size to decide how many posts have to be prefetched
- Will improve the performance of serving newsfeed to a great extent.

- **Data Partitioning:**

- **Sharding posts, users, user-friend relationship, likes, comments metadata:**
 - A procedure similar to the one used in step 8 sharding part of [Facebook System Design](#) article can be used. **Sharding feed data:**
- We want the entire feed of a user in the same shard. So that while providing newsfeed for the user we don't have to collect it from different shards. Thus sharding based on user id is the solution. If we have 100 shards, then we can do $\text{userid} \% 100$ to get the shard number in which data has to be stored. In order to make it adaptable to failing servers or in case of the addition of new shards, we can use a consistent hashing method to shard the newsfeed data.
- **Feed Ranking:**

Please refer to [this](#) article to understand the feed ranking procedure.

Step 9: Add redundancy:

What if this shard fails or catches fire? Can we afford to lose data of these users whose data was stored in this shard? No.

As we are using Cassandra, the data will be replicated across more than one shard. Hence, problems will be caused if all the shards with the same replica fail! which is almost impossible. Also, these shards can be placed in different geographical regions. Hence if a warehouse in the US catches fire, data can be obtained from the warehouse in Asia.

Instead of one load balancer, we show more load balancers, more cache, and more than 1 object storage instances. We can use Active-passive or active-active relationships to keep make our system Resilient. Basically, in order to make our system reliable, we cannot rely on only one device to do any task. We must have an additional same device may it be a load balancer, caches, etc. So if one fails, the other one can start to work instead of the failed device.

How do we know if a device is working? We can use heartbeat mechanism to know if a device is up and running all the time. a **heartbeat** is a periodic signal generated by hardware or software to indicate normal operation or to synchronize other parts of a computer **system**. Usually, a **heartbeat** is sent between machines at a regular interval in the order of seconds; a **heartbeat** message.

Solution 4:

A News Feed is a collection of the trending stories on the homepage of the web application. It can include status updates, posts, comments, likes, photos, and videos from the user's followees on the application.

Basic Design Requirements

System design interview questions are almost always open ended, so there's never a standard answer. Ask your interviewer a set of questions until you have the key features, around which you can formulate an answer.

During a 45-minutes interview, you cannot be expected to produce a high level architecture close to the one Twitter actually uses, but the basics need to be covered well. So here are the key features a basic News Feed architecture should involve:

- Users should be able to see posts, statuses and app activities from other users, pages and groups they have followed.
- Users can 'like' and comment on posts.
- The Feeds can be text, images or videos.
- The News Feed should automatically update with the latest/trending posts.

Design Goals For A Scalable System

- The system should be scalable to 300M active users.
- Users can have millions of followers.
- Assume 6k new tweet requests per second (writes) and 600k requests made by users for fetching their timeline each second (read) on average.
- A user can see their customized News Feed in real time with minimum latency of no more than 2 seconds.
- User's posts should be available to millions of followers with low latency of no more than 5 seconds.
- Up to 800 feeds can appear on the user's timeline.

Three Types Of Timelines

In applications like Facebook and Twitter, there are three different types of timelines:

Home Timeline

This one includes the tweets from all the pages, groups and people that the user has followed.

User Timeline

This timeline displays all the tweets you have made. It appears on your profile page.

Search Timeline

When the user searches a keyword, all the tweets that are related to that particular keyword appear on a timeline. This is the search timeline.

Twitter Characteristics

High Availability

From the estimates made under the design goals you see that while there are 6k write requests each seconds, there are 600k read requests per second. The write:read ratio here is 1:100. Since the News Feed application is read-heavy, we need a system that is highly available and scalable for the high number of read requests.

An eventual consistency model is a good approach in this case because it is acceptable for the users to see posts from their followers slightly delayed but availability needs to be a priority.

Storage

As for the storage, if we assume a size of 1KB for each post and 800 posts appear on each user's timeline, the system will need to allocate 800 KB storage capacity for each user.

$$1\text{KB} * 800 = 800 \text{ KB}$$

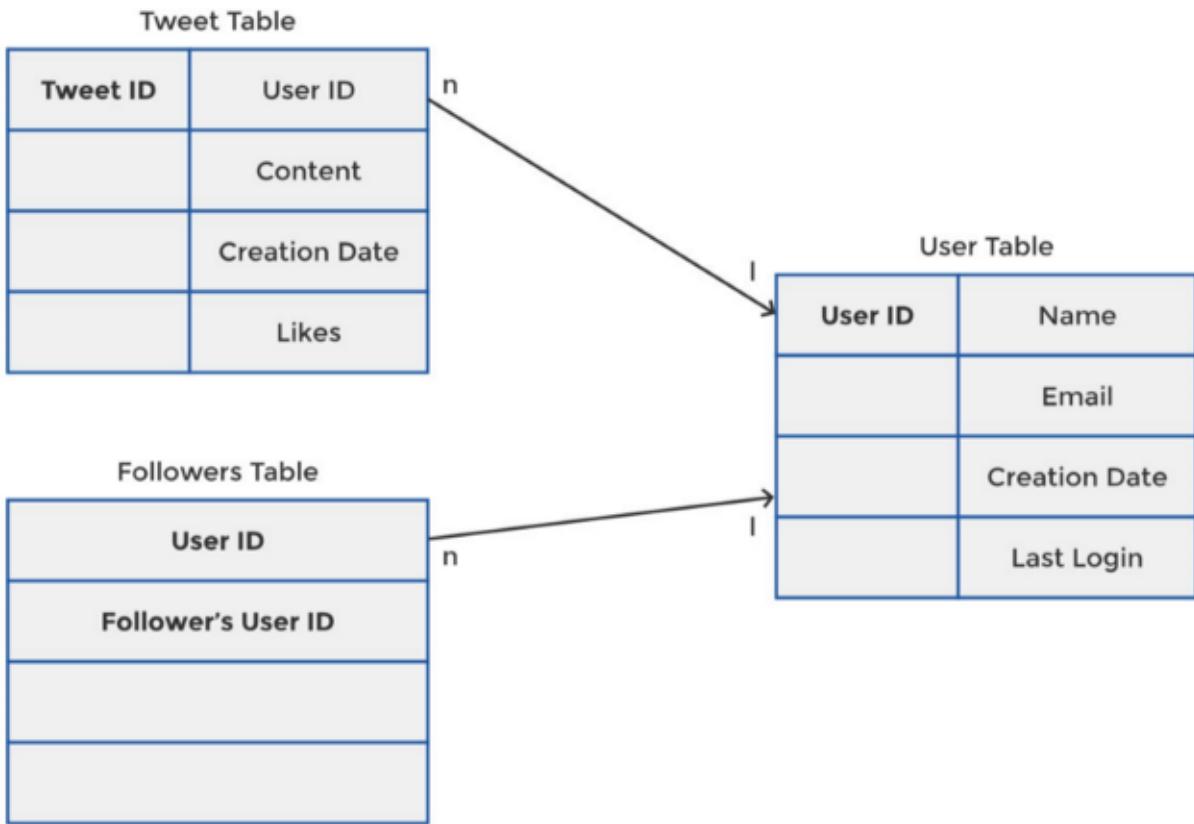
Now since twitter has 300M+ active users, a total storage of at least 240TB capacity is required.

$$800 \text{ KB} * 300 \text{ M} = 240 \text{ TB}$$

Since a single server will not accommodate the entire storage and ensure high availability at the same time, you need to design an approach to scale the application across a framework of servers.

Basic Database Architecture

Since we need a system that's horizontally scalable and can accommodate the heavy read traffic, Redis is the most fitting choice for storing data. However, you will also need to maintain a copy of the Tweets and user data in the database.



A basic database architecture for twitter will include three tables:

User table

The User table will have entries for different users. Each new profile will be appended in this table.

Tweet table

The Tweet table will have entries for all the tweets created globally. Each entry will have a unique Tweet ID, content and the user ID that points to the entry in the user table for that specific user. Each user ID can have multiple entries in the Tweet table (User table has a one-to-many relationship with the Tweet table).

Followers table

To create a user's home timeline, the system needs to know the entities (people, pages and groups) that the user is following so a Followers table will also be maintained. Whenever a user follows an entity, it is appended to the Followers table, where there can be multiple entries for each user ID (User table has one-to-many relationship with the Followers table).

How To Display User Timeline

Redis will carry the cached data for simpler and faster access of all the Tweets made by a particular User ID to display entries on the user timeline, in chronological order, as soon as it's requested. Retweets are also saved as Tweets, pointing to the original Tweet. The

system will not have to scan the entire database each time the user timeline needs to be displayed.

How To Display Home Timeline — Basic Approach

To create the home timeline, the system will map the Followers table to the Tweet table to pick the tweets from the followers and display them in chronological order. Of course, this is the basic solution. The bottleneck here would be that the Tweet table can become very big and mapping the followers onto the Tweet table will take a very long time.

Since the user should be able to access the timeline within seconds, a delay isn't acceptable. On Facebook, Twitter and similar apps, it barely takes 5 seconds to display your home timeline once you request it. So, moving towards the high level system architecture, you'll need a different approach rather than mapping Followers onto Tweets. This technique is called **fanout**, discussed in the next section.

Fanout For Faster Fetching Of Home Timeline

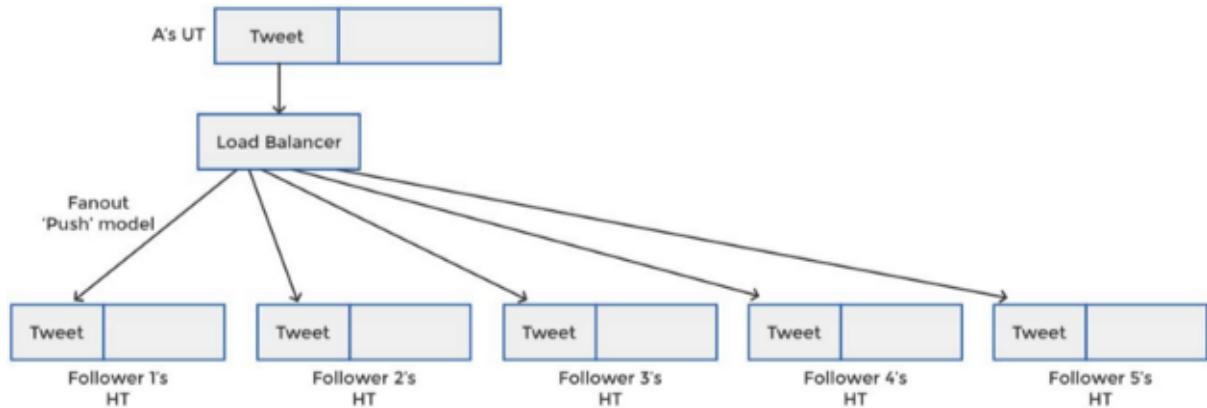
Fanout is the technique whereby a post is spread out to all the followers. Twitter will have several data centers operating in several regions across the globe. Whenever a new tweet is generated by a user, it's pushed towards a central point close to your IP address from where it can be sent to all the followers. This point from where the tweet is distributed to different servers and data centers is called the load balancer.

Now, there are two types of fanout approach, depending on the number of followers of a user.

The Push Model

Say, for example, a user A has 5 followers. When A makes a Tweet, it's pushed to the load balancer and from there it's pushed to the in-memory home timelines maintained in the individual cache of all the 5 followers. Redis maintains a home timeline for each user which contains the latest Tweet data from all their followees.

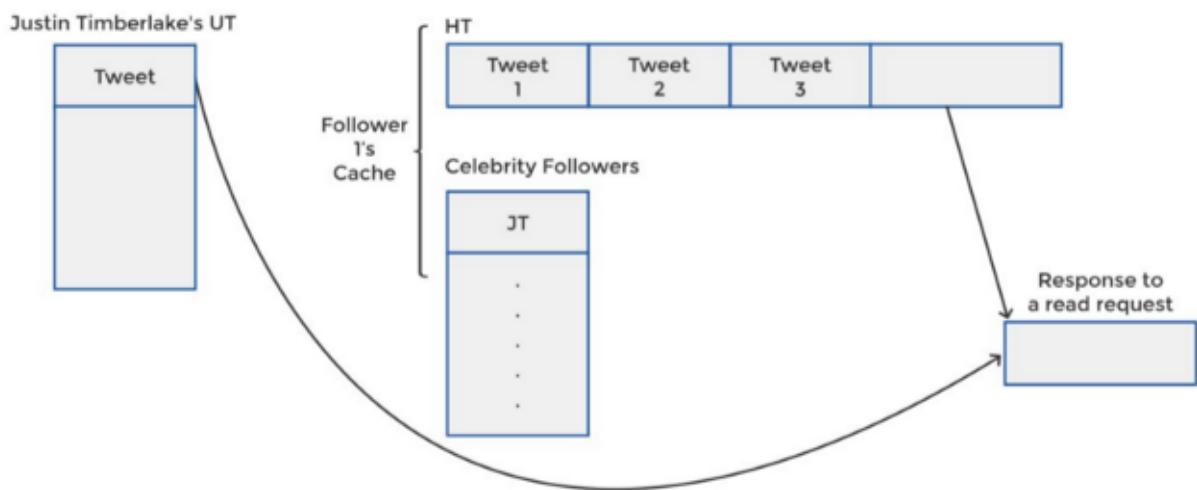
New entries from A and other followees are appended to the home timeline of each of these 5 followers through the fanout approach. Even if these 5 followers don't visit Twitter in the next few minutes, or even days, their in-memory home timeline gets updated each time one of their followees make a Tweet. As soon as the user fetches their home timeline, they'll see all the latest Tweets from their followees.



Now the ‘push’ model will work as long as the user has up to a few thousands of followers. What if Justin Timberlake (with 64M+ followers) makes a Tweet? A single tweet from a celebrity will have to be replicated in the cached home timelines of millions of followers with the ‘push’ fanout model. This isn’t the best approach in this case since pushing a single update to so many people will waste valuable resources. Instead, take a look at the alternative model discussed next.

The ‘Pull’ Model

If Follower 1 from the above diagram also follows Justin Timberlake and the celebrity makes a tweet, the process displayed in the above diagram will not happen. To save resources, Twitter follows a different approach for tweets from celebrities. This is the ‘pull’ model. Follower 1 will have all the tweets from his followees sitting in his cached home timeline, by ‘push’ fanout, except for tweets by the celebrities he follows. Instead, the cache for Follower 1 will also maintain a table for the celebrities he follows.



When Follower 1 fetches his home timeline, the application will scan his celebrity table, ‘pull’ the latest tweets from the cached user timeline of the celebrities, merge them with the tweets already present in Follower 1’s home timeline and display all the entries in chronological order.

Optimized Use Of Cache — Don’t Allocate Memory To Infrequent Users

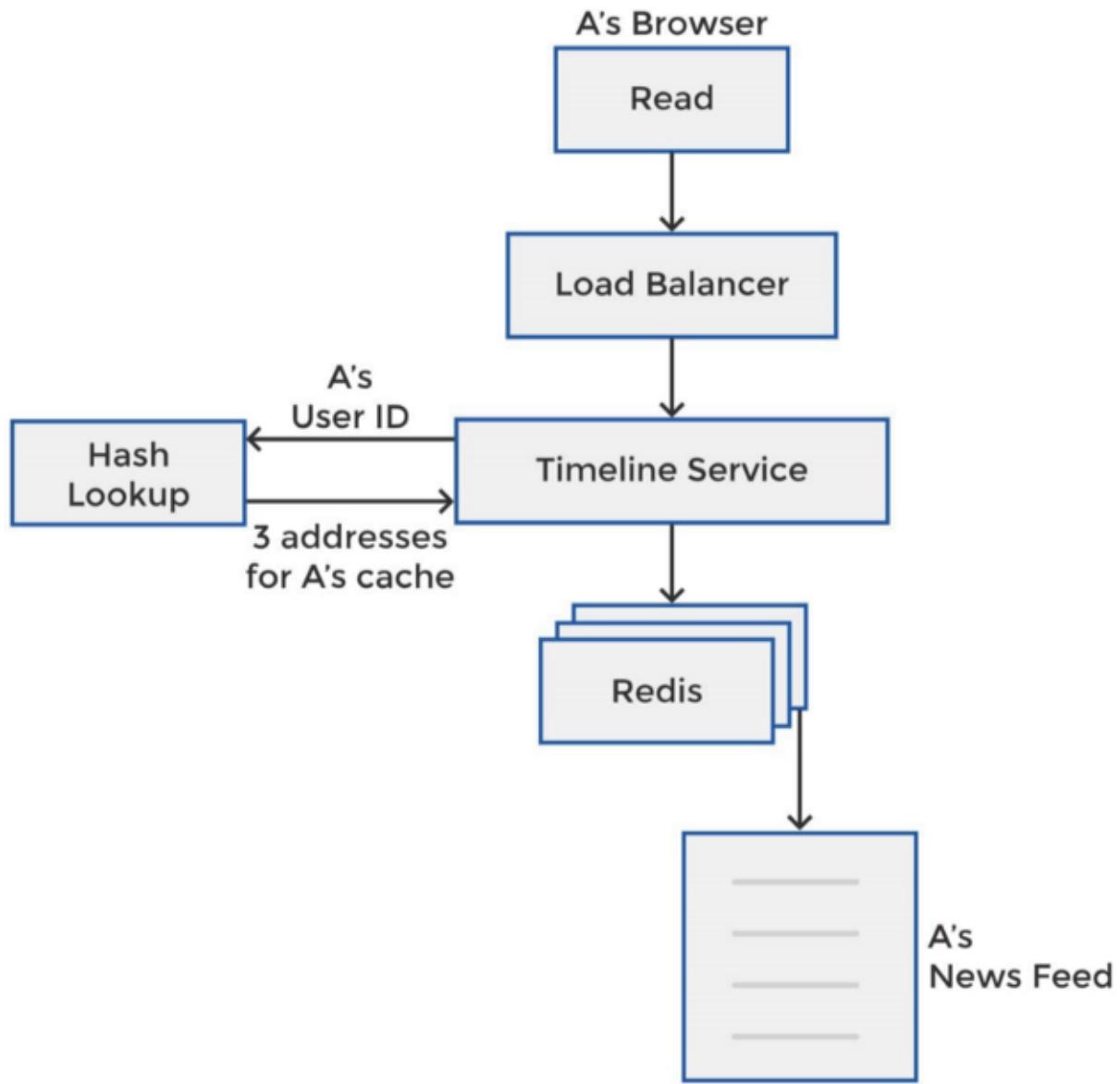
Now, the Redis maintains and updates home timelines for all the Twitter users. What if a user doesn't login for, say, 15 days. There are many users who don't login very frequently. Allocating space to them in the cache wastes resources unnecessarily.

To optimize memory, you can use an LRU-based cache that frees the memory space allocated to users who haven't accessed their News Feed for a long time. It would mean that the first time the user revisits Twitter after a long time, his hometime will be computed at that point and there will be some delay before the latest tweets are displayed on his homepage. The reason is that his precomputed home timeline does not sit in the Redis cache as in the case of the basic architecture described earlier in the post (used for frequent users).

Scaling Cache For Faster Reads

You already know that Twitter needs to optimize the system for the fastest reads. Whenever a user accesses his/her timeline, it should appear in no more than a few seconds. For faster processing, each user's cache is replicated across 3 Redis machines. It's possible that one of these machines gets updated faster and will hold the user's latest tweets, but eventually they will all have the same update.

To see how scaling user's cache across 3 Redis machines helps, let's take a look at what happens when user A accesses his timeline:

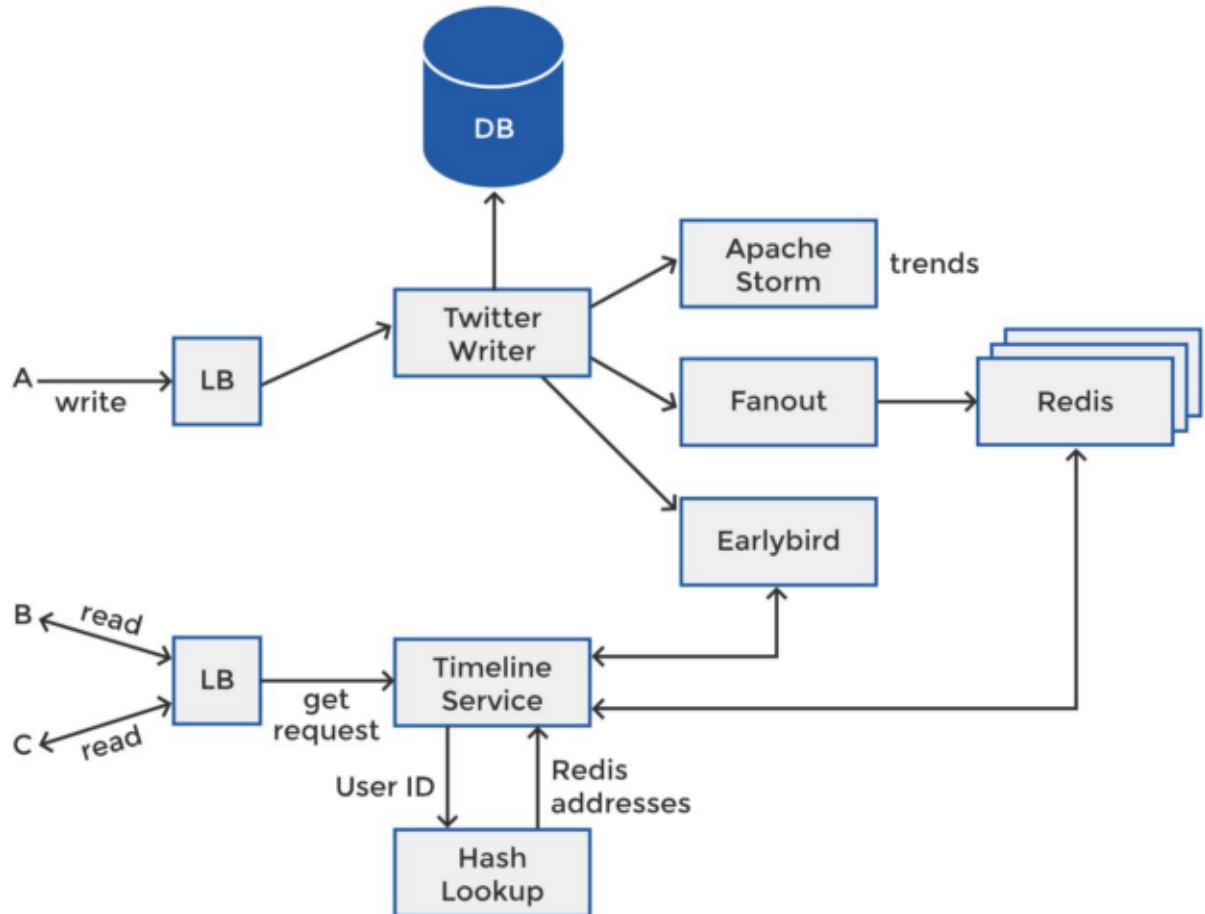


User A opens his browser and access his Twitter homepage. This initiates a *read* request that is sent to a load balancer from where it is forwarded to the timeline service. This service is simply an API to generate the user's News Feed. Now the service knows that A's cache is present at 3 Redis IP addresses. But since there are thousands of Redis machines, it needs to know which address to point to. This can be done by a hash ring lookup. The system can pass A's user ID to the hash lookup function. Hash lookup uses A's user ID to return the 3 Redis addresses where A's cache is present.

Now that the timeline service knows the 3 addresses where A's tweets are stored, it can reach out to the 3 Redis machines. However, it does not need a response from all three of the machines. The fastest of the 3 Redis machines to respond will display the tweets from A's followers on his home timeline. In the diagram above, the third Redis machine was the fastest to respond and display Tweets on A's News Feed.

Twitter System Diagram

Now that we've discussed many of the concepts in some detail, we have enough information to generate a system diagram to follow the flow of data through Twitter.



Tweet Generation

Let's see what happens when user A posts a tweet. The newly generated Tweet hits the load balancer. From there it's directed to the Twitter Writer which is the API for processing all incoming tweets. From the Twitter Writer, the tweet is replicated across different services:

1. A copy is stored in the database.
2. Another copy is sent to the Fanout service to push it to the home timelines of the followers stored across different Redis machines.
3. A copy of the same tweet is also replicated to Apache Storm to generate trending topics (or #tags).
4. The same tweet is also sent to Earlybird. Earlybird is a search service used by Twitter that splits the tweet into words and tags and then indexes them. These words are used to fetch the related Tweets and display the search timeline for a user when a search request is made.

Feed Publishing and Searches

Suppose user B types a keyword or #tag on Twitter to look for related posts. The query hits the load balancer. This request is sent to the timeline service, which forwards it to Earlybird. Earlybird calls all the different nodes and data centers to send back suitable responses for

the query. The results from all the nodes and data centers are collected and sent back to the application to display them on the user's search timeline.

When the user requests for home timeline or user timeline, the *read* request again hits the load balancer and is forwarded to the timeline service. The timeline service passes the user ID into a hash function that will return the IP address for the Redis machines that the timeline service should call to. The timeline service directly contacts Redis at those addresses to fetch the user's News Feed and return it back to the application.

