System Design Basics

Whenever we are designing a large system, we need to consider few things:

1. What are different architectural pieces that can be used?
2. How do these pieces work with each other?
3. How can we best utilize these pieces, what are the right tradeoffs?

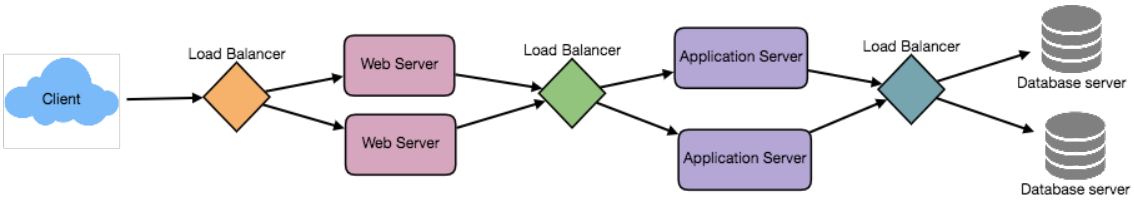
Investing in scaling before it is needed is generally not a smart business proposition; however, some forethought into the design can save valuable time and resources in the future. In the following chapters, we will focus on some of the core building blocks of scalable systems. Familiarizing with these concepts would greatly benefit in understanding distributed system design problems discussed later. In the next section, we will go through Consistent Hashing, CAP Theorem, Load Balancing, Caching, Data Partitioning, Indexes, Proxies, Queues, Replication, and choosing between SQL vs. NoSQL.

Load Balancing

Load balancer (LB) is another critical piece of any distributed system. It helps to distribute load across multiple resources according to some metric (random, round-robin, random with weighting for memory or CPU utilization, etc.). LB also keeps track of the status of all the resources while distributing requests. If a server is not available to take new requests or is not responding or has elevated error rate, LB will stop sending traffic to such a server.

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

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



There are many ways to implement load balancing.

1. Smart Clients

A smart client will take a pool of service hosts and balances load across them. It also detects hosts that are not responding to avoid sending requests their way. Smart clients also have to detect recovered hosts, deal with adding new hosts, etc.

Adding load-balancing functionality into the database (cache, service, etc.) client is usually an attractive solution for the developer. It looks easy to implement and manage especially when the system is not large. But as the system grows, LBs need to be evolved into standalone servers.

2. Hardware Load Balancers

The most expensive–but very high performance–solution to load balancing is to buy a dedicated hardware load balancer (something like a Citrix NetScaler). While they can solve a remarkable range of problems, hardware solutions are very expensive, and they are not trivial to configure.

As such, even large companies with large budgets will often avoid using dedicated hardware for all their load-balancing needs. Instead, they use them only as the first point of contact for user requests to their infrastructure and use other mechanisms (smart clients or the hybrid approach discussed in the next section) for load-balancing for traffic within their network.

3. Software Load Balancers

If we want to avoid the pain of creating a smart client, and since purchasing dedicated hardware is excessive, we can adopt a hybrid approach, called software load-balancers.

HAProxy is a one of the popular open source software LB. Load balancer can be placed between client and server or between two server side layers. If we can control the machine where the client is running, HAProxy could be running on the same machine. Each service we want to load balance can have a locally bound port (e.g., localhost:9000) on that machine, and the client will use this port to connect to the server. This port is, actually, managed by HAProxy; every client request on this port will be received by the proxy and then passed to the backend service in an efficient way (distributing load). If we can't manage client's machine, HAProxy can run on an intermediate server. Similarly, we can have proxies running between different server side components. HAProxy manages health checks and will remove or add machines to those pools. It also balances requests across all the machines in those pools.

For most systems, we should start with a software load balancer and move to smart clients or hardware load balancing as need arises.

Caching

Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible. Caches take advantage of the locality of reference principle: recently requested data is likely to be requested again. They are used in almost every layer of computing: hardware, operating systems, web browsers, web applications and more. A cache is like short-term memory: it has a limited amount of space, but is typically faster than the original data source and contains the most recently accessed items. Caches can exist at all levels in architecture but are often found at the level nearest to the front end, where they are implemented to return data quickly without taxing downstream levels.

1. Application server cache

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

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

2. Distributed cache

In a distributed cache, each of its nodes own part of the cached data. Typically, the cache is divided up using a consistent hashing function, such that if a request node is looking for a certain piece of data, it can quickly know where to look within the distributed cache to determine if that data is available. In this case, each node has a small piece of the cache, and will then send a request to another node for the data before going to the origin. Therefore, one of the advantages of a distributed cache is the ease by which we can increase the cache space, which can be achieved just by adding nodes to the request pool.

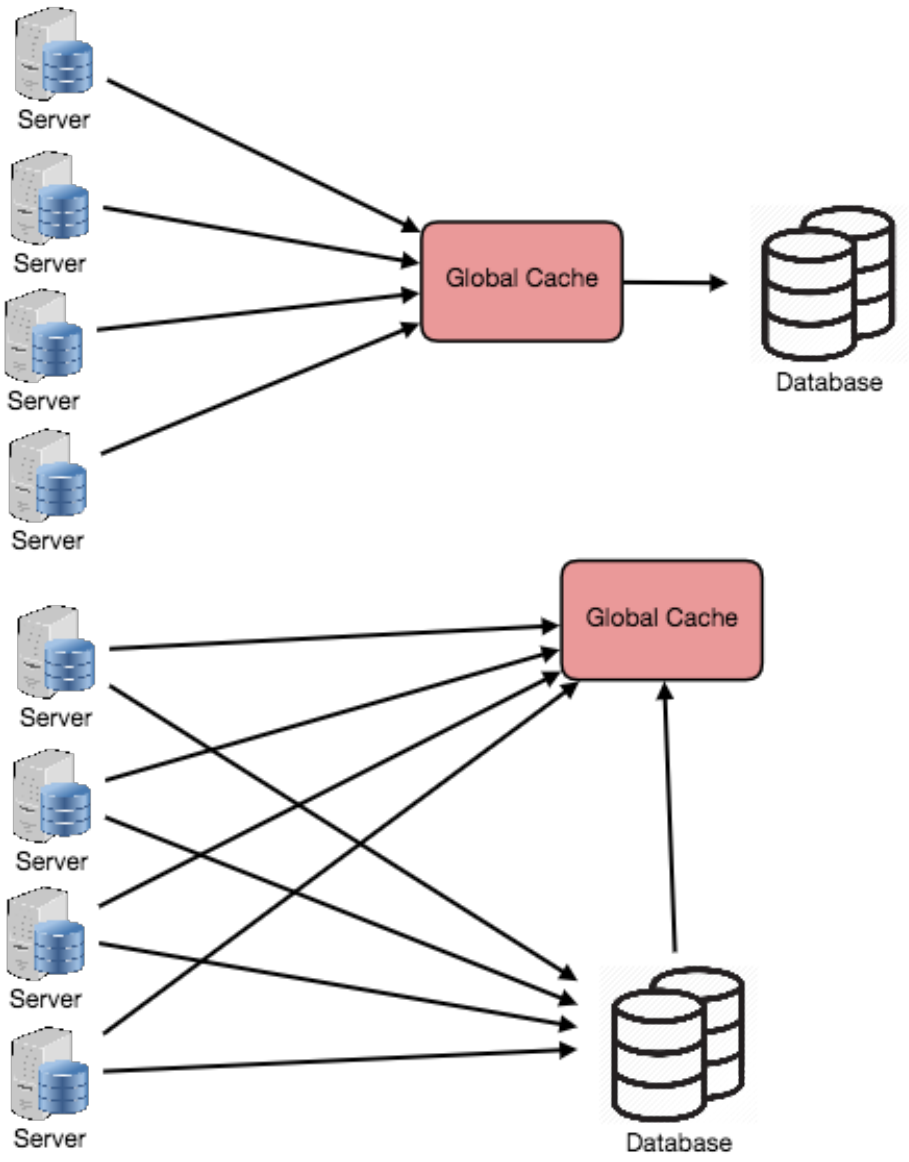
A disadvantage of distributed caching is resolving a missing node. Some distributed caches get around this by storing multiple copies of the data on different nodes; however, you can imagine how this logic can get complicated quickly, especially when you add or remove nodes from the request layer. Although even if a node disappears and part of the cache is lost, the requests will just pull from the origin—so it isn't necessarily catastrophic!

3. Global Cache

A global cache is just as it sounds: all the nodes use the same single cache space. This involves adding a server, or file store of some sort, faster than your original store and accessible by all the request layer nodes. Each of the request nodes queries the cache in the same way it would a local one. This kind of caching scheme can get a bit complicated because it is very easy to overwhelm a single cache as the number of clients and requests increase, but is very effective in some architectures (particularly ones with specialized hardware that make this global cache very fast, or that have a fixed dataset that needs to be cached).

There are two common forms of global caches depicted in the following diagram. First, when a cached response is not found in the cache, the cache itself becomes responsible for retrieving the missing piece of data from the underlying store. Second, it is the responsibility of request nodes to retrieve any data that is not found in the cache.

Most applications leveraging global caches tend to use the first type, where the cache itself manages eviction and fetching data to prevent a flood of requests for the same data from the clients. However, there are some cases where the second implementation makes more sense. For example, if the cache is being used for very large files, a low cache hit percentage would cause the cache buffer to become overwhelmed with cache misses; in this situation, it helps to have a large percentage of the total data set (or hot data set) in the cache. Another example is an architecture where the files stored in the cache are static and shouldn't be evicted. (This could be because of application requirements around that data latency—certain pieces of data might need to be very fast for large data sets—where the application logic understands the eviction strategy or hot spots better than the cache.)



4. Content Distribution Network (CDN)

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

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

Cache Invalidation

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

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

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

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

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

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

Cache eviction policies

Following are some of the most common cache eviction policies:

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

Sharding or Data Partitioning

Data partitioning (also known as sharding) is a technique to break up a big database (DB) into many smaller parts. It is the process of splitting up a DB/table across multiple machines to improve the manageability, performance, availability and load balancing of an application. The justification for data sharding is that, after a certain scale point, it is cheaper and more feasible to scale horizontally by adding more machines than to grow it vertically by adding beefier servers.

1. Partitioning Methods

There are many different schemes one could use to decide how to break up an application database into multiple smaller DBs. Below are three of the most popular schemes used by various large scale applications.

a. Horizontal partitioning: In this scheme, we put different rows into different tables. For example, if we are storing different places in a table, we can decide that locations with ZIP codes less than 10000 are stored in one table, and places with ZIP codes greater than 10000 are stored in a separate table. This is also called a range based sharding, as we are storing different ranges of data in separate tables.

The key problem with this approach is that if the value whose range is used for sharding isn't chosen carefully, then the partitioning scheme will lead to unbalanced servers. In the previous example, splitting location based on their zip codes assumes that places will be evenly distributed across the different zip codes. This assumption is not valid as there will be a lot of places in a thickly populated area like Manhattan compared to its suburb cities.

b. Vertical Partitioning: In this scheme, we divide our data to store tables related to a specific feature to their own server. For example, if we are building Instagram like application, where we need to store data related to users, all the photos they upload and people they follow, we can decide to place user profile information on one DB server, friend lists on another and photos on a third server.

Vertical partitioning is straightforward to implement and has a low impact on the application. The main problem with this approach is that if our application experiences additional growth, then it may be necessary to further partition a feature specific DB across various servers (e.g. it would not be possible for a single server to handle all the metadata queries for 10 billion photos by 140 million users).

c. Directory Based Partitioning: A loosely coupled approach to work around issues mentioned in above schemes is to create a lookup service which knows your current partitioning scheme and abstracts it away from the DB access code. So, to find out where does a particular data entity resides, we query our directory server that holds the mapping between each tuple key to its DB server. This loosely coupled approach means we can perform tasks like adding servers to the DB pool or change our partitioning scheme without having to impact your application.

2. Partitioning Criteria

a. Key or Hash-based partitioning: Under this scheme, we apply a hash function to some key attribute of the entity we are storing, that yields the partition number. For example, if we have 100 DB servers and our ID is a numeric value that gets incremented by one, each time a new record is inserted. In this example, the hash function could be 'ID % 100', which will give us the server number where we can store/read that record. This approach should ensure a uniform allocation of data among servers. The fundamental problem with this approach is that it effectively fixes the total number of DB servers, since adding new servers means changing the hash function which would require redistribution of data and downtime for the service. A workaround for this problem is to use Consistent Hashing.

b. List partitioning: In this scheme, each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there. For example, we can decide all users living in Iceland, Norway, Sweden, Finland or Denmark will be stored in a partition for the Nordic countries.

c. Round-robin partitioning: This is a very simple strategy that ensures uniform data distribution. With 'n' partitions, the 'i' tuple is assigned to partition (i mod n).

d. Composite partitioning: Under this scheme, we combine any of above partitioning schemes to devise a new scheme. For example, first applying a list partitioning and then a hash based partitioning. Consistent hashing could be considered a composite of hash and list partitioning where the hash reduces the key space to a size that can be listed.

3. Common Problems of Sharding

On a sharded database, there are certain extra constraints on the different operations that can be performed. Most of these 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. Below are some of the constraints and additional complexities introduced by sharding:

a. Joins and Denormalization: Performing joins on a database which is running on one server is straightforward, but once a database is partitioned and spread across multiple machines it is often not feasible to perform joins that span database shards. Such joins will not be performance efficient since data has to be compiled from multiple servers. A common workaround for this problem is to denormalize the database so that queries that previously required joins can be performed from a single table. Of course, the service now has to deal with all the perils of denormalization such as data inconsistency.

b. Referential integrity: As we saw that performing a cross-shard query on a partitioned database is not feasible, similarly trying to enforce data integrity constraints such as foreign keys in a sharded database can be extremely difficult.

Most of RDBMS do not support foreign keys constraints across databases on different database servers. Which means that applications that require referential integrity on sharded databases often have to enforce it in application code. Often in such cases, applications have to run regular SQL jobs to clean up dangling references.

c. Rebalancing: There could be many reasons we have to change our sharding scheme:

1. The data distribution is not uniform, e.g., there are a lot of places for a particular ZIP code, that cannot fit into one database partition.
2. There are a lot of load on a shard, e.g., there are too many requests being handled by the DB shard dedicated to user photos.

In such cases, either we have to create more DB shards or have to rebalance existing shards, which means the partitioning scheme changed and all existing data moved to new locations. Doing this without incurring downtime is extremely difficult. Using a scheme like directory based partitioning does make rebalancing a more palatable experience at the cost of increasing the complexity of the system and creating a new single point of failure (i.e. the lookup service/database).

Indexes

Indexes are well known when it comes to databases; they are used to improve the speed of data retrieval operations on the data store. An index makes the trade-offs of increased storage overhead, and slower writes (since we not only have to write the data but also have to update the index) for the benefit of faster reads. Indexes are used to quickly locate data without having to examine every row in a database table. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

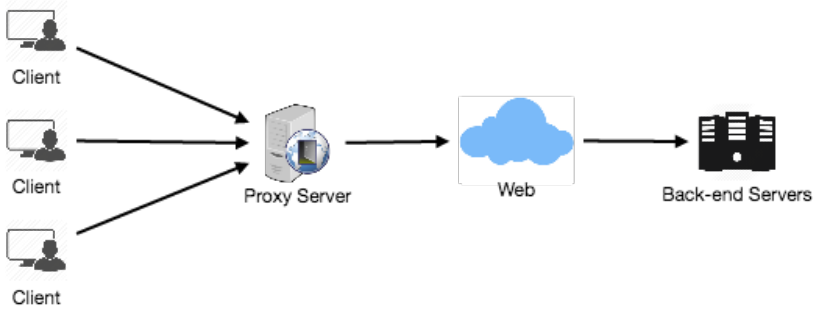
An index is a data structure that can be perceived as a table of contents that points us to the location where actual data lives. So when we create an index on a column of a table, we store that column and a pointer to the whole row in the index. Indexes are also used to create different views of the same data. For large data sets, this is an excellent way to specify different filters or sorting schemes without resorting to creating multiple additional copies of the data.

Just as to a traditional relational data store, we can also apply this concept to larger data sets. The trick with indexes is that we must carefully consider how users will access the data. In the case of data sets that are many TBs in size but with very small payloads (e.g., 1 KB), indexes are a necessity for optimizing data access. Finding a small payload in such a large data set can be a real challenge since we can't possibly iterate over that much data in any reasonable time. Furthermore, it is very likely that such a large data set is spread over several physical devices—this means we need some way to find the correct physical location of the desired data. Indexes are the best way to do this.

Proxies

A proxy server is an intermediary piece of hardware/software that sits between the client and the back-end server. It receives requests from clients and relays them to the origin servers. Typically, proxies are used to filter requests or log requests, or sometimes transform requests (by adding/removing headers,

encrypting/decrypting, or compression). Another advantage of a proxy server is that its cache can serve a lot of requests. If multiple clients access a particular resource, the proxy server can cache it and serve all clients without going to the remote server.



Proxies are also extremely helpful when coordinating requests from multiple servers and can be used to optimize request traffic from a system-wide perspective. For example, we can collapse the same (or similar) data access requests into one request and then return the single result to the user; this scheme is called collapsed forwarding.

Imagine there is a request for the same data across several nodes, and that piece of data is not in the cache. If these requests are routed through the proxy, then all them can be collapsed into one, which means we will be reading the required data from the disk only once.

Another great way to use the proxy is to collapse requests for data that is spatially close together in the storage (consecutively on disk). This strategy will result in decreasing request latency. For example, let's say a bunch of servers request parts of file: part1, part2, part3, etc. We can set up our proxy in such a way that it can recognize the spatial locality of the individual requests, thus collapsing them into a single request and reading complete file, which will greatly minimize the reads from the data origin. Such scheme makes a big difference in request time when we are doing random accesses across TBs of data. Proxies are particularly useful under high load situations, or when we have limited caching since proxies can mostly batch several requests into one.

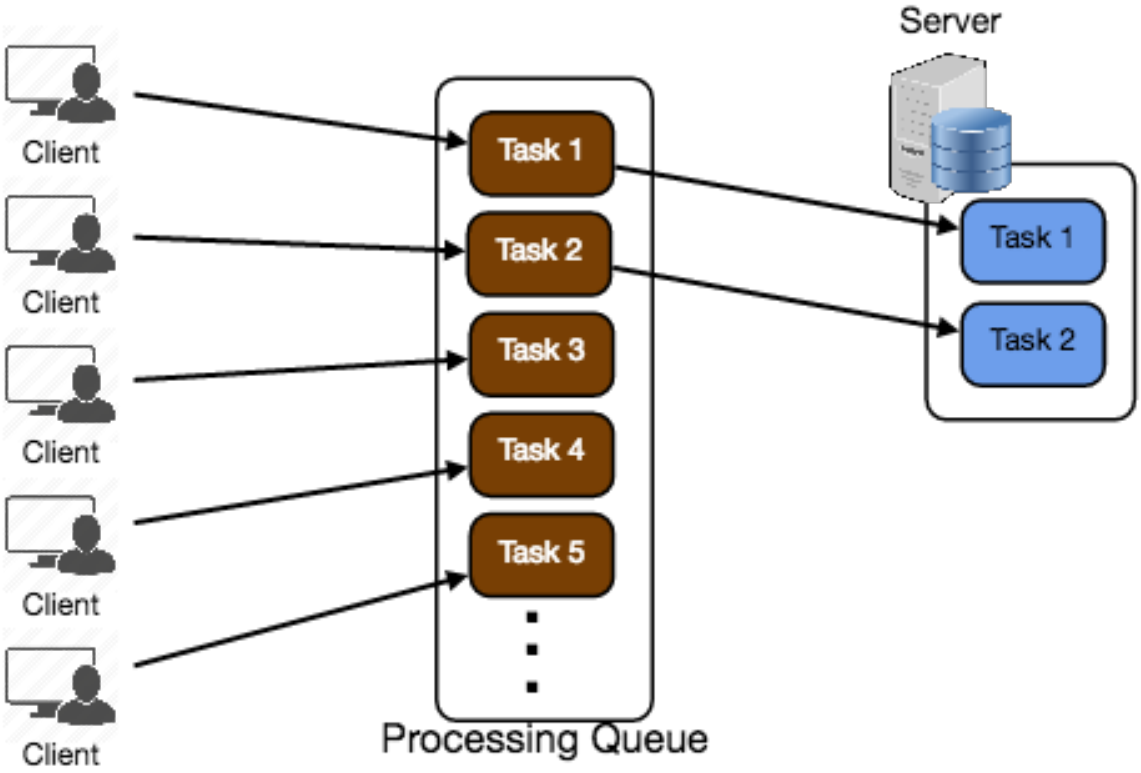
Queues

Queues are used to effectively manage requests in a large-scale distributed system. In small systems with minimal processing loads and small databases, writes can be predictably fast; however, in more complex and large systems

writes can take an almost non-deterministically long time. For example, data may have to be written in different places on different servers or indices, or the system could simply be under high load. In such cases where individual writes (or tasks) may take a long time, achieving high performance and availability requires different components of the system to work in an asynchronous way; a common way to do that is with queues.

Let's assume a system where each client is requesting a task to be processed on a remote server. Each of these clients sends their requests to the server, and the server tries to finish the tasks as quickly as possible to return the results to the respective clients. In small systems where one server can handle incoming requests just as fast as they come, this kind of situation should work just fine. However, when the server gets more requests than it can handle, then each client is forced to wait for other clients' requests to finish before a response can be generated.

This kind of synchronous behavior can severely degrade client's performance; the client is forced to wait, effectively doing zero work, until its request can be responded. Adding extra servers to address high load does not solve the problem either; even with effective load balancing in place, it is very difficult to ensure the fair and balanced distribution of work required to maximize client performance. Further, if the server processing the requests is unavailable, or fails, then the clients upstream will fail too. Solving this problem effectively requires building an abstraction between the client's request and the actual work performed to service it.



A processing queue is as simple as it sounds: all incoming tasks are added to the queue, and as soon as any worker has the capacity to process, they can pick up a task from the queue. These tasks could represent a simple write to a database, or something as complex as generating a thumbnail preview image for a document.

Queues are implemented on the asynchronous communication protocol, meaning when a client submits a task to a queue they are no longer required to wait for the results; instead, they need only acknowledgment that the request was properly received. This acknowledgment can later serve as a reference for the results of the work when the client requires it. Queues have implicit or explicit limits on the size of data that may be transmitted in a single request and the number of requests that may remain outstanding on the queue.

Queues are also used for fault tolerance as they can provide some protection from service outages and failures. For example, we can create a highly robust queue that can retry service requests that have failed due to transient system failures. It

is preferable to use a queue to enforce quality-of-service guarantees than to expose clients directly to intermittent service outages, requiring complicated and often inconsistent client-side error handling.

Queues play a vital role in managing distributed communication between different parts of any large-scale distributed system. There are a lot of ways to implement them and quite a few open source implementations of queues available like RabbitMQ, ZeroMQ, ActiveMQ, and BeanstalkD.

Redundancy and Replication

Redundancy means duplication of critical data or services with the intention of increased reliability of the system. For example, if there is only one copy of a file stored on a single server, then losing that server means losing the file. Since losing data is seldom a good thing, we can create duplicate or redundant copies of the file to solve this problem.

This same principle applies to services too. If we have a critical service in our system, ensuring that multiple copies or versions of it are running simultaneously can secure against the failure of a single node.

Creating redundancy in a system can remove single points of failure and provide backups if needed in a crisis. For example, if we have two instances of a service running in production, and if one fails or degrades, the system can failover to the other one. These failovers can happen automatically or can be done manually.

Primay Server → Secondary Server

Data replication

Active Data----------→ Mirrored Data

Another important part of service redundancy is to create a shared-nothing architecture, where each node can operate independently of one another. There should not be any central service managing state or orchestrating activities for the other nodes. This helps a lot with scalability since new servers can be added without special conditions or knowledge and most importantly, such systems are more resilient to failure as there is no single point of failure.

SQL vs. NoSQL

In the world of databases, there are two main types of solutions: SQL and NoSQL - or relational databases and non-relational databases. Both of them differ in the way they were built, the kind of information they store, and how they store it.

Relational databases are structured and have predefined schemas, like phone books that store phone numbers and addresses. Non-relational databases are unstructured, distributed and have a dynamic schema, like file folders that hold everything from a person's address and phone number to their Facebook 'likes' and online shopping preferences.

SQL

Relational databases store data in rows and columns. Each row contains all the information about one entity, and columns are all the separate data points. Some of the most popular relational databases are MySQL, Oracle, MS SQL Server, SQLite, Postgres, MariaDB, etc.

NoSQL

Following are most common types of NoSQL:

Key-Value Stores: Data is stored in an array of key-value pairs. The 'key' is an attribute name, which is linked to a 'value'. Well-known key value stores include Redis, Voldemort and Dynamo.

Document Databases: In these databases data is stored in documents, instead of rows and columns in a table, and these documents are grouped together in collections. Each document can have an entirely different structure. Document databases include the CouchDB and MongoDB.

Wide-Column Databases: Instead of 'tables,' in columnar databases we have column families, which are containers for rows. Unlike relational databases, we don't need to know all the columns up front, and each row doesn't have to have the same number of columns. Columnar databases are best suited for analyzing large datasets - big names include Cassandra and HBase.

Graph Databases: These databases are used to store data whose relations are best represented in a graph. Data is saved in graph structures with nodes (entities), properties (information about the entities) and lines (connections between the entities). Examples of graph database include Neo4J and InfiniteGraph.

High level differences between SQL and NoSQL

Storage: SQL stores data in tables, where each row represents an entity, and each column represents a data point about that entity; for example, if we are storing a car entity in a table, different columns could be 'Color', 'Make', 'Model', and so on.

NoSQL databases have different data storage models. The main ones are key-value, document, graph and columnar. We will discuss differences between these databases below.

Schema: In SQL, each record conforms to a fixed schema, meaning the columns must be decided and chosen before data entry and each row must have data for each column. The schema can be altered later, but it involves modifying the whole database and going offline.

Whereas in NoSQL, schemas are dynamic. Columns can be added on the fly, and each 'row' (or equivalent) doesn't have to contain data for each 'column.'

Querying: SQL databases uses SQL (structured query language) for defining and manipulating the data, which is very powerful. In NoSQL database, queries are focused on a collection of documents. Sometimes it is also called UnQL (Unstructured Query Language). Different databases have different syntax for using UnQL.

Scalability: In most common situations, SQL databases are vertically scalable, i.e., by increasing the horsepower (higher Memory, CPU, etc.) of the hardware, which can get very expensive. It is possible to scale a relational database across multiple servers, but this is a challenging and time-consuming process.

On the other hand, NoSQL databases are horizontally scalable, meaning we can add more servers easily in our NoSQL database infrastructure to handle large traffic. Any cheap commodity hardware or cloud instances can host NoSQL databases, thus making it a lot more cost-effective than vertical scaling. A lot of NoSQL technologies also distribute data across servers automatically.

Reliability or ACID Compliancy (Atomicity, Consistency, Isolation, Durability): The vast majority of relational databases are ACID compliant. So, when it comes to data reliability and safe guarantee of performing transactions, SQL databases are still the better bet.

Most of the NoSQL solutions sacrifice ACID compliance for performance and scalability.

SQL VS. NoSQL - Which one to use?

When it comes to database technology, there's no one-size-fits-all solution. That's why many businesses rely on both relational and non-relational databases for different needs. Even as NoSQL databases are gaining popularity for their speed and scalability, there are still situations where a highly structured SQL database may perform better; choosing the right technology hinges on the use case.

Reasons to use SQL database

Here are a few reasons to choose a SQL database:

1. We need to ensure ACID compliance. ACID compliance reduces anomalies and protects the integrity of your database by prescribing exactly how transactions interact with the database. Generally, NoSQL databases sacrifice ACID compliance for scalability and processing speed, but for many e-commerce and financial applications, an ACID-compliant database remains the preferred option.
2. Your data is structured and unchanging. If your business is not experiencing massive growth that would require more servers and if you're only working with data that's consistent, then there may be no reason to use a system designed to support a variety of data types and high traffic volume.

Reasons to use NoSQL database

When all the other components of our application are fast and seamless, NoSQL databases prevent data from being the bottleneck. Big data is contributing to a large success for NoSQL databases, mainly because it handles data differently than the traditional relational databases. A few popular examples of NoSQL databases are MongoDB, CouchDB, Cassandra, and HBase.

1. Storing large volumes of data that often have little to no structure. A NoSQL database sets no limits on the types of data we can store together and allows us to add different new types as the need changes. With document-based databases, you can store data in one place without having to define what "types" of data those are in advance.
2. Making the most of cloud computing and storage. Cloud-based storage is an excellent cost-saving solution but requires data to be easily spread across multiple servers to scale up. Using commodity (affordable, smaller) hardware on-site or in the cloud saves you the hassle of additional software, and NoSQL databases like Cassandra are designed to be scaled across multiple data centers out of the box without a lot of headaches.

3. Rapid development. NoSQL is extremely useful for rapid development as it doesn't need to be prepped ahead of time. If you're working on quick iterations of your system which require making frequent updates to the data structure without a lot of downtime between versions, a relational database will slow you down.

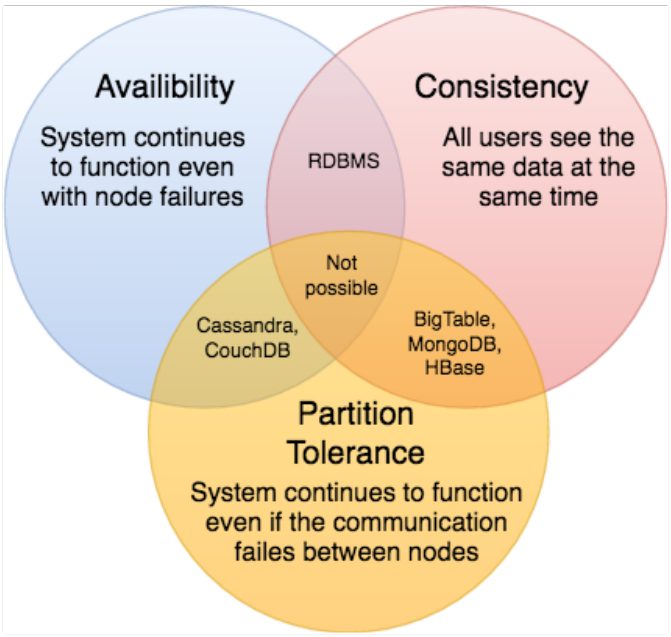
CAP Theorem

CAP theorem states that it is impossible for a distributed software system to simultaneously provide more than two out of three of the following guarantees (CAP): Consistency, Availability and Partition tolerance. When we design a distributed system, trading off among CAP is almost the first thing we want to consider. CAP theorem says while designing a distributed system we can pick only two of:

Consistency: All nodes see the same data at the same time. Consistency is achieved by updating several nodes before allowing further reads.

Availability: Every request gets a response on success/failure. Availability is achieved by replicating the data across different servers.

Partition tolerance: System continues to work despite message loss or partial failure. A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.



We cannot build a general data store that is continually available, sequentially consistent and tolerant to any partition failures. We can only build a system that has any two of these three properties. Because, to be consistent, all nodes should see the same set of updates in the same order. But if the network suffers a partition, updates in one partition might not make it to the other partitions before a client reads from the out-of-date partition after having read from the up-to-date one. The only thing that can be done to cope with this possibility is to stop serving requests from the out-of-date partition, but then the service is no longer 100% available.

Consistent Hashing

Distributed Hash Table (DHT) is one of the fundamental component used in distributed scalable systems. Hash Tables need key, value and a hash function, where hash function maps the key to a location where the value is stored.

index = hash_function(key)

Suppose we are designing a distributed caching system. Given 'n' cache servers, an intuitive hash function would be 'key % n'. It is simple and commonly used. But it has two major drawbacks:

1. It is NOT horizontally scalable. Whenever a new cache host is added to the system, all existing mappings are broken. It will be a pain point in maintenance if the caching system contains lots of data. Practically it becomes difficult to schedule a downtime to update all caching mappings.
2. It may NOT be load balanced, especially for non-uniformly distributed data. In practice, it can be easily assumed that the data will not be distributed uniformly. For the caching system, it translates into some caches becoming hot and saturated while the others idle and almost empty.

In such situations, consistent hashing is a good way to improve the caching system.

What is Consistent Hashing?

Consistent hashing is a very useful strategy for distributed caching system and DHTs. It allows distributing data across a cluster in such a way that will minimize reorganization when nodes are added or removed. Hence, making the caching system easier to scale up or scale down.

In Consistent Hashing when the hash table is resized (e.g. a new cache host is added to the system), only k/n keys need to be remapped, where k is the total number of keys and n is the total number of servers. Recall that in a caching system using the 'mod' as the hash function, all keys need to be remapped.

In consistent hashing objects are mapped to the same host if possible. When a host is removed from the system, the objects on that host are shared by other hosts; and when a new host is added, it takes its share from a few hosts without touching other's shares.

How it works?

As a typical hash function, consistent hashing maps a key to an integer. Suppose the output of the hash function is in the range of [0, 256). Imagine that the integers in the range are placed on a ring such that the values are wrapped around.

Here's how consistent hashing works:

1. Given a list of cache servers, hash them to integers in the range.
2. To map a key to a server,
    - Hash it to a single integer.
    - Move clockwise on the ring until finding the first cache it encounters.

- That cache is the one that contains the key. See animation below as an example: key1 maps to cache A; key2 maps to cache C.

To add a new server, say D, keys that were originally residing at C will be split. Some of them will be shifted to D, while other keys will not be touched.

To remove a cache or if a cache failed, say A, all keys that were originally mapping to A will fall into B, and only those keys need to be moved to B, other keys will not be affected.

For load balancing, as we discussed in the beginning, the real data is essentially randomly distributed and thus may not be uniform. It may make the keys on caches unbalanced.

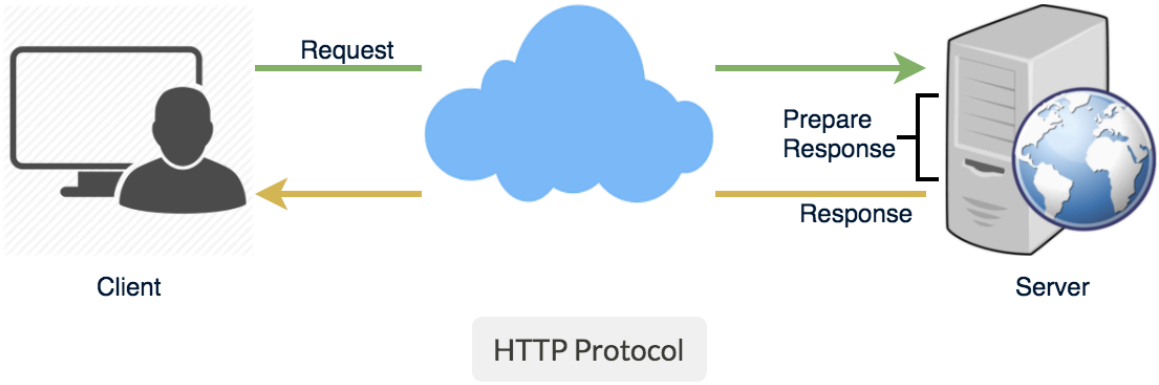
To handle this issue, we add “virtual replicas” for caches. Instead of mapping each cache to a single point on the ring, we map it to multiple points on the ring, i.e. replicas. This way, each cache is associated with multiple portions of the ring.

If the hash function is “mixes well,” as the number of replicas increases, the keys will be more balanced.

Long-Polling vs WebSockets vs Server-Sent Events

Long-Polling, WebSockets, and Server-Sent Events are popular communication protocols between a client like a web browser and a web server. First, let’s start with understanding what a standard HTTP web request looks like. Following are a sequence of events for regular HTTP request:

1. Client opens a connection and requests data from the server.
2. The server calculates the response.
3. The server sends the response back to the client on the opened request.

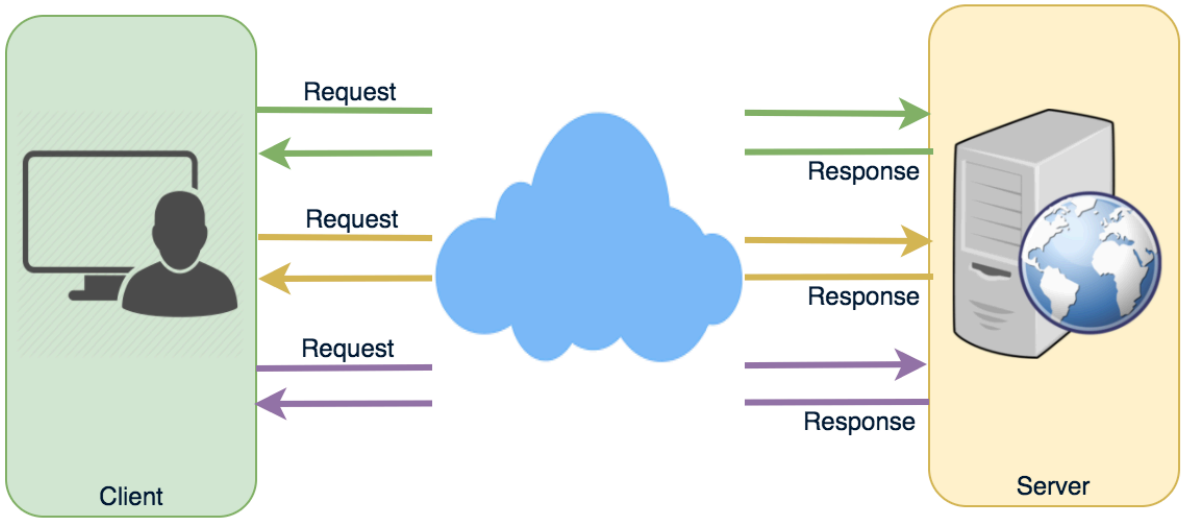


Ajax Polling

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

1. 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, just like regular HTTP traffic.
4. Client repeats the above three steps periodically to get updates from the server.

Problem with Polling is that the client has to keep asking the server for any new data. As a result, a lot of responses are empty creating HTTP overhead.



Ajax Polling Protocol

HTTP Long-Polling

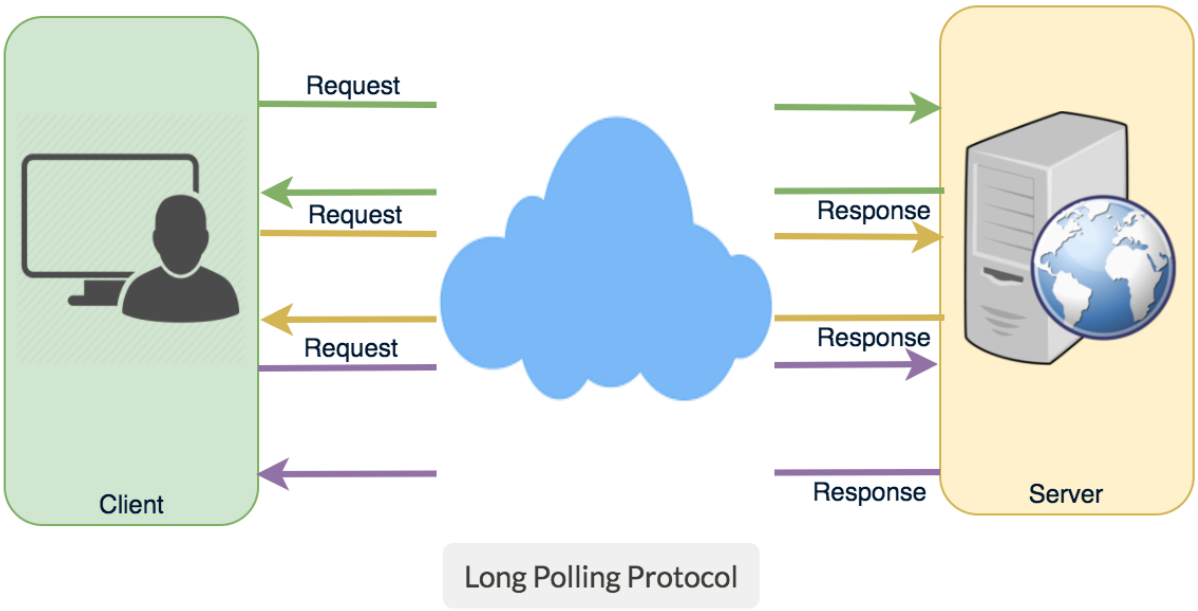
A variation of the traditional polling technique that allows the server to push information to a client, whenever the data is available. With Long-Polling, the client requests information from the server exactly as in normal polling, but with the expectation that the server may not respond immediately. That's why this technique is sometimes referred to as a "Hanging GET".

- If the server does not have any data available for the client, instead of sending an empty response, the server holds the request and waits until some data becomes available.
- Once the data becomes available, a full response is sent to the client. The client then immediately re-request information from the server so that the server will almost always have an available waiting request that it can use to deliver data in response to an event.

The basic life cycle of an application using HTTP Long-Polling is as follows:

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.

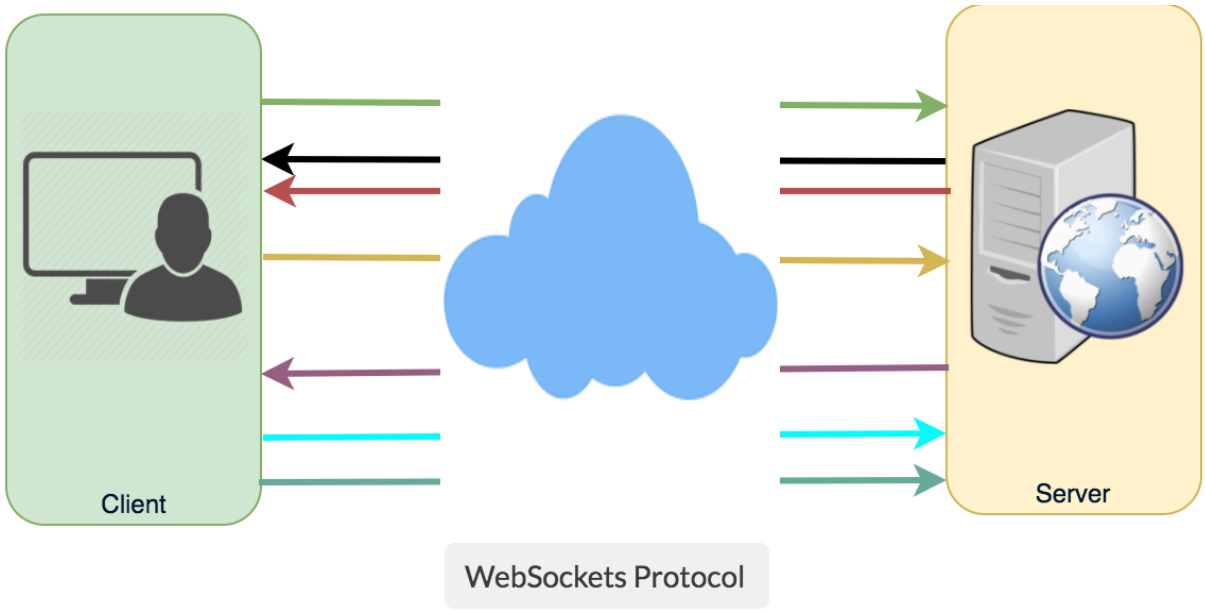
5. Each Long-Poll request has a timeout. The client has to reconnect periodically after the connection is closed, due to timeouts.



Long Polling Protocol

WebSockets

WebSocket provides Full duplex communication channels over a single TCP connection. It provides a persistent connection between a client and a server that both parties can use to start sending 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. The WebSocket protocol enables communication between a client and a server with lower overheads, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the browser without being asked by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way, a two-way (bi-directional) ongoing conversation can take place between a client and a server.



Server-Sent Events (SSEs)

Under SSEs 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 to do so.

1. 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.

SSEs are best when we need real-time traffic from the server to the client or if the server is generating data in a loop and will be sending multiple events to the client.

Server Sent Events Protocol

System Design Interviews: A step by step guide

Lots of people struggle with system design interviews (SDIs) primarily because of 1) Unstructured nature of SDIs, where you're asked to work on an open-ended design problem that doesn't have a standard answer, 2) Your lack of experience in developing large scale systems and, 3) You've not spend enough time to prepare for SDIs.

Just like coding interviews, candidates who haven't spent time preparing for SDIs mostly perform poorly. This gets aggravated when you're interviewing at the top companies like Google, Facebook, Uber, etc. In these companies, if a candidate doesn't perform above average, they have a limited chance to get an offer. On the other hand, a good performance always results in a better offer (higher position and salary), since it reflects your ability to handle complex systems.

In this course, we'll follow a step by step approach to solve multiple design problems. Here are those seven steps:

Step 1: Requirements clarifications

Always ask questions to find the exact scope of the problem you're solving. Design questions are mostly open-ended, and they don't have ONE correct answer, that's why clarifying ambiguities early in the interview becomes critical.

Candidates who spend enough time to clearly define the end goals of the system, always have a better chance to be successful in the interview. Also, since you only have 35-40 minutes to design a (supposedly) large system, you should clarify what parts of the system you would be focusing on.

Under each step, we'll try to give examples of different design considerations for developing a Twitter-like service.

Here are some questions for designing Twitter that should be answered before moving on to next steps:

- Will users of our service be able to post tweets and follow other people?
- Should we also design to create and display user's timeline?
- Will tweets contain photos and videos?
- Are we focusing on backend only or are we developing front-end too?
- Will users be able to search tweets?
- Do we need to display hot trending topics?
- Would there be any push notification for new (or important) tweets?

All such question will determine how our end design will look like.

Step 2: System interface definition

Define what APIs are expected from the system. This would not only establish the exact contract expected from the system but would also ensure if you haven't gotten any requirements wrong. Some examples for our Twitter-like service would be:

```
postTweet(user_id, tweet_data, tweet_location, user_location, timestamp, …)
```

```
generateTimeline(user_id, current_time, user_location, …)
```

```
markTweetFavorite(user_id, tweet_id, timestamp, …)
```

Step 3: Back-of-the-envelope estimation

It's always a good idea to estimate the scale of the system you're going to design. This would also help later when you'll be focusing on scaling, partitioning, load balancing and caching.

- What scale is expected from the system (e.g., number of new tweets, number of tweet views, how many timeline generations per sec., etc.)?
- How much storage would we need? We'll have different numbers if users can have photos and videos in their tweets.

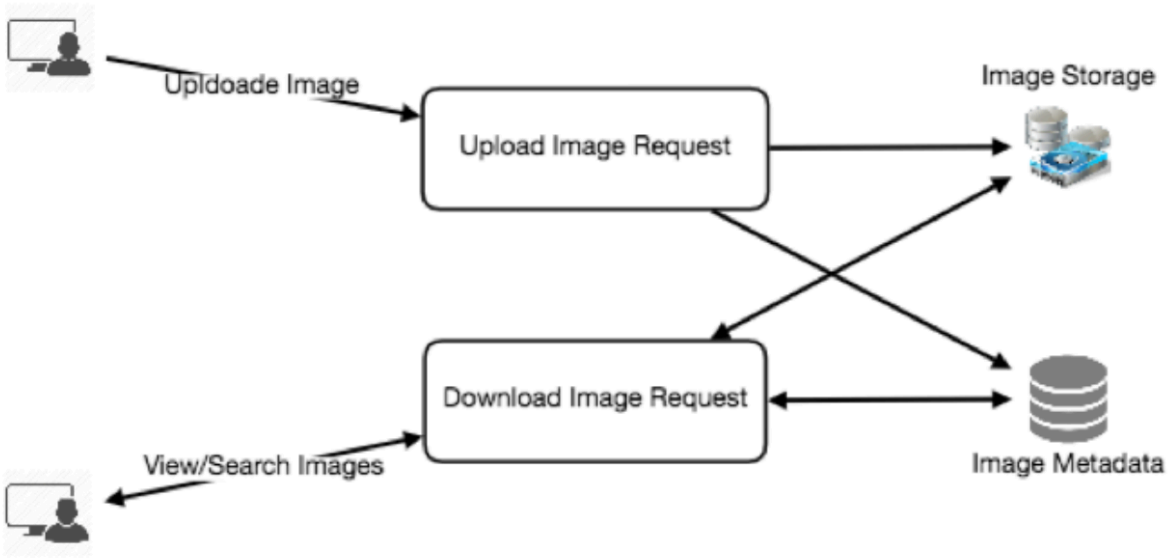
- What network bandwidth usage are we expecting? This would be crucial in deciding how would we manage traffic and balance load between servers.

Step 4: Defining data model

Defining the data model early will clarify how data will flow among different components of the system. Later, it will guide towards data partitioning and management. Candidate should be able to identify various entities of the system, how they will interact with each other and different aspect of data management like storage, transportation, encryption, etc. Here are some entities for our Twitter-like service:

User: UserID, Name, Email, DoB, CreationData, LastLogin, etc.
Tweet: TweetID, Content, TweetLocation, NumberOfLikes, TimeStamp, etc.
UserFollowos: UserdID1, UserID2
FavoriteTweets: UserID, TweetID, TimeStamp

Which database system should we use? Would NoSQL like Cassandra best fits our needs, or we should use MySQL-like solution. What kind of block storage should we use to store photos and videos?



Step 5: High-level design

Draw a block diagram with 5-6 boxes representing core components of your system. You should identify enough components that are needed to solve the actual problem from end-to-end.

For Twitter, at a high level, we would need multiple application servers to serve all the read/write requests with load balancers in front of them for traffic distributions. If we're assuming that we'll have a lot more read traffic (as compared to write), we can decide to have separate servers for handling these scenarios. On the backend, we need an efficient database that can store all the tweets and can support a huge number of reads. We would also need a distributed file storage system for storing photos and videos.

Step 6: Detailed design

Dig deeper into 2-3 components; interviewers feedback should always guide you towards which parts of the system she wants you to explain further. You should be able to provide different approaches, their pros and cons, and why would you choose one? Remember there is no single answer, the only thing important is to consider tradeoffs between different options while keeping system constraints in mind.

- Since we'll be storing a huge amount of data, how should we partition our data to distribute it to multiple databases? Should we try to store all the data of a user on the same database? What issue can it cause?
- How would we handle hot users, who tweet a lot or follow lots of people?
- Since user's timeline will contain most recent (and relevant) tweets, should we try to store our data in such a way that is optimized to scan latest tweets?
- How much and at which layer should we introduce cache to speed things up?
- What components need better load balancing?

Step 7: Identifying and resolving bottlenecks

Try to discuss as many bottlenecks as possible and different approaches to mitigate them.

- Is there any single point of failure in our system? What are we doing to mitigate it?
- Do we've enough replicas of the data so that if we lose a few servers, we can still serve our users?
- Similarly, do we've enough copies of different services running, such that a few failures will not cause total system shutdown?

- How are we monitoring the performance of our service? Do we get alerts whenever critical components fail or their performance degrade?

In summary, preparation and being organized during the interview are the keys to be successful in system design interviews.

Designing a URL Shortening service like TinyURL

1. Why do we need URL shortening?

URL shortening is used to create shorter aliases for long URLs. Users are redirected to the original URL when they hit these aliases. A shorter version of any URL would save a lot of space whenever we use it e.g., when printing or tweeting as tweets have a character limit.

For example, if we shorten this page through TinyURL:

https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904/

We would get:
http://tinyurl.com/jlg8zpc

The shortened URL is nearly 1/3rd of the size of the actual URL.

URL shortening is used for optimizing links across devices, tracking individual links to analyze audience and campaign performance, and hiding affiliated original URLs, etc.

If you haven't used tinyurl.com before, please try creating a new shortened URL and spend some time going through different options their service offers. This will help you a lot in understanding this chapter better.

2. Requirements and Goals of the System

Our URL shortener system should meet the following requirements:

Functional Requirements:

1. Given a URL, our service should generate a shorter and unique alias of it.
2. When users access a shorter URL, our service should redirect them to the original link.
3. Users should optionally be able to pick a custom alias for their URL.

4. Links will expire after a specific timespan automatically; users should also be able to specify expiration time.

Non-Functional Requirements:

1. The system should be highly available. This is required because if our service is down, all the URL redirections will start failing.
2. URL redirection should happen in real-time with minimum latency.
3. Shortened 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.

3. Capacity Estimation and Constraints

Our system would be read-heavy; there would be lots of redirection requests compared to new URL shortenings. Let's assume 100:1 ratio between read and write.

Traffic estimates: If we assume that we would have 500M new URLs shortenings per month, we can expect (100 * 500M => 50B) redirections during the same time. What would be Queries Per Second (QPS) for our system?

New URLs shortenings per second:

500 million / (30 days * 24 hours * 3600 seconds) ~= 200 URLs/s

URLs redirections per second:

50 billion / (30 days * 24 hours * 3600 sec) ~= 19K/s

Storage estimates: Since we expect to have 500M new URLs every month and if we would be keeping these objects for five years; total number of objects we will be storing would be 30 billion.

500 million * 5 years * 12 months = 30 billion

Let's assume that each object we are storing can be of 500 bytes (just a ballpark, we will dig into it later); we would need 15TB of total storage:

30 billion * 500 bytes = 15 TB

Bandwidth estimates: For write requests, since every second we expect 200 new URLs, total incoming data for our service would be 100KB per second.

200 * 500 bytes = 100 KB/s

For read requests, since every second we expect ~19K URLs redirections, total outgoing data for our service would be 9MB per second.

19K * 500 bytes ~= 9 MB/s

Memory estimates: If we want to cache some of the hot URLs that are frequently accessed, how much memory would we need to store them? If we follow the 80-20 rule, meaning 20% of URLs generating 80% of traffic, we would like to cache these 20% hot URLs.

Since we have 19K requests per second, we would be getting 1.7billion requests per day.

19K * 3600 seconds * 24 hours ~= 1.7 billion

To cache 20% of these requests, we would need 170GB of memory.

0.2 * 1.7 billion * 500 bytes ~= 170GB

High level estimates: Assuming 500 million new URLs per month and 100:1 read:write ratio, following is the summary of the high level estimates for our service:

New URLs	200/s
URL redirections	19K/s

Incoming data	100KB/s
Outgoing data	9MB/s
Storage for 5 years	15TB
Memory for cache	170GB

4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definitions of the APIs for creating and deleting URLs:

```
creatURL(api_dev_key, original_url, custom_alias=None user_name=None, expire_date=None)
```

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.
original_url (string): Original URL to be shortened.
custom_alias (string): Optional custom key for the URL.
user_name (string): Optional user name to be used in encoding.
expire_date (string): Optional expiration date for the shortened URL.

Returns: (string)
A successful insertion returns the shortened URL, otherwise, returns an error code.

deleteURL(api_dev_key, url_key)

Where "url_key" is a string representing the shortened URL to be retrieved. A successful deletion returns 'URL Removed'.

How do we detect and prevent abuse? For instance, any service can put us out of business by consuming all our keys in the current design. To prevent abuse, we can limit users through their api_dev_key, how many URL they can create or access in a certain time.

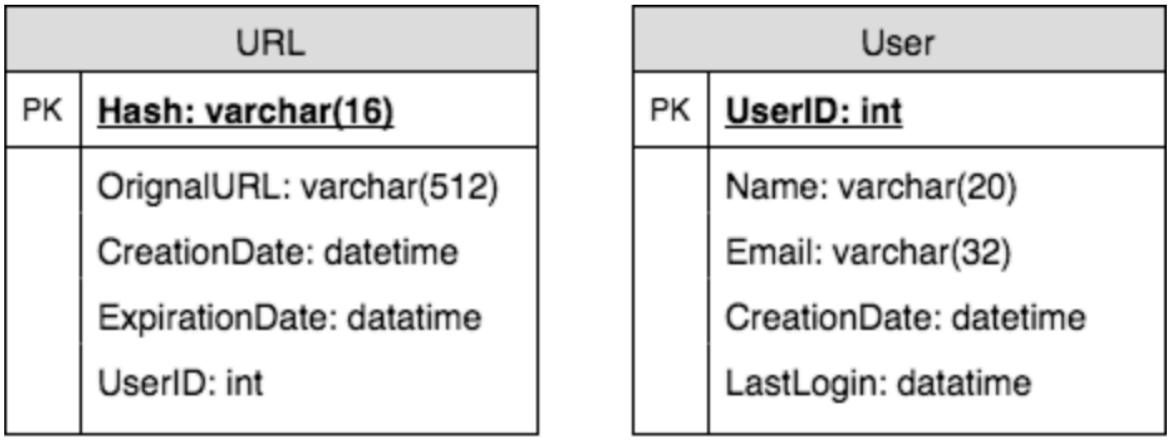
5. Database Design

A few observations about nature of the data we are going to store:

1. We need to store billions of records.
2. Each object we are going to store is small (less than 1K).
3. There are no relationships between records, except if we want to store which user created what URL.
4. Our service is read-heavy.

Database Schema:

We would need two tables, one for storing information about the URL mappings and the other for users' data.



What kind of database should we use? Since we are likely going to store billions of rows and we don't need to use relationships between objects – a NoSQL key-value store like Dynamo or Cassandra is a better choice, which would

also be easier to scale. Please see SQL vs NoSQL for more details. If we choose NoSQL, we cannot store UserID in the URL table (as there are no foreign keys in NoSQL), for that we would need a third table which will store the mapping between URL and the user.

6. Basic System Design and Algorithm

The problem we are solving here is to generate a short and unique key for the given URL. In the above-mentioned example, the shortened URL we got was: "http://tinyurl.com/jlg8zpc", the last six characters of this URL is the short key we want to generate. We'll explore two solutions here:

a. Encoding actual URL

We can compute a unique hash (e.g., MD5 or SHA256, etc.) of the given URL. The hash can then be encoded for displaying. This encoding could be base36 ([a-z ,0-9]) or base62 ([A-Z, a-z, 0-9]) and if we add '-' and '.', we can use base64 encoding. A reasonable question would be; what should be the length of the short key? 6, 8 or 10 characters?

Using base64 encoding, a 6 letter long key would result in 64^6 ~= 68.7 billion possible strings
Using base64 encoding, an 8 letter long key would result in 64^8 ~= 281 trillion possible strings

With 68.7B unique strings, let's assume for our system six letters keys would suffice.

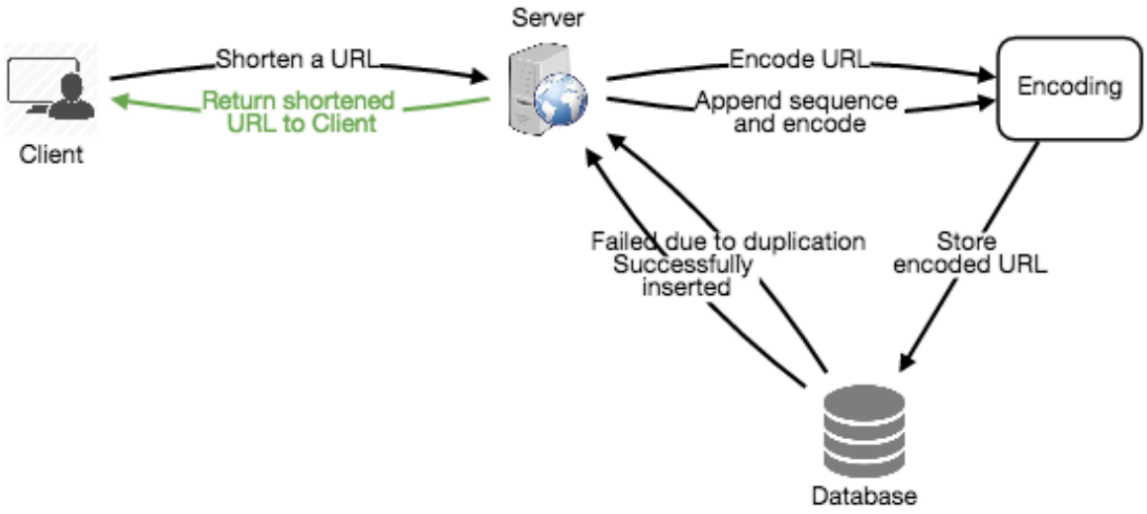
If we use the MD5 algorithm as our hash function, it'll produce a 128-bit hash value. After base64 encoding we'll get a string having more than 20 characters, how will we chose our key then? We can take the first 6 (or 8) letters for the key. This could result in key duplication though, upon which we can choose some other characters out of the encoding string or swap some characters.

What are different issues with our solution? We have the following couple of problems with our encoding scheme:

1. If multiple users enter the same URL, they can get the same shortened URL, which is not acceptable.
2. What if parts of the URL are URL-encoded?
   e.g., http://www.educative.io/distributed.php?id=design,
   and http://www.educative.io/distributed.php%3Fid%3Ddesign are identical except for the URL encoding.

Workaround for the issues: We can append an increasing sequence number to each input URL to make it unique and then generate a hash of it. We don't need to store this sequence number in the databases, though. Possible problems with this approach could be how big this sequence number would be, can it overflow? Appending an increasing sequence number will impact the performance of the service too.

Another solution could be, to append user id (which should be unique) to the input URL. However, if the user has not signed in, we can ask the user to choose a uniqueness key. Even after this if we have a conflict, we have to keep generating a key until we get a unique one.



Request flow for shortening of a URL

Generating keys offline

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

Can concurrency cause problems? As soon as a key is used, it should be marked in the database so that it doesn't get 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 so that whenever a server needs them, it can quickly provide them. For simplicity, as soon as KGS loads some keys in memory, it can move them to used keys table. This way we can make sure each server gets unique keys. If KGS dies before assigning all the loaded keys to some server, we will be wasting those keys, which we can ignore given a 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 to) the data structure holding the keys before removing keys from it and giving them to a server.

What would be the key-DB size? With base64 encoding, we can generate 68.7B unique six letters keys. If we need one byte to store one alpha-numeric character, we can store all these keys in:

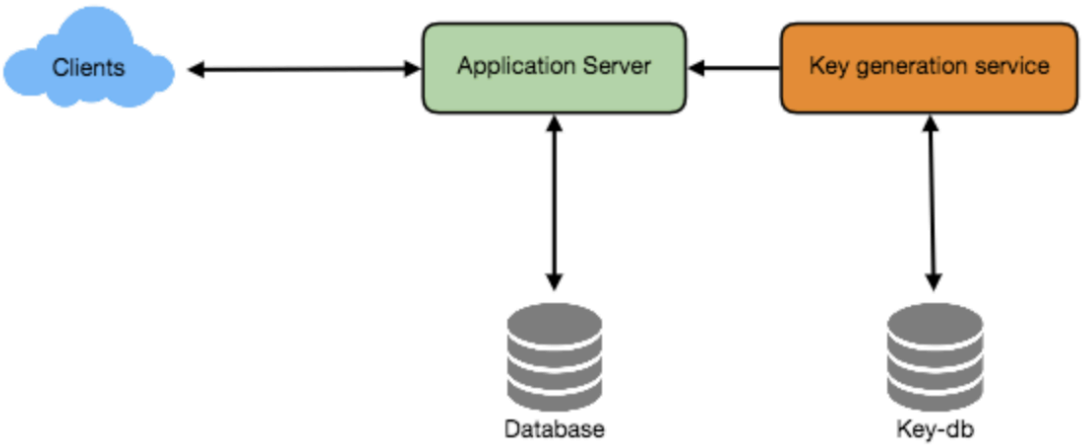
6 (characters per key) * 68.7B (unique keys) => 412 GB.

Isn't KGS the 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.

How would we perform a key lookup? We can look up the key in our database or key-value store to get the full URL. If it's present, issue a "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 a "HTTP 404 Not Found" status, or redirect the user back to the homepage.

Should we impose size limits on custom aliases? Since 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, so that we have a consistent URL database. Let's assume users can specify maximum 16 characters long customer key (as reflected in the above database schema).



High level system design for URL shortening

7. Data Partitioning and Replication

To scale out our DB, we need to partition it so that it can store information about billions of URL. We need to come up with a partitioning scheme that would divide and store our data to different DB servers.

a. Range Based Partitioning: We can store URLs in separate partitions based on the first letter of the URL or the hash key. Hence we save all the URLs starting with letter 'A' in one partition and those that start with letter 'B' into another partition and so on. This approach is called range based partitioning. We can even combine certain less frequently occurring letters into one database partition. We should come up with this partitioning scheme statically so that we can always store/find a file in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers, for instance; if we decide to put all URLs starting with letter 'E' into a DB partition, but later we realize that we have too many URLs that start with letter 'E', which we can't fit into one DB partition.

b. Hash-Based Partitioning: In this scheme, we take a hash of the object we are storing, and based on this hash we figure out the DB partition to which this object should go. In our case, we can take the hash of the 'key' or the actual URL to determine the partition to store the file. Our hashing function will randomly distribute URLs into different partitions, e.g., our hashing function can always

map any key to a number between [1...256], and this number would represent the partition to store our object.

This approach can still lead to overloaded partitions, which can be solved by using Consistent Hashing.

8. Cache

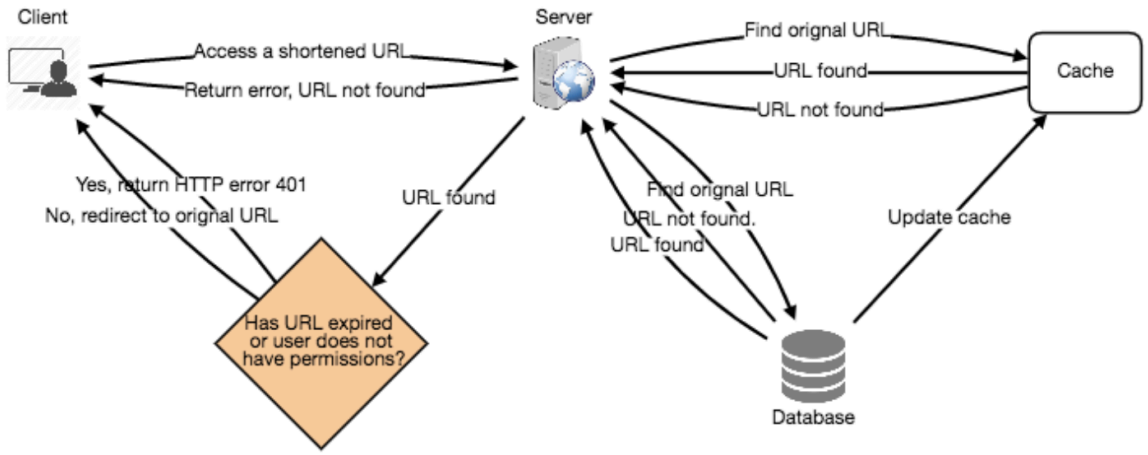
We can cache URLs that are frequently accessed. We can use some off-the-shelf solution like Memcache, that can store full URLs with their respective hashes. The application servers, before hitting backend storage, can quickly check if the cache has desired URL.

How much cache should we have? We can start with 20% of daily traffic and based on clients' usage pattern we can adjust how many cache servers we need. As estimated above we need 170GB 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, or we can choose to 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 which URLs are accessed recently.

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

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



Request flow for accessing a shortened URL

9. Load Balancer (LB)

We can add 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

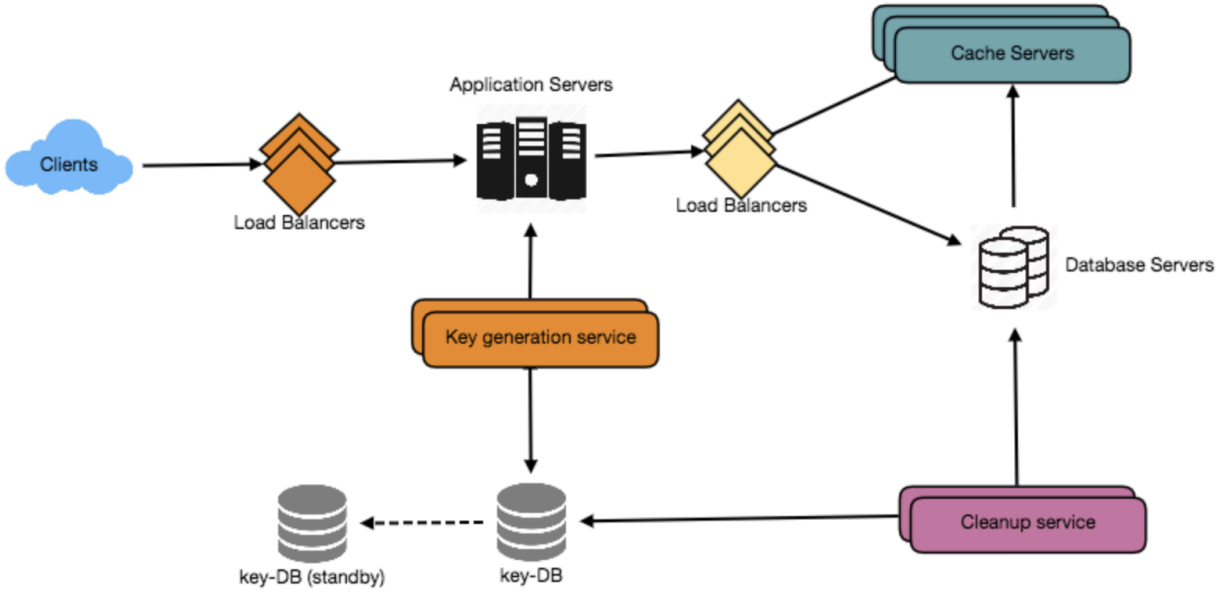
Initially, a simple Round Robin approach can be adopted; that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is if a server is dead, LB will take it out of the rotation and will stop sending any traffic to it. A problem with Round Robin LB is, it won't take 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 backend server about its load and adjusts traffic based on that.

10. 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 actively search for expired links to remove them, it would put a lot of pressure on our database. We can slowly remove expired links and do a lazy cleanup too. Our

service will make sure 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 can be 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.



11. Telemetry

How many times a short URL has been used, what were user locations, etc.? How would we store these statistics? If it is part of a DB row that gets updated on each view, what will happen when a popular URL is slammed with a large number of concurrent requests?

We can have statistics about the country of the visitor, date and time of access, web page that refers the click, browser or platform from where the page was accessed and more.

12. Security and Permissions

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

We can store 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 try 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'), and the columns will store the UserIDs of those users that have permissions to see the URL.

Designing Pastebin

1. What is Pastebin?

Pastebin 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 network quickly, as users would just 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.

2. Requirements and Goals of the System

Our Pastebin service should meet the following requirements:

Functional Requirements:

1. Users should be able to 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 expire after a specific timespan automatically; users should also be able to 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 be able to 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 times a paste was accessed?
2. Our service should also be accessible through REST APIs by other services.

3. Some Design Considerations

Pastebin shares some requirements with 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 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 that 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, so that we have a consistent URL database.

4. Capacity Estimation and Constraints

Our services would be read heavy; there will be more read requests compared to new Pastes creation. We can assume 5:1 ratio between read and write.

Traffic estimates: Pastebin services are not expected to have traffic similar to Twitter or Facebook, let's assume here that we get one million new pastes added to our system every day. This leaves us with five million reads per day.

New Pastes per second:

$$1M / (24 \text{ hours} * 3600 \text{ seconds}) \sim= 12 \text{ pastes/sec}$$

Paste reads per second:

$$5M / (24 \text{ hours} * 3600 \text{ seconds}) \sim= 58 \text{ reads/sec}$$

Storage estimates: Users can upload maximum 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 10KB.

At this rate, we will be storing 10GB of data per day.

$$1M * 10KB => 10 \text{ GB/day}$$

If we want to store this data for ten years, we would need the total storage capacity of 36TB.

With 1M pastes every day we will have 3.6 billion Pastes in 10 years. We need to generate and store keys to uniquely identify these pastes. If we use base64 encoding ([A-Z, a-z, 0-9, ., -]) we would need six letters strings:

64^6 ~= 68.7 billion unique strings

If it takes one byte to store one character, total size required to store 3.6B keys would be:

3.6B * 6 => 22 GB

22GB is negligible compared to 36TB. 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 47TB.

Bandwidth estimates: For write requests, we expect 12 new pastes per second, resulting in 120KB of ingress per second.

12 * 10KB => 120 KB/s

As for read request, we expect 58 requests per second. Therefore, total data egress (sent to users) will be 0.6 MB/s.

58 * 10KB => 0.6 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 80-20 rule, meaning 20% of hot pastes generate 80% of traffic, we would like to cache these 20% pastes

Since we have 5M read requests per day, to cache 20% of these requests, we would need:

0.2 * 5M * 10KB ~= 10 GB

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definitions of the APIs to create/retrieve/delete Pastes:

addPaste(api_dev_key, paste_data, custom_url=None user_name=None, paste_name=None, expire_date=None)

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.
paste_data (string): Textual data of the paste.

custom_url (string): Optional custom URL.
user_name (string): Optional user name to be used to generate URL.
paste_name (string): Optional name of the paste expire_date (string): Optional expiration date for the paste.

Returns: (string)
A successful insertion returns the URL through which the paste can be accessed, otherwise, returns an error code.

Similarly, we can have retrieve and delete Paste APIs:

getPaste(api_dev_key, api_paste_key)

Where "api_paste_key" is a string representing the Paste Key of the paste to be retrieved. This API will return the textual data of the paste.

deletePaste(api_dev_key, api_paste_key)

A successful deletion returns 'true', otherwise returns 'false'.

6. Database Design

A few observations about nature of the data we are going to store:

1. We need to store billions of records.
2. Each metadata object we are going to store would be small (less than 100 bytes).
3. Each paste object we are storing can be of medium size (it can be a few MB).
4. There are no relationships between records, except if we want to store which user created what Paste.
5. Our service is read heavy.

Database Schema:

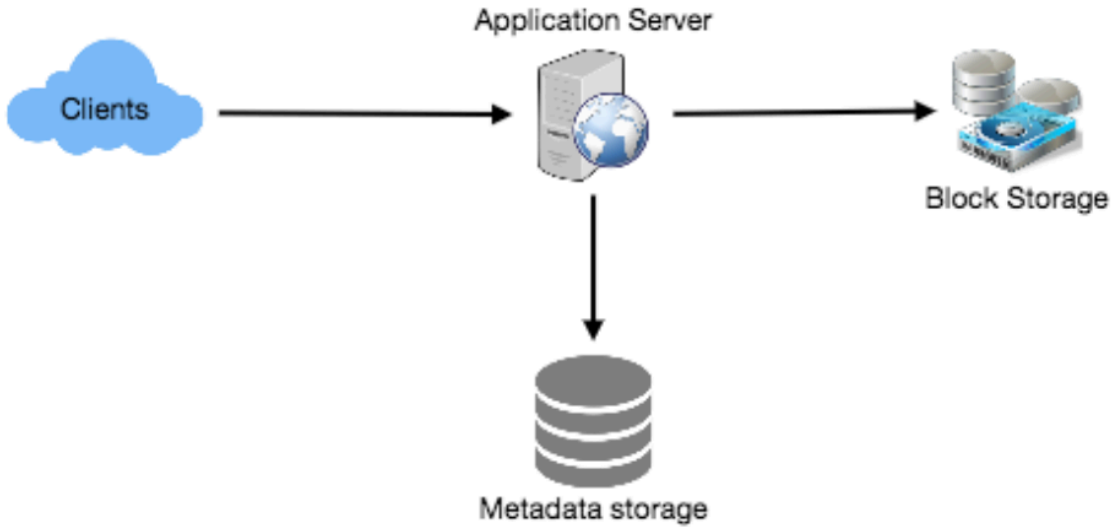
We would need two tables, one for storing information about the Pastes and the other for users' data.

Paste	
PK	**Hash: varchar(16)**
	OrignalURL: varchar(512)
	ContentPath: varchar(512)
	ExpirationDate: datatime
	UserID: int
	CreationDate: datetime

User	
PK	**UserID: int**
	Name: varchar(20)
	Email: varchar(32)
	CreationDate: datetime
	LastLogin: datatime

High Level Design

At a high level, we need an application layer that will serve all the read and write requests. 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 sort of block storage or a database. This division of data will allow us to scale them individually.



High level design for Pastebin

8. Component Design

a. 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 of 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). Whenever we want to store a new paste, we will just 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 will make sure all the keys inserted in key-DB are unique. 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 give 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 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 a huge number of keys we have.

Isn't KGS 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.

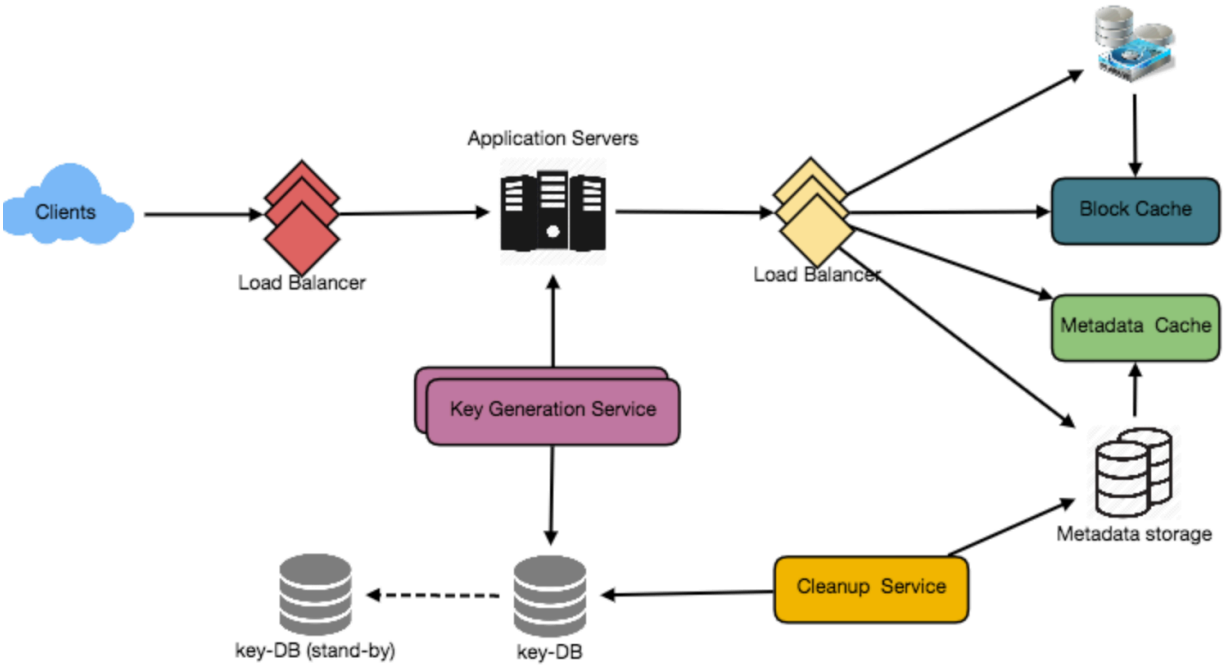
Can each app server cache some keys from key-DB? Yes, this can surely speed things up. Although in this case, if the application server dies before consuming all the keys, we will end up losing those keys. This could be acceptable since we have 68B unique six letters keys, which are a lot more than we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

How to handle a paste read request? Upon receiving a read paste 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.

b. Datastore layer

We can divide our datastore layer into two:

1. Metadata database: We can use a relational database like MySQL or a Distributed Key-Value store like Dynamo or Cassandra.
2. Block storage: We can store our contents in a block storage that could be a distributed file storage or an SQL-like database. Whenever we feel like hitting our full capacity on content storage, we can easily increase it by adding more servers.



Detailed component design for Pastebin

9. Purging or DB Cleanup

Please see Designing a URL Shortening service.

10. Data Partitioning and Replication

Please see Designing a URL Shortening service.

11. Cache and Load Balancer

Please see Designing a URL Shortening service.

12. Security and Permissions

Please see Designing a URL Shortening service.

Designing Instagram

1. Why Instagram?

Instagram is a social networking service, which enables its users to upload and share their pictures and videos with other users. Users can share either publicly or privately, as well as through a number of other social networking platforms, such as Facebook, Twitter, Flickr, and Tumblr.

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

2. Requirements and Goals of the System

We will 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 be able to generate and display a user's timeline 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 timeline 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 photo/video uploaded should not be lost.

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

3. Some Design Considerations

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

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

4. Capacity Estimation and Constraints

- Let's assume we have 300M total users, with 1M daily active users.
- 2M new photos every day, 23 new photos every second.
- Average photo file size => 200KB
- Total space required for 1 day of photos

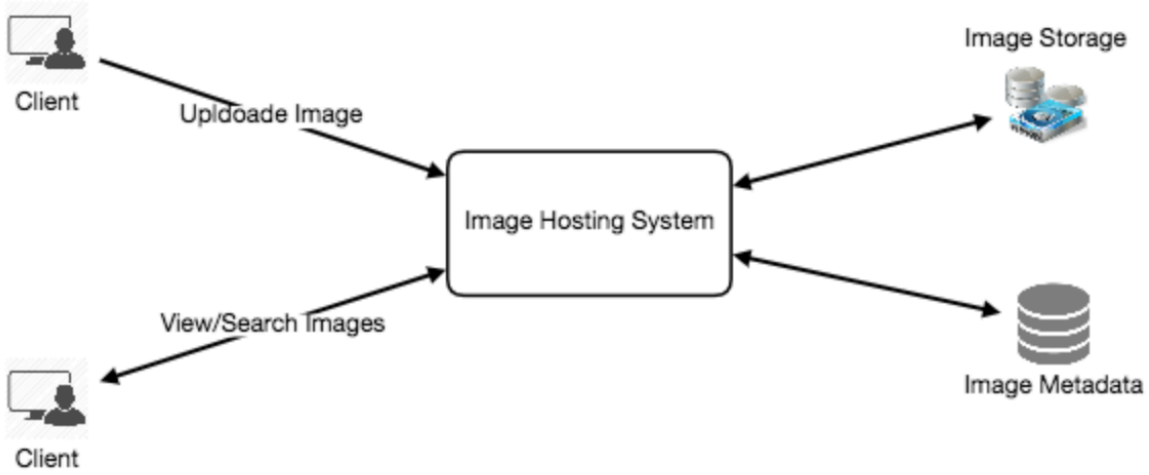
2M * 200KB => 400 GB

- Total space required for 5 years:

400GB * 365 (days a year) * 5 (years) ~= 712 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 block storage servers to store photos and also some database servers to store metadata information about the photos.



6. Database Schema

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

One simple 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 benefits offered by NoSQL. All the metadata related to photos can go to a table, where the 'key' would be the 'PhotoID' and the 'value' would be an object containing PhotoLocation, UserLocation, CreationTimestamp, etc.

We also need to store relationships between users and photos, to know who owns which photo. Another relationship we would need to store is the list of people a user follows. For both of these tables, 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.

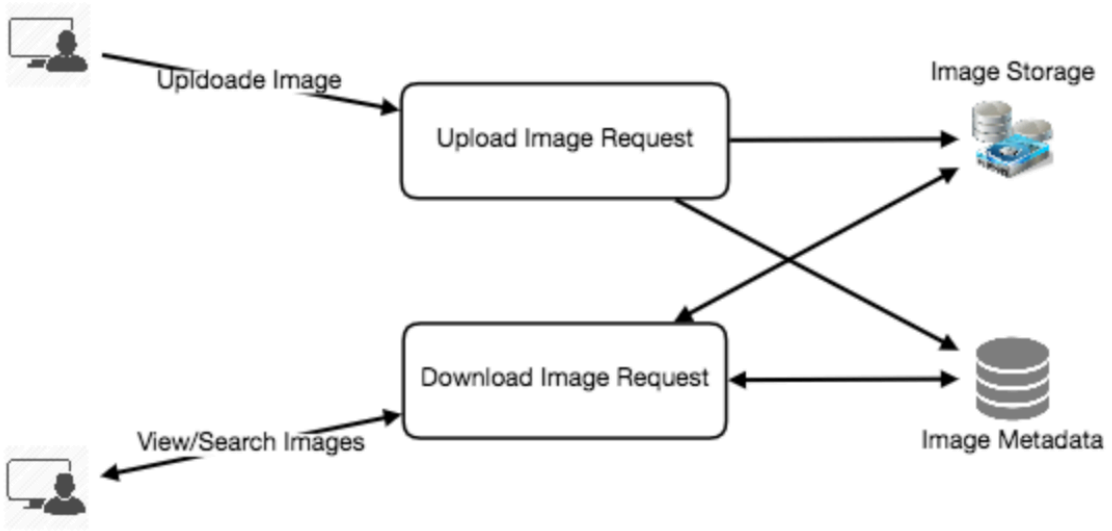
Cassandra or key-value stores in general, always maintain a certain number of replicas to offer reliability. Also, in such data stores, deletes don't get applied instantly, data is retained for certain days (to support undeleting) before getting removed from the system permanently.

7. Component Design

Writes or photo uploads could be slow as they have to go to the disk, whereas reads could be faster, especially, if they are being served from cache.

Uploading users can consume all the connections, as uploading would be a slower process. This means reads cannot be served if the system gets busy with all the write requests. To handle this bottleneck we can split out read and writes into separate services. Since web servers have connection limit, we should keep this thing in mind before designing our system. Let's assume if a web server can have maximum 500 connections at any time, which means it can't have more than 500 concurrent uploads or reads. This guides us to have separate dedicated servers for reads and writes so that uploads don't hog the system.

Separating image read and write requests will also allow us to scale or optimize each of them independently.



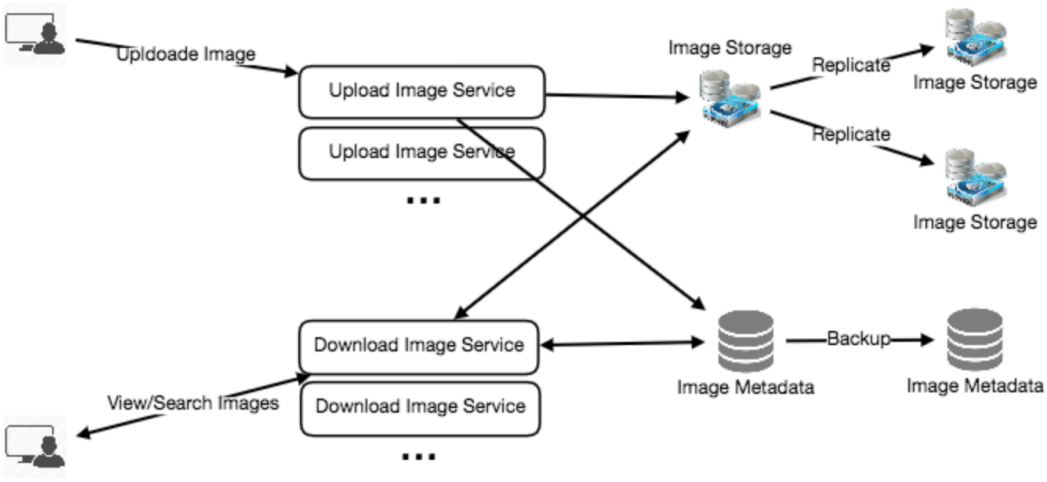
8. Reliability and Redundancy

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 image from the other copy present on a different storage server.

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

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

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.



9. 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 4TB, we will have 712/4 => 178 shards. Let's assume for future growths we keep 200 shards.

So we will find the shard number by UserID % 200 and then store the data there. To uniquely identify any photo in our system, we can append shard number with each PhotoID.

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

What are different issues with this partitioning scheme?

1. How would we handle hot users? Several people follow such hot users, and any photo they upload is seen by a lot of other people.
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 pictures of a user on one shard can cause issues like unavailability of all of the user's data if that shard is down or higher latency if it is serving high load etc.

b. Partitioning based on PhotoID If we can generate unique PhotoIDs first and then find shard number through PhotoID % 200, this can solve above problems. We would not need to append ShardID with PhotoID in this case as PhotoID will itself be unique throughout the system.

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

Wouldn't this key generating DB be a single point of failure? Yes, it will be. A workaround for that could be, we can define two such databases, with one generating even numbered IDs and the other odd numbered. For MySQL 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 down time. Both these servers could be out of sync with one generating more keys than the other, but this will not cause any issue in our system. We can extend this design by defining separate ID tables for Users, Photo-Comments or other objects present in our system.

Alternately, we can implement a key generation scheme similar to what we have discussed in Designing a URL Shortening service like TinyURL.

How can we plan for future growth of our system? We can have a large number of logical partitions to accommodate future data growth, such that, in the beginning, multiple logical partitions reside on a single physical database server.

Since each database server can have multiple database instances on it, we can have separate databases for each logical partition on any server. So whenever we feel that a certain 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 just have to update the config file to announce the change.

10. Ranking and Timeline Generation

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

For simplicity, let's assume we need to fetch top 100 photos for a user's timeline. Our application server will first get a list of people the user follows and then fetches metadata info of latest 100 photos from each user. In the final step, the server will submit all these photos to our ranking algorithm which will determine the top 100 photos (based on recency, likeness, etc.) to be returned 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 timeline and store it in a separate table.

Pre-generating the timeline: We can have dedicated servers that are continuously generating users' timelines and storing them in a 'UserTimeline' table. So whenever any user needs the latest photos for their timeline, we will simply query this table and return the results to the user.

Whenever these servers need to generate the timeline of a user, they will first query the UserTimeline table to see what was the last time the timeline was generated for that user. Then, new timeline data will be generated from that time onwards (following the abovementioned steps).

What are the different approaches for sending timeline data to the users?

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

2. Push: Servers can push new data to the users as soon as it is available. To efficiently manage this, users have to maintain a Long Poll request with the server for receiving the updates. A possible problem with this approach is when a

user has a lot of follows 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 with high followings to pull based model and only push data to those users who have a few hundred (or thousand) follows. Another approach could be that the server pushes updates to all the users not more than a certain frequency, letting users with a lot of follows/updates to regularly pull data.

For a detailed discussion about timeline generation, take a look at Designing Facebook's Newsfeed.

11. Timeline Creation with Sharded Data

To create the timeline for any given user, one of the most important requirements is to fetch latest photos from all people the user follows. For this, we need to have a mechanism to sort photos on their time of creation. This can be done efficiently if we can make photo creation time part of the PhotoID. Since we will have a primary index on PhotoID, it will be quite quick to find 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 seconds and the second part will be an auto-incrementing sequence. So to make a new PhotoID, we can take the current epoch time and append an auto incrementing ID from our key generating DB. We can figure out shard number from this PhotoID (PhotoID % 200) and store the photo there.

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

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

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

We will discuss more details about this technique under 'Data Sharding' in Designing Twitter.

12. Cache and Load balancing

To serve globally distributed users, our service needs a massive-scale photo delivery system. 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 database can quickly check if the cache has desired rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system. Under this policy, we discard the least recently viewed row first.

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

Designing Dropbox

Let's design a file hosting service like Dropbox or Google Drive. Cloud file storage enables users to store their data on remote servers. Usually, these servers are maintained by cloud storage providers and made available to users over a network (typically through the Internet). Users pay for their cloud data storage on a monthly basis. Similar Services: OneDrive, Google Drive Difficulty Level: Medium

1. Why Cloud Storage?

Cloud file storage services have become very popular recently as they simplify the storage and exchange of digital resources among multiple devices. The shift from using single personal computers to using multiple devices with different platforms and operating systems such as smartphones and tablets and their portable access from various geographical locations at any time is believed to be accountable for the huge popularity of cloud storage services. Some of the top benefits of such services are:

Availability: The motto of cloud storage services is to have data availability anywhere anytime. Users can access their files/photos from any device whenever and wherever they like.

Reliability and Durability: Another benefit of cloud storage is that it offers 100% reliability and durability of data. Cloud storage ensures that users will never lose their data, by keeping multiple copies of the data stored on different geographically located servers.

Scalability: Users will never have to worry about getting out of storage space. With cloud storage, you have unlimited storage as long as you are ready to pay for it.

If you haven't used dropbox.com before, we would highly recommend creating an account there and uploading/editing a file and also going through different options their service 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 Cloud Storage system? Here are the top-level requirements for our system:

1. Users should be able to upload and download their files/photos from any device.
2. Users should be able to share files or folders with other users.
3. Our service should support automatic synchronization between devices, i.e., after updating a file on one device, it should get synchronized on all devices.
4. The system should support storing large files up to a GB.
5. ACID-ity is required. Atomicity, Consistency, Isolation and Durability of all file operations should be guaranteed.
6. Our system should support offline editing. Users should be able to add/delete/modify files while offline, and as soon as they come online, all their changes should be synced to the remote servers and other online devices.

Extended Requirements

- The system should support snapshotting of the data, so that users can go back to any version of the files.

3. Some Design Considerations

- We should expect huge read and write volumes.
- Read to write ratio is expected to be nearly the same.
- Internally, files can be stored in small parts or chunks (say 4MB), this can provide a lot of benefits e.g. all failed operations shall only be retried for smaller parts of a file. If a user fails to upload a file, then only the failing chunk will be retried.
- We can reduce the amount of data exchange by transferring updated chunks only.
- By removing duplicate chunks, we can save storage space and bandwidth usage.
- Keeping a local copy of the metadata (file name, size, etc.) with the client can save us a lot of round trips to the server.
- For small changes, clients can intelligently upload the diffs instead of the whole chunk.

4. Capacity Estimation and Constraints

- Let's assume that we have 500M total users, and 100M daily active users (DAU).
- Let's assume that on average each user connects from three different devices.
- On average if a user has 200 files/photos, we will have 100 billion total files.
- Let's assume that average file size is 100KB, this would give us ten petabytes of total storage.

100B * 100KB => 10PB

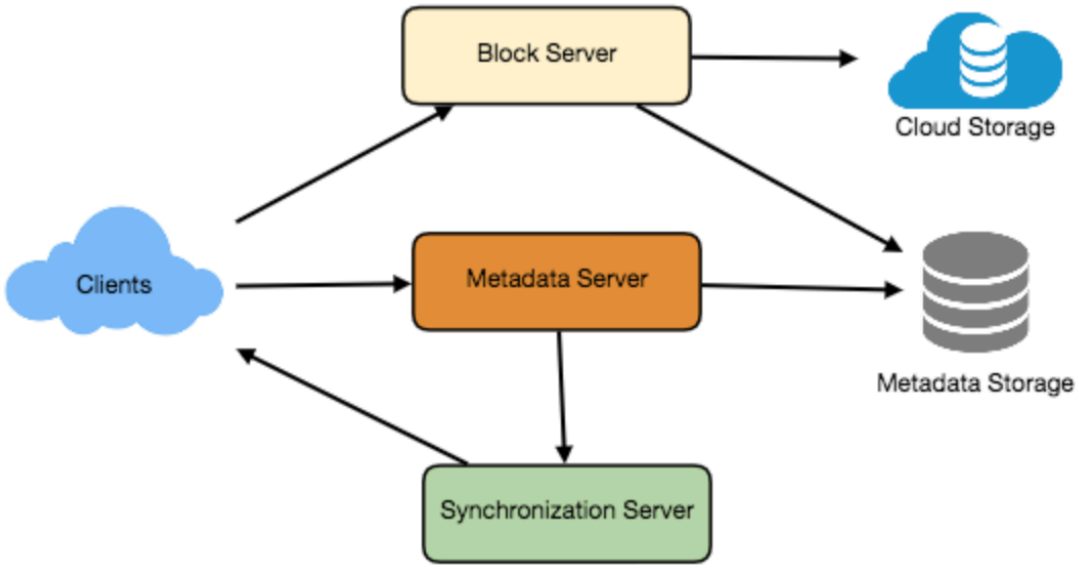
- Let's also assume that we will have one million active connections per minute.

5. High Level Design

The user will specify a folder as the workspace on their device. Any file/photo/folder placed in this folder will be uploaded to the cloud, and whenever a file is modified or deleted, it will be reflected in the same way in the cloud storage. The user can specify similar workspaces on all their devices and any modification done on one device will be propagated to all other devices to have the same view of the workspace everywhere.

At a high level, we need to store files and their metadata information like File Name, File Size, Directory, etc., and who this file is shared with. So, we need some servers that can help the clients to upload/download files to Cloud Storage and some servers that can facilitate updating metadata about files and users. We also need some mechanism to notify all clients whenever an update happens so they can synchronize their files.

As shown in the diagram below, Block servers will work with the clients to upload/download files from cloud storage, and Metadata servers will keep metadata of files updated in a SQL or NoSQL database. Synchronization servers will handle the workflow of notifying all clients about different changes for synchronization.



High level design for Dropbox

6. Component Design

Let's go through the major components of our system one by one:

a. Client

The Client Application monitors the workspace folder on user's machine and syncs all files/folders in it with the remote Cloud Storage. The client application will work with the storage servers to upload, download and modify actual files to backend Cloud Storage. The client also interacts with the remote Synchronization Service to handle any file metadata updates e.g. change in the file name, size, modification date, etc.

Here are some of the essential operations of the client:

1. Upload and download files.
2. Detect file changes in the workspace folder.
3. Handle conflict due to offline or concurrent updates.

How do we handle file transfer efficiently? As mentioned above, we can break each file into smaller chunks so that we transfer only those chunks that are modified and not the whole file. Let's say we divide each file into fixed size of

4MB chunks. We can statically calculate what could be an optimal chunk size based on 1) Storage devices we use in the cloud to optimize space utilization and Input/output operations per second (IOPS) 2) Network bandwidth 3) Average file size in the storage etc. In our metadata, we should also keep a record of each file and the chunks that constitute it.

Should we keep a copy of metadata with Client? Keeping a local copy of metadata not only enable us to do offline updates but also saves a lot of round trips to update remote metadata.

How can clients efficiently listen to changes happening on other clients? One solution could be that the clients periodically check with the server if there are any changes. The problem with this approach is that we will have a delay in reflecting changes locally as clients will be checking for changes periodically compared to server notifying whenever there is some change. If the client frequently checks the server for changes, it will not only be wasting bandwidth, as the server has to return empty response most of the time but will also be keeping the server busy. Pulling information in this manner is not scalable too.

A solution to above problem could be to use HTTP long polling. With long polling, the client requests information from the server with the expectation that the 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 an HTTP/S response to the client, completing the open HTTP/S Request. Upon receipt of the server response, the client can immediately issue another server request for future updates.

Based on the above considerations we can divide our client into following four parts:

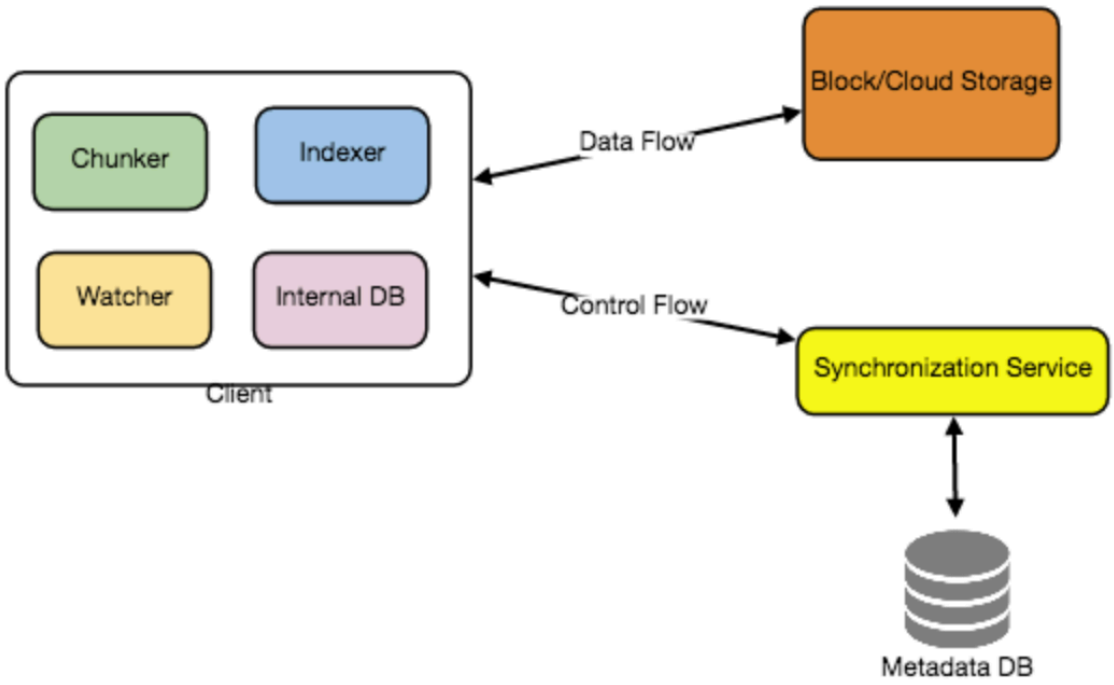
I. **Internal Metadata Database** will keep track of all the files, chunks, their versions, and their location in the file system.

II. **Chunker** will split the files into smaller pieces called chunks. It will also be responsible for reconstructing a file from its chunks. Our chunking algorithm will detect the parts of the files that have been modified by the user and only transfer those parts to the Cloud Storage; this will save us bandwidth and synchronization time.

III. **Watcher** will monitor the local workspace folders and notify the Indexer (discussed below) of any action performed by the users, e.g., when users create,

delete, or update files or folders. Watcher also listens to any changes happening on other clients that are broadcasted by Synchronization service.

IV. **Indexer** will process the events received from the Watcher and update the internal metadata database with information about the chunks of the modified files. Once the chunks are successfully submitted/downloaded to the Cloud Storage, the Indexer will communicate with the remote Synchronization Service to broadcast changes to other clients and update remote metadata database.



How should clients handle slow servers? Clients should exponentially back-off if the server is busy/not-responding. Meaning, if a server is too slow to respond, clients should delay their retries, and this delay should increase exponentially.

Should mobile clients sync remote changes immediately? Unlike desktop or web clients, that check for file changes on a regular basis, mobile clients usually sync on demand to save user's bandwidth and space.

b. Metadata Database

The Metadata Database is responsible for maintaining the versioning and metadata information about files/chunks, users, and workspaces. The Metadata Database can be a relational database such as MySQL, or a NoSQL database service such as DynamoDB. Regardless of the type of the database, the Synchronization Service should be able to provide a consistent view of the files using a database, especially if more than one user work with the same file simultaneously. 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 Synchronization Service in case we opt for this kind of databases. However, using a relational database can simplify the implementation of the Synchronization Service as they natively support ACID properties.

Metadata Database should be storing information about following objects:

1. Chunks
2. Files
3. User
4. Devices
5. Workspace (sync folders)

c. Synchronization Service

The Synchronization Service is the component that processes file updates made by a client and applies these changes to other subscribed clients. It also synchronizes clients' local databases with the information stored in the remote Metadata DB. The Synchronization Service is the most important part of the system architecture due to its critical role in managing the metadata and synchronizing users' files. Desktop clients communicate with the Synchronization Service to either obtain updates from the Cloud Storage or send files and updates to the Cloud Storage and potentially other users. If a client was offline for a period, it polls the system for new updates as soon as it becomes online. When the Synchronization Service receives an update request, it checks with the Metadata Database for consistency and then proceeds with the update. Subsequently, a notification is sent to all subscribed users or devices to report the file update.

The Synchronization Service should be designed in such a way to transmit less data between clients and the Cloud Storage to achieve better response time. To meet this design goal, the Synchronization Service can employ a differencing algorithm to reduce the amount of the data that needs to be synchronized.

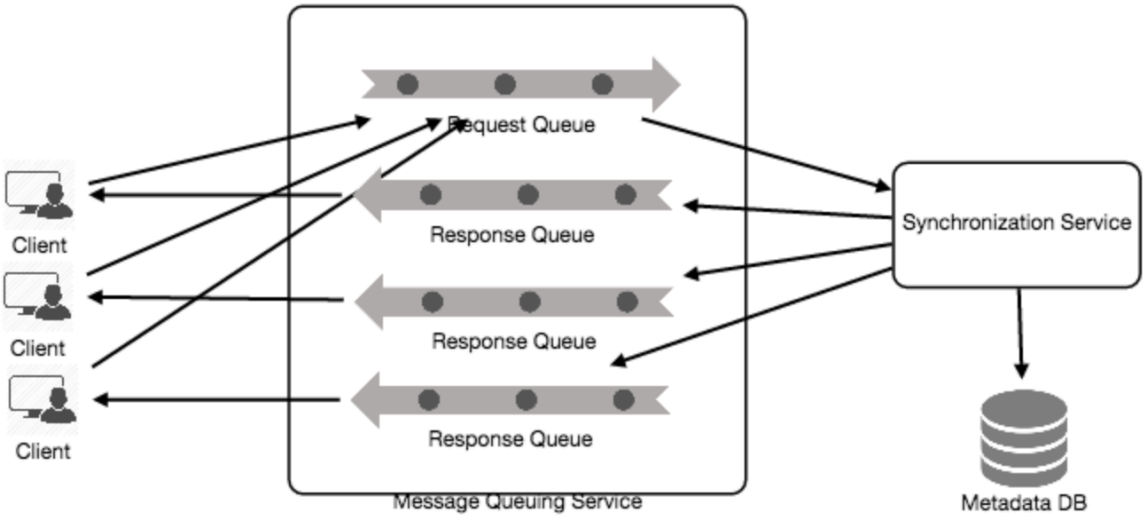
Instead of transmitting entire files from clients to the server or vice versa, we can just transmit the difference between two versions of a file. Therefore, only the part of the file that has been changed is transmitted. This also decreases bandwidth consumption and cloud data storage for the end user. As described above we will be dividing our files into 4MB chunks and will be transferring modified chunks only. Server and clients can calculate a hash (e.g., SHA-256) to see whether to update the local copy of a chunk or not. On server if we already have a chunk with a similar hash (even from another user) we don't need to create another copy, we can use the same chunk. This is discussed in detail later under Data Deduplication.

To be able to provide an efficient and scalable synchronization protocol we can consider using a communication middleware between clients and the Synchronization Service. The messaging middleware should provide scalable message queuing and change notification to support a high number of clients using pull or push strategies. This way, multiple Synchronization Service instances can receive requests from a global request Queue, and the communication middleware will be able to balance their load.

d. Message Queuing Service

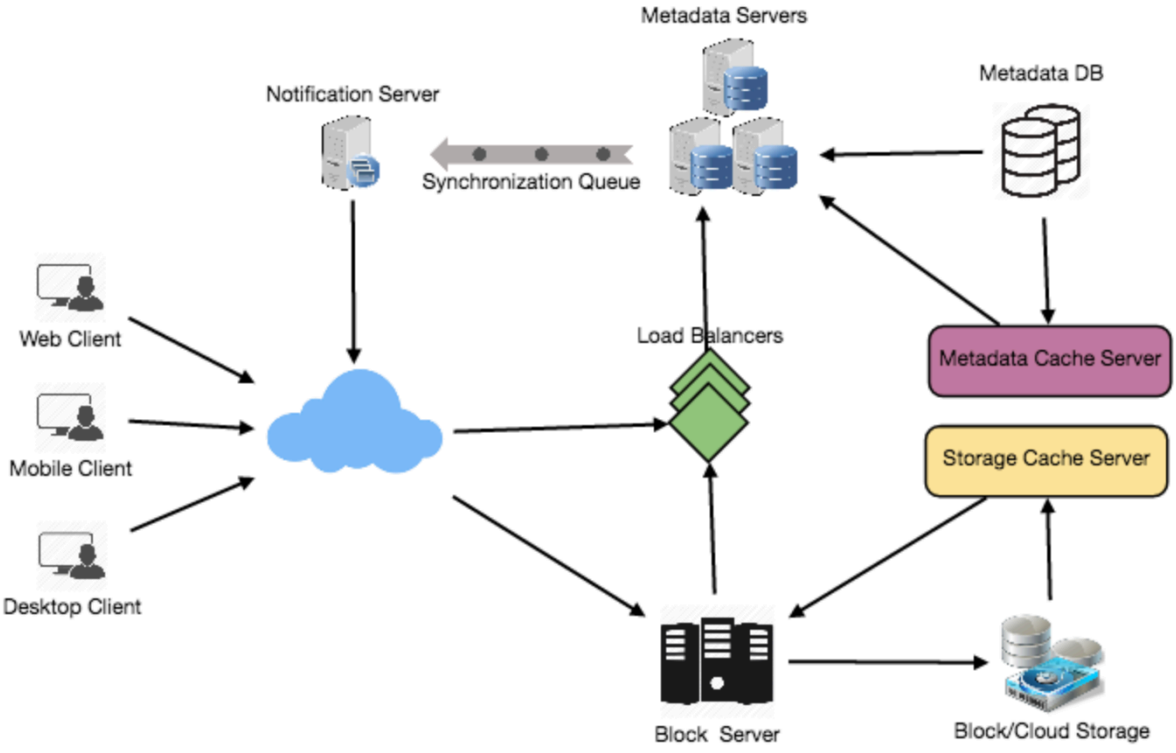
An important part of our architecture is a messaging middleware that should be able to handle a substantial number of requests. A scalable Message Queuing Service that supports asynchronous message-based communication between clients and the Synchronization Service instances best fits the requirements of our application. The Message Queuing Service supports asynchronous and loosely coupled message-based communication between distributed components of the system. The Message Queuing Service should be able to efficiently store any number of messages in a highly available, reliable and scalable queue.

Message Queuing Service will implement two types of queues in our system. The Request Queue is a global queue, and all client will share it. Clients' requests to update the Metadata Database will be sent to the Request Queue first, from there Synchronization Service will take it to update metadata. The Response Queues that correspond to individual subscribed clients are responsible for delivering the update messages to each client. Since a message will be deleted from the queue once received by a client, we need to create separate Response Queues for each subscribed client to share update messages.



e. Cloud/Block Storage

Cloud/Block Storage stores chunks of files uploaded by the users. Clients directly interact with the storage to send and receive objects from it. Separation of the metadata from storage enables us to use any storage either in cloud or in-house.



Detailed component design for Dropbox

7. File Processing Workflow

The sequence below shows the interaction between the components of the application in a scenario when Client A updates a file that is shared with Client B and C, so they should receive the update too. If the other clients were not online at the time of the update, the Message Queuing Service keeps the update notifications in separate response queues for them until they become online later.

1. Client A uploads chunks to cloud storage.
2. Client A updates metadata and commits changes.
3. Client A gets confirmation, and notifications are sent to Clients B and C about the changes.
4. Client B and C receive metadata changes and download updated chunks.

8. Data Deduplication

Data deduplication is a technique used for eliminating duplicate copies of data to improve storage utilization. It can also be applied to network data transfers to reduce the number of bytes that must be sent. For each new incoming chunk, we

can calculate a hash of it and compare that hash with all the hashes of the existing chunks to see if we already have same chunk present in our storage.

We can implement deduplication in two ways in our system:

a. Post-process deduplication
With post-process deduplication, new chunks are first stored on the storage device, and later some process analyzes the data looking for duplication. The benefit is that clients will not need to wait for the hash calculation or lookup to complete before storing the data, thereby ensuring that there is no degradation in storage performance. Drawbacks of this approach are 1) We will unnecessarily be storing duplicate data, though for a short time, 2) Duplicate data will be transferred consuming bandwidth.

b. In-line deduplication
Alternatively, deduplication hash calculations can be done in real-time as the clients are entering data on their device. If our system identifies a chunk which it has already stored, only a reference to the existing chunk will be added in the metadata, rather than the full copy of the chunk. This approach will give us optimal network and storage usage.

9. Metadata Partitioning

To scale out metadata DB, we need to partition it so that it can store information about millions of users and billions of files/chunks. We need to come up with a partitioning scheme that would divide and store our data to different DB servers.

1. Vertical Partitioning: We can partition our database in such a way that we store tables related to one particular feature on one server. For example, we can store all the user related tables in one database and all files/chunks related tables in another database. Although this approach is straightforward to implement it has some issues:

1. Will we still have scale issues? What if we have trillions of chunks to be stored and our database cannot support to store such huge number of records? How would we further partition such tables?
2. Joining two tables in two separate databases can cause performance and consistency issues. How frequently do we have to join user and file tables?

2. Range Based Partitioning: What if we store files/chunks in separate partitions based on the first letter of the File Path. So, we save all the files starting with letter ‘A’ in one partition and those that start with letter ‘B’ into another partition and so on. This approach is called range based partitioning. We can even combine certain less frequently occurring letters into one database

partition. We should come up with this partitioning scheme statically so that we can always store/find a file in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers. For example, if we decide to put all files starting with letter 'E' into a DB partition, and later we realize that we have too many files that start with letter 'E', to such an extent that we cannot fit them into one DB partition.

3. Hash-Based Partitioning: In this scheme we take a hash of the object we are storing and based on this hash we figure out the DB partition to which this object should go. In our case, we can take the hash of the 'FileID' of the File object we are storing to determine the partition the file will be stored. Our hashing function will randomly distribute objects into different partitions, e.g., our hashing function can always map any ID to a number between [1...256], and this number would be the partition we will store our object.

This approach can still lead to overloaded partitions, which can be solved by using Consistent Hashing.

10. Caching

We can have two kinds of caches in our system. To deal with hot files/chunks, we can introduce a cache for Block storage. We can use an off-the-shelf solution like Memcache, that can store whole chunks with their respective IDs/Hashes, and Block servers before hitting Block storage can quickly check if the cache has desired chunk. Based on clients' usage pattern we can determine how many cache servers we need. A high-end commercial server can have up to 144GB of memory; So, one such server can cache 36K chunks.

Which cache replacement policy would best fit our needs? When the cache is full, and we want to replace a chunk with a newer/hotter chunk, 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 chunk first.

Similarly, we can have a cache for Metadata DB.

11. Load Balancer (LB)

We can add Load balancing layer at two places in our system 1) Between Clients and Block servers and 2) Between Clients and Metadata 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 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, it won't take 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 backend server about their load and adjusts traffic based on that.

12. Security, Permissions and File Sharing

One of primary concern users will have while storing their files in the cloud would be the privacy and security of their data. Especially since in our system users can share their files with other users or even make them public to share it with everyone. To handle this, we will be storing permissions of each file in our metadata DB to reflect what files are visible or modifiable by any user.

Designing Facebook Messenger

Let's design an instant messaging service like Facebook Messenger, where users can send text messages to each other through web and mobile interfaces.

1. 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.

2. 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:

- Group Chats: Messenger should support multiple people talking to each other in a group.
- Push notifications: Messenger should be able to notify users of new messages when they are offline.

3. Capacity Estimation and Constraints

Let's assume that we have 500 million daily active users and on average each user sends 40 messages daily; this gives us 20 billion messages per day.

Storage Estimation: Let's assume that on average a message is 100 bytes, so to store all the messages for one day we would need 2TB of storage.

20 billion messages * 100 bytes => 2 TB/day

Although Facebook Messenger stores all previous chat history, but just for estimation to save five years of chat history, we would need 3.6 petabytes of storage.

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, this will give us 25MB of incoming data for each second.

2 TB / 86400 sec ~= 25 MB/s

Since each incoming message needs to go out to another user, we will need the same amount of bandwidth 25MB/s for both upload and download.

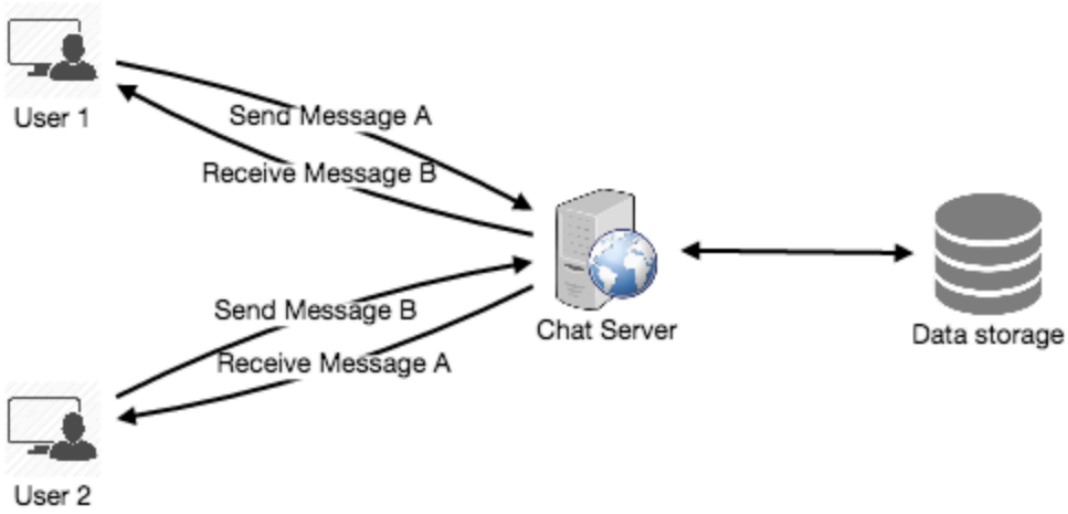
High level estimates:

Total messages	20 billion per day
Storage for each day	2TB
Storage for 5 years	3.6PB

Incomming data	25MB/s
Outgoing data	25MB/s

4. High Level Design

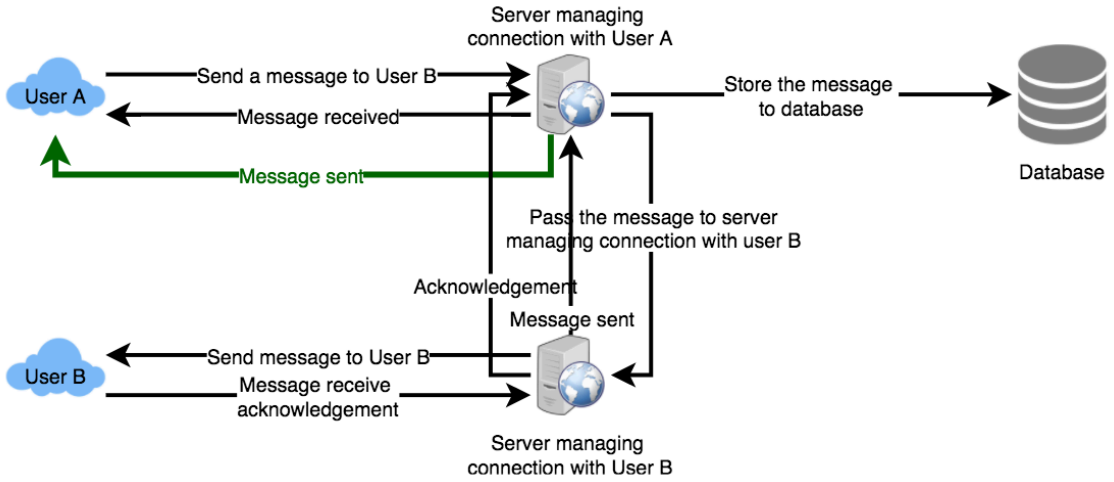
At a high level, we will need a chat server that would be the central piece orchestrating all the communications between users. When a user wants to send a message to another user, they will connect to the chat server and send the message to the server; the server then passes that message to the other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



The detailed workflow would look like this:

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.



Request flow for sending a message

5. 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.

Let's talk about these scenarios one by one:

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.
2. **Push model:** Users can keep a connection open with the server and can depend upon the server to notify them whenever there are new messages.

If we go with our first approach, then the server needs to keep track of messages that are still waiting to be delivered, and as soon as the receiving user connects to the server to ask for any new message, the server can return all the pending messages. To minimize latency for the user, they have to check the server quite frequently, and most of the time they will be getting an empty response if there are no pending message. This will waste a lot of resources and does not look like an efficient solution.

If we go with our second approach, where all the active users keep a connection open with the server, then 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, as the messages are delivery instantly on the opened connection.

How will clients maintain an open connection with the server? We can use HTTP Long Polling or WebSockets. In long polling, clients can request information from the server with the expectation that the 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? The server can maintain a hash table, where "key" would be the UserID and "value" would be the connection object. So whenever the server receives a message for a user, it looks up that user in the hash table to find 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. Assuming a modern server can handle 50K concurrent connections at any time, 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; that can map each UserID to a server to redirect the request.

How should the 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.

The chat server will first find the server that holds the connection for the receiver and pass the message to that server to send it to the receiver. The chat server can then send the acknowledgment to the sender; we don't need to wait for storing the message in the database; this can happen in the background. Storing the message is discussed in the next section.

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:

1. User-1 sends a message M1 to the server for User-2.
2. The server receives M1 at T1.
3. Meanwhile, User-2 sends a message M2 to the server for User-1.
4. The server receives the message M2 at T2, such that T2 > T1.
5. 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 the messages from database

Whenever the chat server receives a new message, it needs to store it in the database. To do so, we have two options:

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 that can 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. 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.

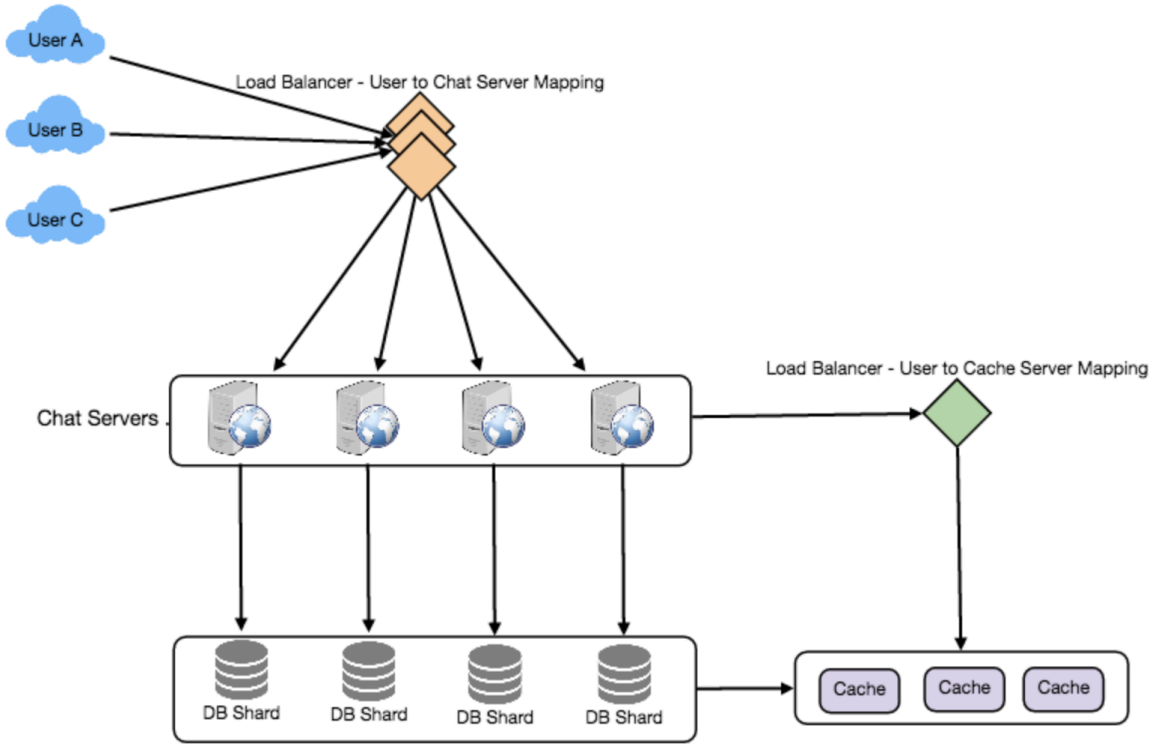
How should clients efficiently fetch data from the server? Clients should paginate while fetching data from the server. Page size could be different for different clients, e.g., cell phones have smaller screens, so we need lesser number of message/conversations in the viewport.

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, if we have 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 the server is broadcasting the 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.



Detailed component design for Facebook messenger

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.

6. 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. If one DB shard is 4TB, we will have "3.6PB/4TB ~= 900" shards for five years. For simplicity, let's assume we keep 1K shards. So 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. Our 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.

7. 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.

8. 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.

9. 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 cannot 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.

10. 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 our 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, which 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.

Designing Twitter

Let's design a Twitter like social networking service. Users of the service will be able to post tweets, follow other people and favorite tweets. Difficulty Level: Medium

1. What is Twitter?

Twitter is an online social networking service where users post and read short 140-character messages called "tweets". Registered users can post and read tweets, but those who are not registered can only read them. Users access Twitter through their website interface, SMS or mobile app.

2. Requirements and Goals of the System

We will be designing a simpler version of Twitter with following requirements:

Functional Requirements

1. Users should be able to post new tweets.
2. A user should be able to follow other users.
3. Users should be able to mark tweets favorite.
4. The service should be able to create and display user's timeline consisting of top tweets from all the people the user follows.
5. Tweets can contain photos and videos.

Non-functional Requirements

1. Our service needs to be highly available.
2. Acceptable latency of the system is 200ms for timeline generation.
3. Consistency can take a hit (in the interest of availability), if a user doesn't see a tweet for a while, it should be fine.

Extended Requirements

1. Searching tweets.
2. Reply to a tweet.
3. Trending topics – current hot topics/searches.
4. Tagging other users.
5. Tweet Notification.
6. Who to follow? Suggestions?
7. Moments.

3. Capacity Estimation and Constraints

Let's assume we have one billion total users, with 200 million daily active users (DAU). Also, we have 100 million new tweets every day, and on average each user follows 200 people.

How many favorites per day? If on average each user favorites five tweets per day, we will have:

200M users * 5 favorites => 1B favorites

How many total tweet-views our system will generate? Let's assume on average a user visits their timeline two times a day and visits five other people's pages. One each page if a user sees 20 tweets, total tweet-views our system will generate:

200M DAU * ((2 + 5) * 20 tweets) => 28B/day

Storage Estimates Let's say each tweet has 140 characters and we need two bytes to store a character without compression. Let's assume we need 30 bytes to store metadata with each tweet (like ID, timestamp, user ID, etc.). Total storage we would need:

100M * (280 + 30) bytes => 30GB/day

What would be our storage needs for five years? How much storage we would need for users' data, follows, favorites? We will leave this for exercise.

Not all tweets will have media, let's assume that on average every fifth tweet has a photo and every tenth has a video. Let's also assume on average a photo is 200KB and a video is 2MB. This will lead us to have 24TB of new media every day.

(100M/5 photos * 200KB) + (100M/10 videos * 2MB) ~= 24TB/day

Bandwidth Estimates Since total ingress is 24TB per day, this would translate into 290MB/sec.

Remember that we have 28B tweet views per day. We must show the photo of every tweet (if it has a photo), but let's assume that the users watch every 3rd video they see in their timeline. So, total egress will be:

(28B * 280 bytes) / 86400s of text => 93MB/s
+ (28B/5 * 200KB) / 86400s of photos => 13GB/S
+ (28B/10/3 * 2MB) / 86400s of Videos => 22GB/s

Total ~= 35GB/s

4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definition of the API for posting a new tweet:

tweet(api_dev_key, tweet_data, tweet_location, user_location, media_ids, maximum_results_to_return)

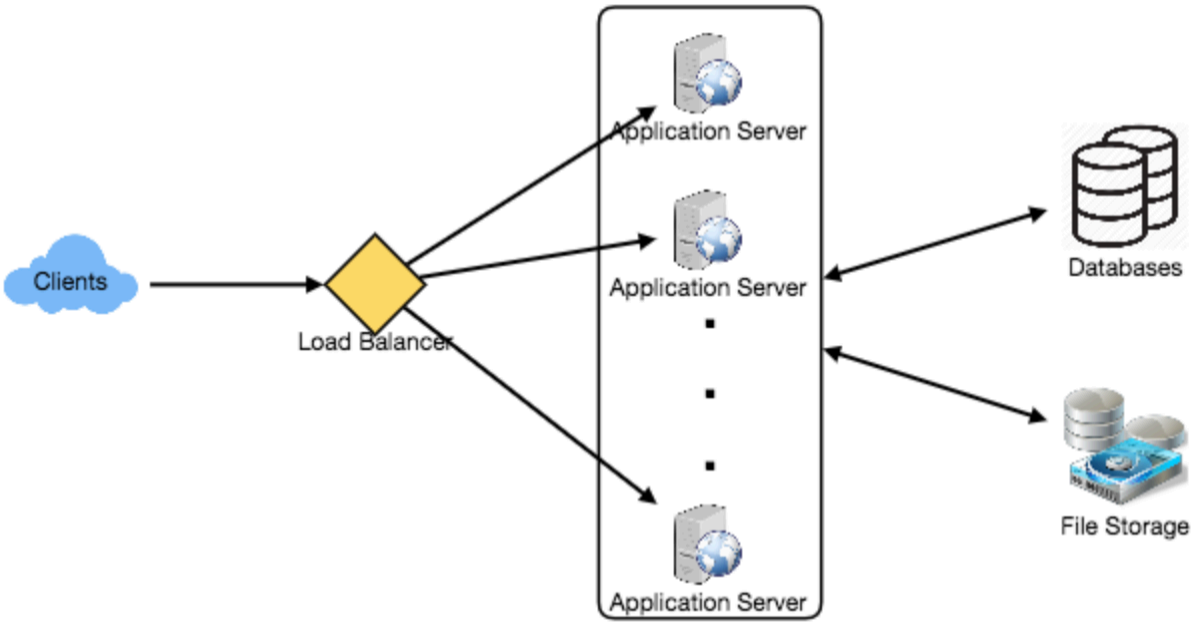
Parameters:
api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.
tweet_data (string): The text of the tweet, typically up to 140 characters.
tweet_location (string): Optional location (longitude, latitude) this Tweet refers to. user_location (string): Optional location (longitude, latitude) of the user adding the tweet.
media_ids (number[]): Optional list of media_ids to be associated with the Tweet. (All the media photo, video, etc.) need to be uploaded separately.

Returns: (string)
A successful post will return the URL to access that tweet. Otherwise, an appropriate HTTP error is returned.

5. High Level System Design

We need a system that can efficiently store all the new tweets, 100M/86400s => 1150 tweets per second and read 28B/86400s => 325K tweets per second. It is clear from the requirements that this will be a read-heavy system.

At a high level, we need multiple application servers to serve all these requests with load balancers in front of them for traffic distributions. On the backend, we need an efficient database that can store all the new tweets and can support a huge number of reads. We would also need some file storage to store photos and videos.



Although our expected daily write load is 100 million and read load is 28 billion tweets. This means, on average our system will receive around 1160 new tweets and 325K read requests per second. This traffic will be distributed unevenly throughout the day, though, at peak time we should expect at least a few thousand write requests and around 1M read requests per second. We should keep this thing in mind while designing the architecture of our system.

6. Database Schema

We need to store data about users, their tweets, their favorite tweets, and people they follow.

Tweet	
PK	**TweetID: int**
	UserID: int
	Content: varchar(140)
	TweetLatitude: int
	TweetLongitude: int
	UserLatitude: int
	UserLongitude: int
	CreationDate: datetime
	NumFavorites: int

User	
PK	**UserID: int**
	Name: varchar(20)
	Email: varchar(32)
	DateOfBirth: datetime
	CreationDate: datetime
	LastLogin: datatime

UserFollow	
PK	**UserID1: int** **UserID2: int**

Favorite	
PK	**FavoriteID: int**
	TweetID: int
	UserID: int
	CreationDate: datetime

For choosing between SQL and NoSQL databases to store the above schema, please see 'Database schema' under Designing Instagram.

7. Data Sharding

Since we have a huge number of new tweets every day and our read load is extremely high too, we need to distribute our data onto multiple machines such that we can read/write it efficiently. We have many options to shard our data; let's go through them one by one:

Sharding based on UserID: We can try storing all the data of a user on one server. While storing, we can pass the UserID to our hash function that will map the user to a database server where we will store all of the user's tweets, favorites, follows, etc. While querying for tweets/follows/favorites of a user, we can ask our hash function where can we find the data of a user and then read it from there. This approach has a couple of issues:

1. What if a user becomes hot? There could be a lot of queries on the server holding the user. This high load will affect the performance of our service.
2. Over time some users can end up storing a lot of tweets or have a lot of follows 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 use consistent hashing.

Sharding based on TweetID: Our hash function will map each TweetID to a random server where we will store that Tweet. To search tweets, we have to query all servers, and each server will return a set of tweets. A centralized server will aggregate these results to return them to the user. Let's look into timeline generation example, here are the number of steps our system has to perform to generate a user's timeline:

1. Our application (app) server will find all the people the user follows.
2. App server will send the query to all database servers to find tweets from these people.
3. Each database server will find the tweets for each user, sort them by recency and return the top tweets.
4. App server will merge all the results and sort them again to return the top results to the user.

This approach solves the problem of hot users, but in contrast to sharding by UserID, we have to query all database partitions to find tweets of a user, which can result in higher latencies.

We can further improve our performance by introducing cache to store hot tweets in front of the database servers.

Sharding based on Tweet creation time: Storing tweets based on recency will give us the advantage of fetching all the top tweets quickly, and we only have to query a very small set of servers. But the problem here is that the traffic load will not be distributed, e.g., while writing, all new tweets will be going to one server, and the remaining servers will be sitting idle. Similarly while reading, the server holding latest data will have a very high load as compared to servers holding old data.

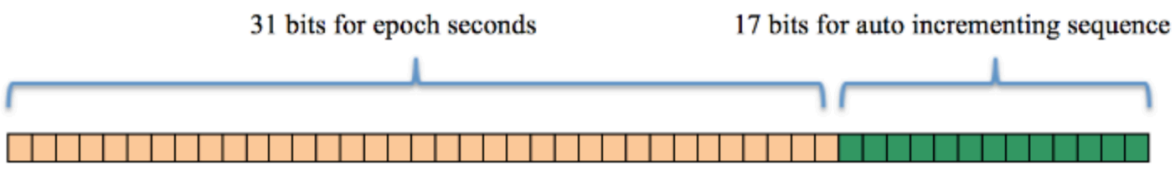
What if we can combine sharding by TweedID and Tweet creation time? If we don't store tweet creation time separately and use TweetID to reflect that, we can get benefits of both the approaches. This way it will be quite quick to find latest Tweets. For this, we must make each TweetID universally unique in our system, and each TweetID should contain timestamp too.

We can use epoch time for this. Let's say our TweetID will have two parts; the first part will be representing epoch seconds and the second part will be an auto-incrementing sequence. So, to make a new TweetID, we can take the current epoch time and append an auto-incrementing number to it. We can figure out shard number from this TweetID and store it there.

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

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

We would need 31 bits to store this number. Since on average we are expecting 1150 new tweets per second, we can allocate 17 bits to store auto incremented sequence; this will make our TweetID 48 bits long. So, every second we can store (2^17 => 130K) new tweets. We can reset our auto incrementing sequence every second. For fault tolerance and better performance, we can have two database servers to generate auto-incrementing keys for us, one generating even numbered keys and the other generating odd numbered keys.



If we assume our current epoch seconds are “1483228800”, our TweetID will look like this:

1483228800 000001
1483228800 000002
1483228800 000003
1483228800 000004
...

If we make our TweetID 64bits (8 bytes) long, we can easily store tweets for next 100 years and also store them for mili-seconds granularity.

8. Cache

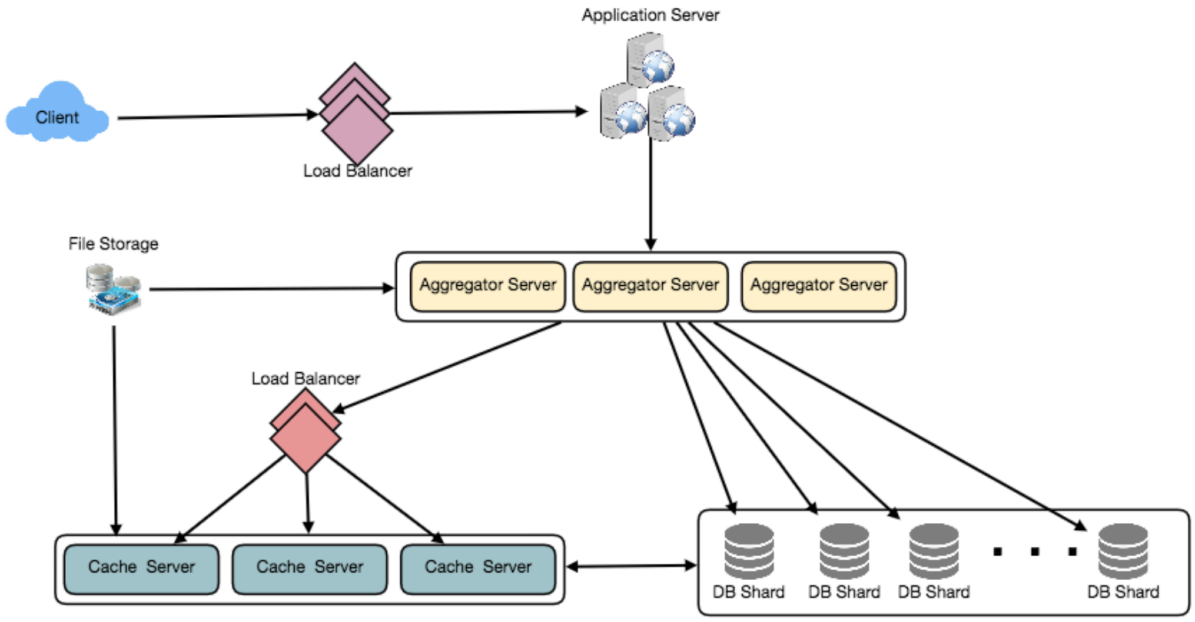
We can introduce a cache for database servers to cache hot tweets and users. We can use an off-the-shelf solution like Memcache that can store the whole tweet objects. Application servers before hitting database can quickly check if the cache has desired tweets. Based on clients’ usage pattern we can determine how many cache servers we need.

Which cache replacement policy would best fit our needs? When the cache is full, and we want to replace a tweet with a newer/hotter tweet, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system. Under this policy, we discard the least recently viewed tweet first.

How can we have more intelligent cache? If we go with 80-20 rule, that is 20% of tweets are generating 80% of read traffic which means that certain tweets are so popular that majority of people read them. This dictates that we can try to cache 20% of daily read volume from each shard.

What if we cache the latest data? Our service can benefit from this approach. Let’s say if 80% of our users see tweets from past three days only; we can try to cache all the tweets from past three days. Let’s say we have dedicated cache servers that cache all the tweets from all users from past three days. As estimated above, we are getting 100 million new tweets or 30GB of new data every day (without photos and videos). If we want to store all the tweets from last three days, we would need less than 100GB of memory. This data can easily fit into one server, but we should replicate it onto multiple servers to distribute all the read traffic to reduce the load on cache servers. So whenever we are generating a user’s timeline, we can ask the cache servers if they have all the recent tweets for that user, if yes, we can simply return all the data from the cache. If we don’t have enough tweets in the cache, we have to query backend to fetch that data. On a similar design, we can try caching photos and videos from last three days.

Our cache would be like a hash table, where 'key' would be 'OwnerID' and 'value' would be a doubly linked list containing all the tweets from that user in past three days. Since we want to retrieve most recent data first, we can always insert new tweets at the head of the linked list, which means all the older tweets will be near the tail of the linked list. Therefore, we can remove tweets from the tail to make space for newer tweets.



9. Timeline Generation

For a detailed discussion about timeline generation, take a look at Designing Facebook's Newsfeed.

10. Replication and Fault Tolerance

Since our system is read-heavy, we can have multiple secondary database servers for each DB partition. Secondary servers will be used for read traffic only. All writes will first go to the primary server and then will be replicated to secondary servers. This scheme will also give us fault tolerance, as whenever the primary server goes down, we can failover to a secondary server.

11. Load Balancing

We can add Load balancing layer at three places in our system 1) Between Clients and Application servers 2) Between Application servers and database replication servers and 3) Between Aggregation servers and Cache server. Initially, a simple Round Robin approach can be adopted; that distributes incoming requests equally among servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is 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, it won't take 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 backend server about their load and adjusts traffic based on that.

12. Monitoring

Having the ability to monitor our systems is crucial. We should constantly collect data to get an instant insight into how our system is doing. We can collect following metrics/counters to get an understanding of the performance of our service:

1. New tweets per day/second, what is the daily peak?
2. Timeline delivery stats, how many tweets per day/second our service is delivering.
3. Average latency that is seen by the user to refresh timeline.

By monitoring these counters, we will realize if we need more replication or load balancing or caching, etc.

13. Extended Requirements

How to serve feeds? Get all the latest tweets from the people someone follows and merge/sort them by time. Use pagination to fetch/show tweets. Only fetch top N tweets from all the people someone follows. This N will depend on the client's Viewport, as on mobile we show fewer tweets compared to a Web client. We can also cache next top tweets to speed things up.

Alternately, we can pre-generate the feed to improve efficiency, for details please see 'Ranking and timeline generation' under Designing Instagram.

Retweet: With each Tweet object in the database, we can store the ID of original Tweet and not store any contents on this retweet object.

Trending Topics: We can cache most frequently occurring hashtags or searched queries in the last N seconds and keep updating them after every M seconds. We can rank trending topics based on the frequency of tweets or search queries or retweets or likes. We can give more weight to topics which are shown to more people.

Who to follow? How to give suggestions? This feature will improve user engagement. We can suggest friends of people someone follows. We can go two or three level down to find famous people for the suggestions. We can give preference to people with more followers.

As only a few suggestions can be made at any time, use Machine Learning (ML) to shuffle and re-prioritize. ML signals could include people with recently increased follow-ship, common followers if the other person is following this user, common location or interests, etc.

Moments: Get top news for different websites for past 1 or 2 hours, figure out related tweets, prioritize them, categorize them (news, support, financials, entertainment, etc.) using ML – supervised learning or Clustering. Then we can show these articles as trending topics in Moments.

Search: Search involves Indexing, Ranking, and Retrieval of tweets. A similar solution is discussed in our next problem Design Twitter Search.

Designing Youtube or Netflix

Let's design a video sharing service like Youtube, where users will be able to upload/view/search videos. Similar Services: netflix.com, vimeo.com, dailymotion.com, veoh.com Difficulty Level: Medium

1. Why Youtube?

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.

2. Requirements and Goals of the System

For the sake of this exercise, we plan to design a simpler version of Youtube with following requirements:

Functional Requirements:

1. Users should be able to upload videos.
2. Users should be able to share and view videos.
3. Users can perform searches based on video titles.
4. Our services should be able to record stats of videos, e.g., likes/dislikes, total number of views, etc.
5. Users should be able to add and view comments on videos.

Non-Functional Requirements:

1. The system should be highly reliable, any video uploaded should not be lost.
2. 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.
3. 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.

3. Capacity Estimation and Constraints

Let's assume we have 1.5 billion total users, 800 million of whom are daily active users. If, on the average, a user views five videos per day, total video-views per second would be:

800M * 5 / 86400 sec => 46K videos/sec

Let's assume our upload:view ratio is 1:200 i.e., for every video upload we have 200 video viewed, giving us 230 videos uploaded per second.

46K / 200 => 230 videos/sec

Storage Estimates: Let's assume that every minute 500 hours worth of videos are uploaded to Youtube. If on average, 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 hours * 60 min * 50MB => 1500 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.

500 hours * 60 mins * 10MB => 300GB/min (5GB/sec)

Assuming an upload:view ratio of 1:200, we would need 1TB/s outgoing bandwidth.

4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definitions of the APIs for uploading and searching videos:

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.
vide_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), and 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.

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.

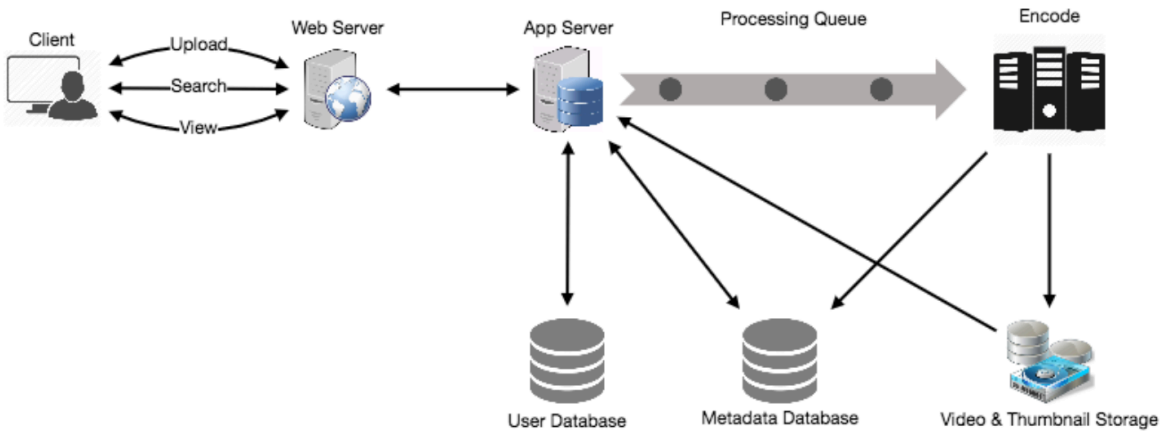
Returns: (JSON)
A JSON containing information about the list of video resources matching the search query. Each video resource will have a video title, a thumbnail, a video creation date and how many views it has.

5. 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.



High level design of Youtube

6. Database Schema

Video metadata storage - MySql
Videos 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

- UserID, Name, email, address, age, registration details etc.

7. Detailed Component Design

The service would be read-heavy, so we will focus on building a system that can retrieve videos quickly. We can expect our read:write ratio as 200:1, which means for every video upload there are 200 video views.

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, our 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 five thumbnails, we need to have a very efficient storage system that can serve a huge read traffic. There will be two consideration before deciding which storage system will be used for thumbnails:

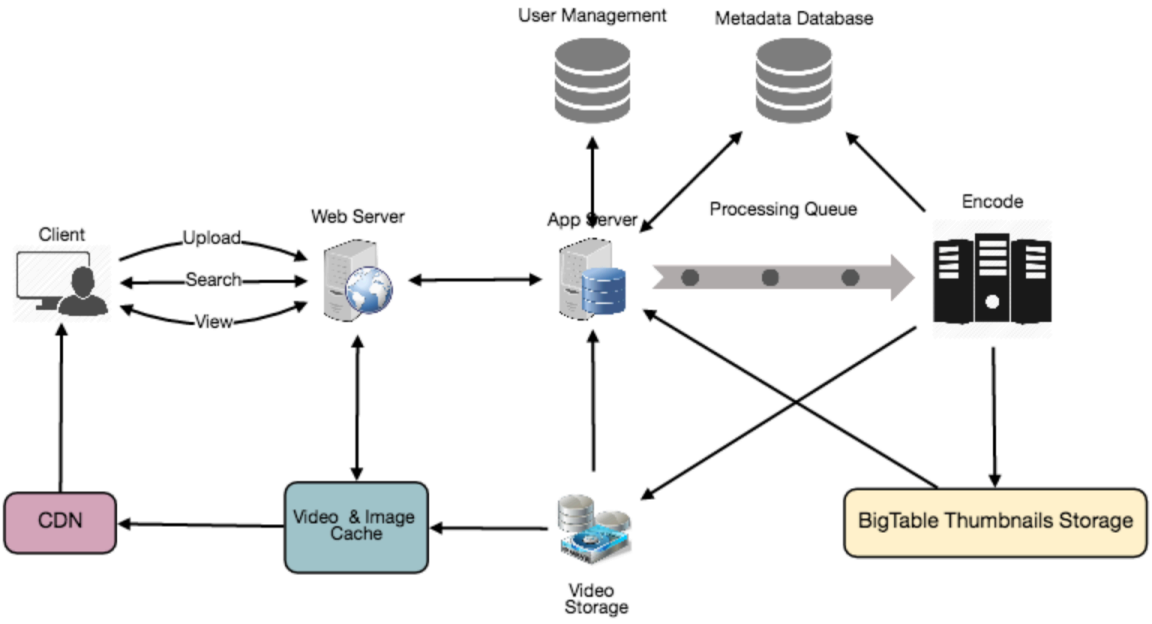
1. Thumbnails are small files, say maximum 5KB each.
2. Read traffic for thumbnails will be huge compared to videos. Users will be watching one video at a time, but they might be looking at a page that has 20 thumbnails of other videos.

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 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 the video into multiple formats. Once all the encoding is completed; uploader is notified, and video is made available for view/sharing.



Detailed component design of Youtube

8. 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: 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.

This approach has a couple of issues:

1. 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.
2. 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: 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. 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.

9. Video Deduplication

With a huge number 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 the end user, these inefficiencies will be realized in the form of 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.

10. 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. First, since our service tries to load balance locally, it leads to multiple redirections if the host that receives the redirection can't serve the video. Also, each redirection requires a client to make an additional HTTP request; it also leads to 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.

11. 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. Under this policy, we discard the least recently viewed row first.

How can we build more intelligent cache? If we go with 80-20 rule, i.e., 20% of daily read volume for 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.

12. 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. Take a look at 'CDN' section in our Cachingchapter.

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.

13. 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.

Designing Typeahead Suggestion

Let's design a real-time suggestion service, which will recommend terms to users as they enter text for searching. Similar Services: Auto-suggestions, Typeahead search

1. What is Typeahead Suggestion?

Typeahead suggestions enable users to search for known and frequently searched terms. 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 suggestion to complete the query. Typeahead suggestions help the user to articulate their search queries better. It's not about speeding up the search process but rather about guiding the users and lending them a helping hand in constructing their search query.

2. Requirements and Goals of the System

Functional Requirements: As the user types in their query, our service should suggest top 10 terms starting with whatever user has typed.

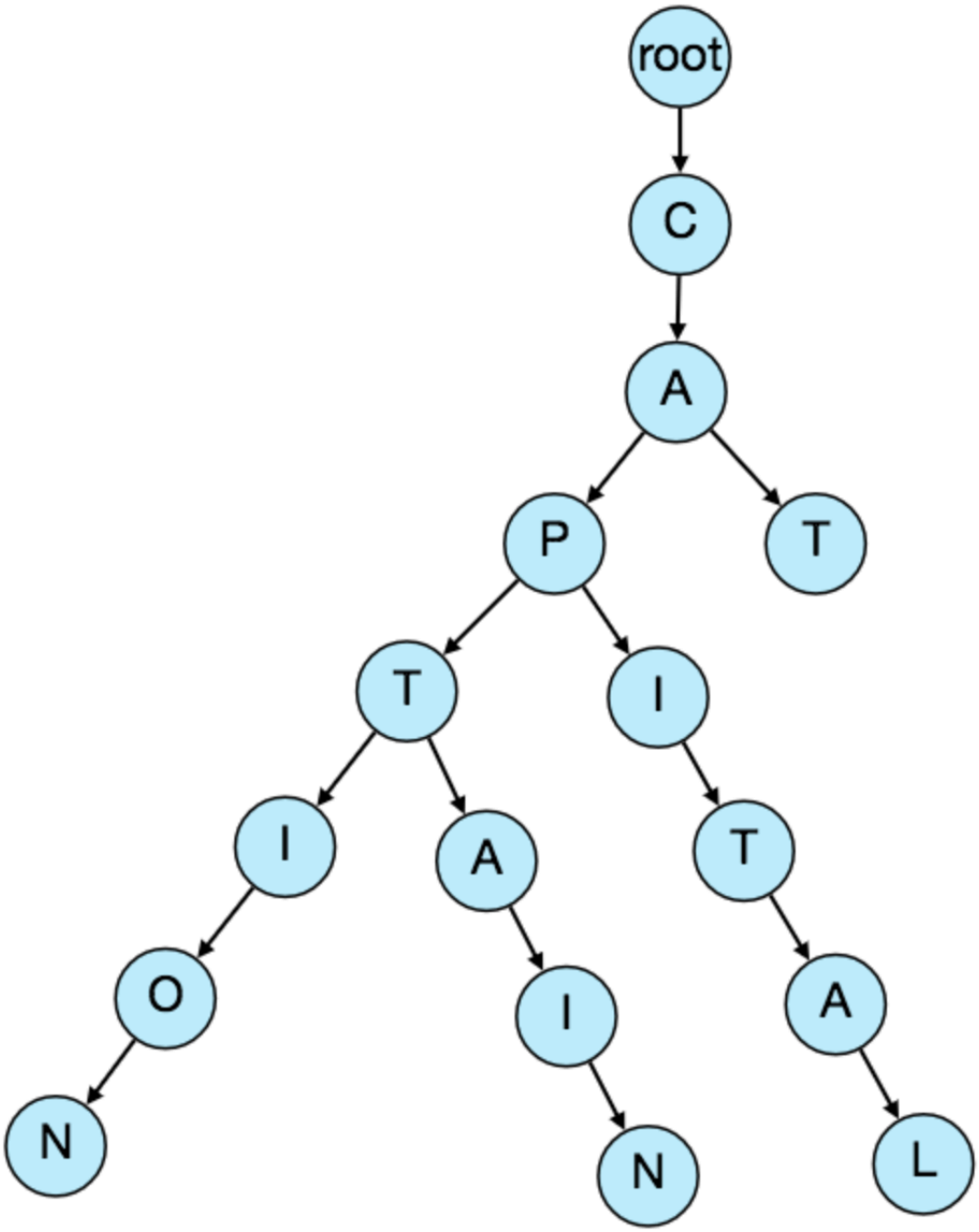
Non-function Requirements: The suggestions should appear in real-time. The user should be able to see the suggestions within 200ms.

3. Basic System Design and Algorithm

The problem we are solving is that we have a lot of 'strings' that we need to store in such a way that users can search on any prefix. Our service will suggest next terms that will match the given prefix. For example, if our database contains following terms: cap, cat, captain, capital; and the user has typed in 'cap', our system should suggest 'cap', 'captain' and 'capital'.

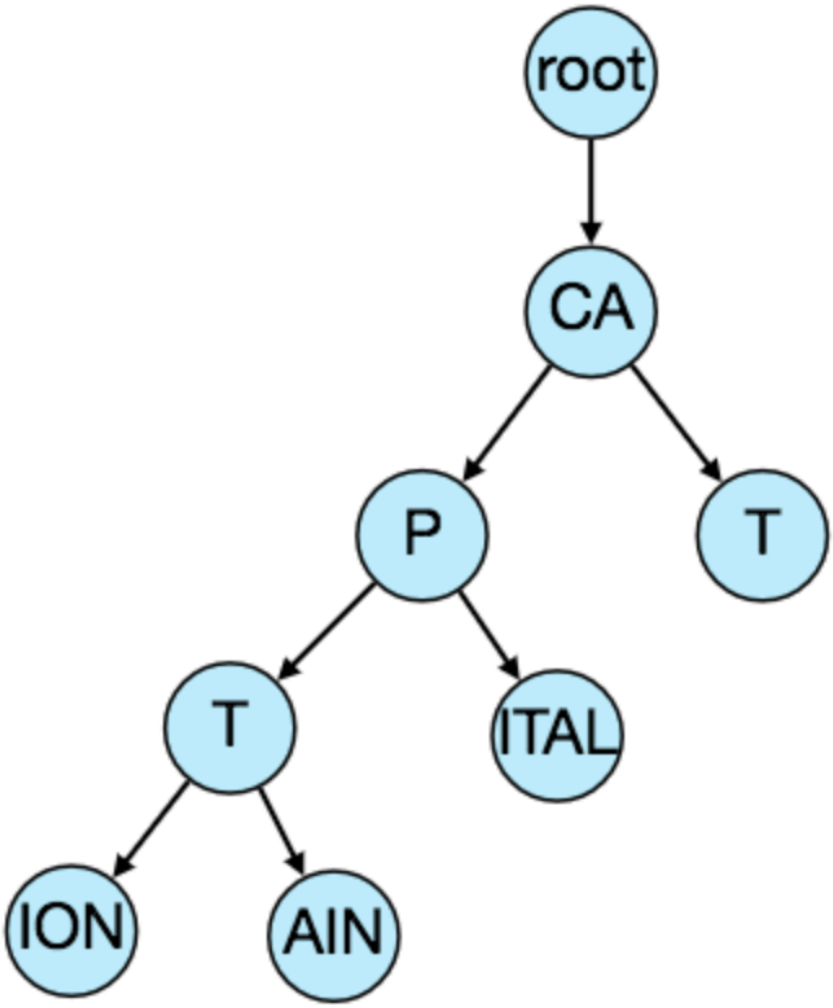
Since we've to serve a lot of queries with minimum latency, we need to come up with a scheme that can efficiently store our data such that it can be queried quickly. We can't depend upon some database for this; we need to store our index in memory in a highly efficient data structure.

One of the most appropriate data structure that can serve our purpose would be the Trie (pronounced "try"). A trie is a tree-like data structure used to store phrases where each node stores a character of the phrase in a sequential manner. For example, if we need to store 'cap, cat, caption, captain, capital' in the trie, it would look like:



Now if the user has typed ‘cap’, our service can traverse the trie to go to the node ‘P’ to find all the terms that start with this prefix (e.g., cap-tion, cap-ital etc).

We can merge nodes that have only one branch to save storage space. The above trie can be stored like this:



Should we've case insensitive trie? For simplicity and search use case let's assume our data is case insensitive.

How to find top suggestion? Now that we can find all the terms given a prefix, how can we know what're the top 10 terms that we should suggest? One simple solution could be to store the count of searches that terminated at each node, e.g., if users have searched about 'CAPTAIN' 100 times and 'CAPTION' 500 times, we can store this number with the last character of the phrase. So now if the user has typed 'CAP' we know the top most searched word under the prefix 'CAP' is 'CAPTION'. So given a prefix, we can traverse the sub-tree under it, to find the top suggestions.

Given a prefix, how much time it can take to traverse its sub-tree? Given the amount of data we need to index, we should expect a huge tree. Even, traversing a

sub-tree would take really long, e.g., the phrase 'system design interview questions' is 30 levels deep. Since we've very tight latency requirements, we do need to improve the efficiency of our solution.

Can we store top suggestions with each node? This can surely speed up our searches but will require a lot of extra storage. We can store top 10 suggestions at each node that we can return to the user. We've to bear the big increase in our storage capacity to achieve the required efficiency.

We can optimize our storage by storing only references of the terminal nodes rather than storing the entire phrase. To find the suggested term we've to traverse back using the parent reference from the terminal node. We will also need to store the frequency with each reference to keep track of top suggestions.

How would we build this trie? We can efficiently build our trie bottom up. Each parent node will recursively call all the child nodes to calculate their top suggestions and their counts. Parent nodes will combine top suggestions from all of their children to determine their top suggestions.

How to update the trie? Assumeing five billion searches every day, which would give us approximately 60K queries per second. If we try to update our trie for every query it'll be extremely resource intensive and this can hamper our read requests too. One solution to handle this could be to update our trie offline after a certain interval.

As the new queries come in, we can log them and also track their frequencies. Either we can log every query or do sampling and log every 1000th query. For example, if we don't want to show a term which is searched for less than 1000 times, it's safe to log every 1000th searched term.

We can have a Map-Reduce (MR) setup to process all the logging data periodically, say every hour. These MR jobs will calculate frequencies of all searched terms in the past hour. We can then update our trie with this new data. We can take the current snapshot of the trie and update it with all the new terms and their frequencies. We should do this offline, as we don't want our read queries to be blocked by update trie requests. We can have two options:

1. We can make a copy of the trie on each server to update it offline. Once done we can switch to start using it and discard the old one.
2. Another option is we can have a master-slave configuration for each trie server. We can update slave while the master is serving traffic. Once the update is complete, we can make the slave our new master. We can later update our old master, which can then start serving traffic too.

How can we update the frequencies of typeahead suggestions? Since we are storing frequencies of our typeahead suggestions with each node, we need to update them too. We can update only difference in frequencies rather than recounting all search terms from scratch. If we're keeping count of all the terms searched in last 10 days, we'll need to subtract the counts from the time period no longer included and add the counts for the new time period being included. We can add and subtract frequencies based on Exponential Moving Average (EMA) of each term. In EMA, we give more weight to the latest data. It's also known as the exponentially weighted moving average.

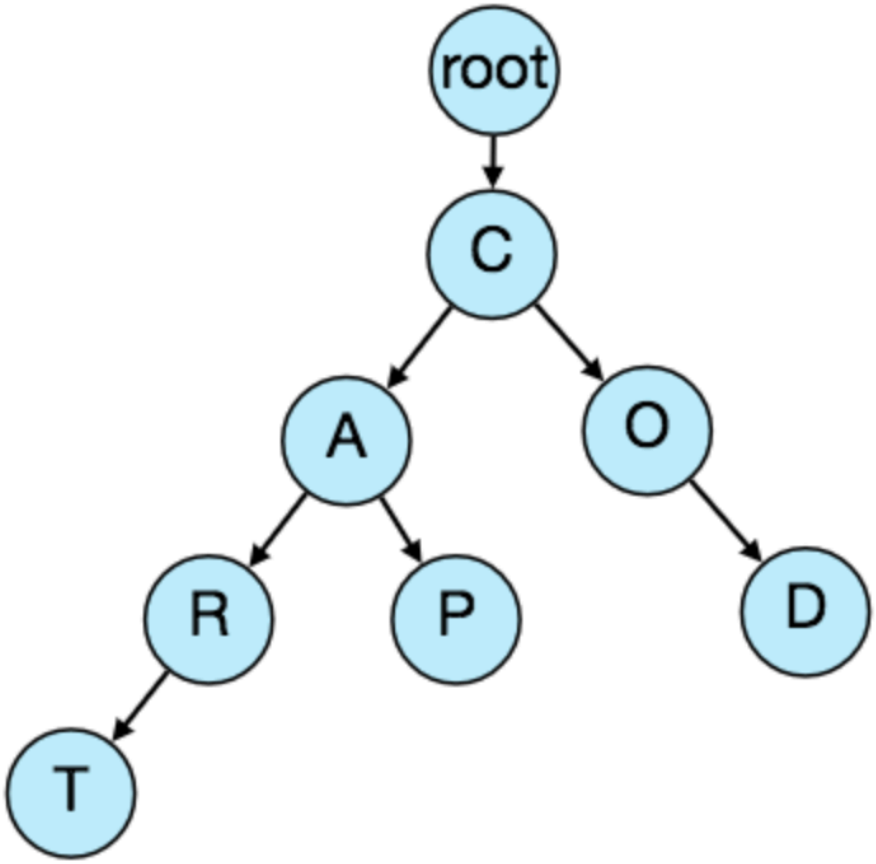
After inserting a new term in the trie, we'll go to the terminal node of the phrase and increase its frequency. Since we're storing the top 10 queries in each node, it is possible that this particular search term jumped into the top 10 queries of a few other nodes. So, we need to update the top 10 queries of those nodes then. We've to traverse back from the node to all the way up to the root. For every parent, we check if the current 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 a part of the top 10. If so, we insert this new term and remove the term with the lowest frequency.

How can we remove a term from the trie? Let's say we've to remove a term from the trie, because of some legal issue or hate or piracy etc. We can completely remove such terms from the trie when the regular update happens, meanwhile, we can 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 terms ranking, we have to consider other factors too, e.g., freshness, user location, language, demographics, personal history etc.

4. Permanent Storage of the Trie

How to store trie in a file so that we can rebuild our trie easily - this will be needed when a machine restarts? We can take snapshot of our trie periodically and store it in file. This will enable us to rebuild a trie if the server goes down. To store, we can start with the root node and save the trie level-by-level. With each node we can store what character it contains and how many children it has. Right after each node we should put all of its children. Let's assume we have following trie:



If we store this trie in a file with the above-mentioned scheme, we will have: “C2,A2,R1,T,P,O1,D”. From this, we can easily rebuild our trie.

If you’ve noticed we are not storing top suggestions and their counts with each node, it is hard to store this information, as our trie is being stored top down, we don’t have child nodes created before the parent, so there is no easy way to store their references. For this, we have to recalculate all the top terms with counts. This can be done while we are building the trie. Each node will calculate its top suggestions and pass it to its parent. Each parent node will merge results from all of its children to figure out its top suggestions.

5. Scale Estimation

If we are building a service, which has the same scale as that of Google, we can expect 5 billion searches every day, which would give us approximately 60K queries per second.

Since there will be a lot of duplicates in 5 billion queries, we can assume that only 20% of these will be unique. If we only want to index top 50% of the search terms, we can get rid of a lot of less frequently searched queries. Let's assume we will have 100 million unique terms for which we want to build an index.

Storage Estimation: If on the average each query consists of 3 words, and if the average length of a word is 5 characters, this will give us 15 characters of average query size. Assuming we need 2 bytes to store a character, we will need 30 bytes to store an average query. So total storage we will need:

100 million * 30 bytes => 3 GB

We can expect some growth in this data every day, but we should also be removing some terms that are not searched anymore. If we assume we have 2% new queries every day and if we are maintaining our index for last one year, total storage we should expect:

3GB + (0.02 * 3 GB * 365 days) => 25 GB

6. Data Partition

Although our index can easily fit on one server, we can still partition it in order to meet our requirements of higher efficiency and lower latencies. How can we efficiently partition our data to distribute it onto multiple servers?

a. Range Based Partitioning: What if we store our phrases in separate partitions based on their first letter. So we save all the terms starting with letter 'A' in one partition and those that start with letter 'B' into another partition and so on. We can even combine certain less frequently occurring letters into one database partition. We should come up with this partitioning scheme statically so that we can always store and search terms in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers, for instance; if we decide to put all terms starting with letter 'E' into a DB partition, but later we realize that we have too many terms that start with letter 'E', which we can't fit into one DB partition.

We can see that the above problem will happen with every statically defined scheme. It is not possible to calculate if each of our partitions will fit on one server statically.

b. Partition based on the maximum capacity of the server: Let's say we partition our trie based on the maximum memory capacity of the servers. We can keep storing data on a server as long as it has memory available. Whenever a sub-

tree cannot fit into a server, we break our partition there to assign that range to this server and move on the next server to repeat this process. Let's say, if our first trie server can store all terms from 'A' to 'AABC', which mean our next server will store from 'AABD' onwards. If our second server could store up to 'BXA', next serve will start from 'BXB' and so on. We can keep a hash table to quickly access this partitioning scheme:

Server 1, A-AABC
Server 2, AABD-BXA
Server 3, BXB-CDA

For querying, if the user has typed 'A' we have to query both server 1 and 2 to find the top suggestions. When the user has typed 'AA', still we have to query server 1 and 2, but when the user has typed 'AAA' we only need to query server 1.

We can have a load balancer in front of our trie servers, which can store this mapping and redirect traffic. Also if we are querying from multiple servers, either we need to merge the results at the server side to calculate overall top results, or make our clients do that. If we prefer to do this on the server side, we need to introduce another layer of servers between load balancers and trie servers, let's call them aggregator. These servers will aggregate results from multiple trie servers and return the top results to the client.

Partitioning based on the maximum capacity can still lead us to hotspots e.g., if there are a lot of queries for terms starting with 'cap', the server holding it will have a high load compared to others.

c. Partition based on the hash of the term: Each term will be passed to a hash function, which will generate a server number and we will store the term on that server. This will make our term distribution random and hence minimizing hotspots. To find typeahead suggestions for a term, we have to ask all servers and then aggregate the results. We have to use consistent hashing for fault tolerance and load distribution.

7. Cache

We should realize that caching the top searched terms will be extremely helpful in our service. There will be a small percentage of queries 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.

We can also build a simple Machine Learning (ML) model that can try to predict the engagement on each suggestion based on simple counting, personalization, or trending data etc., and cache these terms.

8. Replication and Load Balancer

We should have replicas for our trie servers both for load balancing and also for fault tolerance. We also need a load balancer that keeps track of our data partitioning scheme and redirects traffic based on the prefixes.

9. Fault Tolerance

What will happen when a trie server goes down? As discussed above we can have a master-slave configuration, if the master dies slave can take over after failover. Any server that comes back up, can rebuild the trie based on the last snapshot.

10. Typeahead Client

We can perform the following optimizations on the client to improve user's experience:

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 requests.
3. Initially, the client can wait until the user enters a couple of characters.
4. Clients can pre-fetch some data from the server to save future requests.
5. Clients can store the recent history of suggestions locally. Recent history has a very high rate of being reused.
6. Establishing an early connection with server turns out to be one of the most important factors. As soon as the user opens the search engine website, the client can open a connection with the server. So when user types in the first character, client doesn't waste time in establishing the connection.
7. The server can push some part of their cache to CDNs and Internet Service Providers (ISPs) for efficiency.

11. 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 cache them on the client too. The server can add these personalized terms in the final set, before sending it to the user. Personalized searches should always come before others.

Designing an API Rate Limiter

Let's design an API Rate Limiter which will throttle users based upon the number of the requests they are sending.

1. What is a Rate Limiter?

Imagine we've a service which is receiving a huge number of requests, but it can only serve a limited number of requests per second. To handle this problem, we would need some kind of throttling or rate limiting mechanism that will allow only a certain number of requests which our service can respond to. A rate limiter, at a high-level, limits the number of events an entity (user, device, IP, etc.) can perform in a particular time window. For example:

- A user can send only one message per second.
- A user is allowed only three failed credit card transactions per day.
- A single IP can only create twenty accounts per day.

In general, a rate limiter caps how many requests a sender can issue in a specific time window. It then blocks requests once the cap is reached.

2. Why do we need API rate limiting?

Rate Limiting helps to protect services against abusive behaviors targeting the application layer like Denial-of-service (DOS) attacks, brute-force password attempts, brute-force credit card transactions, etc. These attacks are usually a barrage of HTTP/S requests which may look like they are coming from real users, but are typically generated by machines (or bots). As a result, these attacks are often harder to detect and can more easily bring down a service, application, or an API.

Rate limiting is also used to prevent revenue loss, to reduce infrastructure costs, to stop spam and online harassment. Following is a list of scenarios that can benefit from Rate limiting by making a service (or API) more reliable:

- **Misbehaving clients/scripts:** Either intentionally or unintentionally, some entities can overwhelm a service by sending a large number of requests. Another scenario could be when a user is sending a lot of lower-priority requests, and we want to make sure that it doesn't affect the high-priority traffic. For example, users sending a high volume of requests for analytics data should not be allowed to hamper critical transactions for other users.

- **Security:** By limiting the number of the second-factor attempts (in 2-factor auth) that the users are allowed to perform, for example, the number of times they're allowed to try with a wrong password.
- **To prevent abusive behavior and bad design practices:** Without API limits, developers of client applications would use sloppy development tactics, for example requesting the same information over and over again.
- **To keep costs and resource usage under control:** Services are generally designed for normal input behavior, for example, a user writing a single post in a minute. Computers could easily push thousands/second through an API. Rate limiter enables controls on service APIs.
- **Revenue:** Certain services might want to limit operations based on the tier of their customer's service, and thus create a revenue model based on rate limiting. There could be default limits for all the APIs a service offers. To go beyond that, the user has to buy higher limits
- **To eliminate spikiness in traffic:** So that a service stays up for everyone else.

3. Requirements and Goals of the System

Our Rate Limiter should meet the following requirements:

Functional Requirements:

1. Limit the number of requests an entity can send to an API within a time window, e.g., 15 requests per second.
2. The APIs are accessible through a cluster, so the rate limit should be considered across different servers. The user should get an error message whenever the defined threshold is crossed within a single server or across a combination of servers.

Non-Functional Requirements:

1. The system should be highly available. The rate limiter should always work since it protects our service from external attacks.
2. Our rate limiter should not introduce substantial latencies affecting the user experience.

4. How to do Rate Limiting?

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 “429” as HTTP status to the user with message content as “Too many requests”.

5. What are different types of throttling?

Here are the three famous throttling types that are used by different services:

Hard Throttling: The number of API requests cannot exceed the throttle limit.

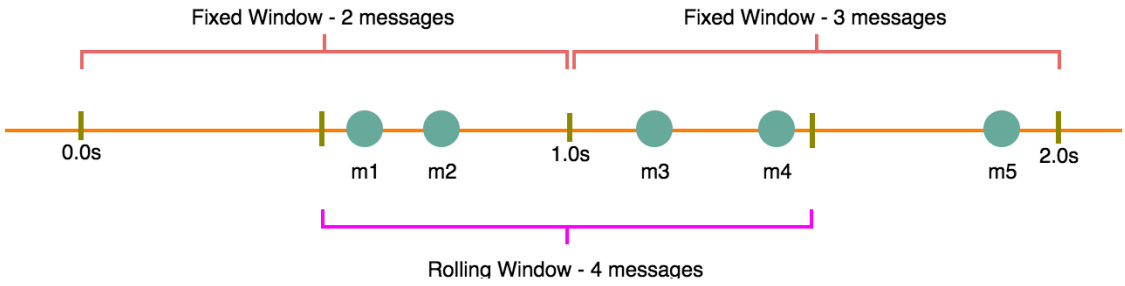
Soft Throttling: In this type, we can set the API request limit to exceed a certain percentage. For example, if we have rate-limit of 100 messages a minute and 10% exceed limit. Our rate limiter will allow up to 110 messages per minute.

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 if there are free resources available in the system.

6. What are different types of algorithms used for Rate Limiting?

Following are the two types of algorithms used for Rate Limiting:

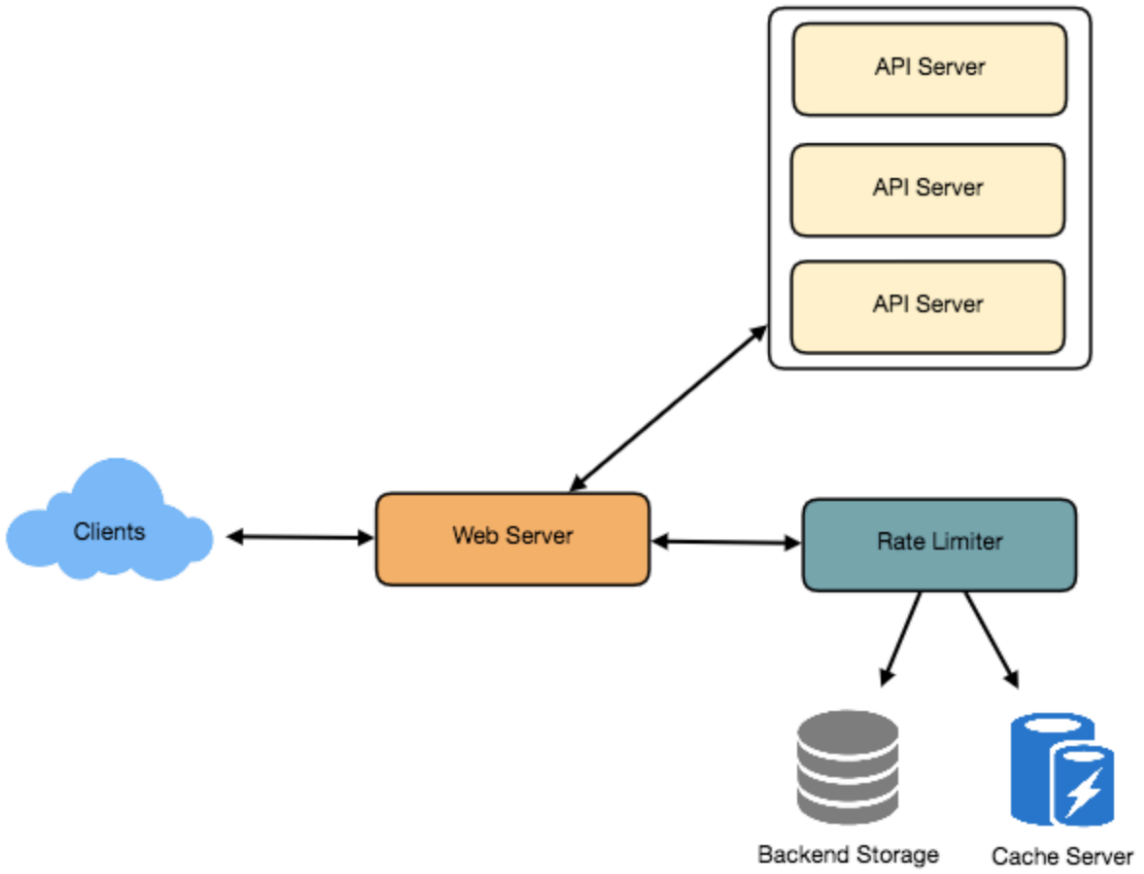
Fixed Window Algorithm: In this algorithm, the time window is considered from the start of the time-unit to the end of the time-unit. For example, a period would be considered as 0-60 seconds for a minute irrespective of the time frame at which the API request has been made. In the diagram below, there are two messages between 0-1 second and three messages between 1-2 second. If we’ve a rate limiting of two messages a second, this algorithm will throttle only ‘m5’.



Rolling Window Algorithm: In this algorithm, the time window is considered from the fraction of the time at which the request is made plus the time window length. For example, if there are two messages sent at 300th millisecond and 400th millisecond of a second, we'll count them as two messages from 300th millisecond of that second up to the 300th millisecond of next second. In the above diagram, keeping two messages a second, we'll throttle 'm3' and 'm4'.

7. High level design for Rate Limiter

Rate Limiter will be responsible for deciding which request will be served by the API servers and which request will be declined. Once a new request arrives, Web Server first asks the Rate Limiter to decide if it will be served or throttled. If the request is not throttled, then it'll be passed to the API servers.



High level design for Rate Limiter

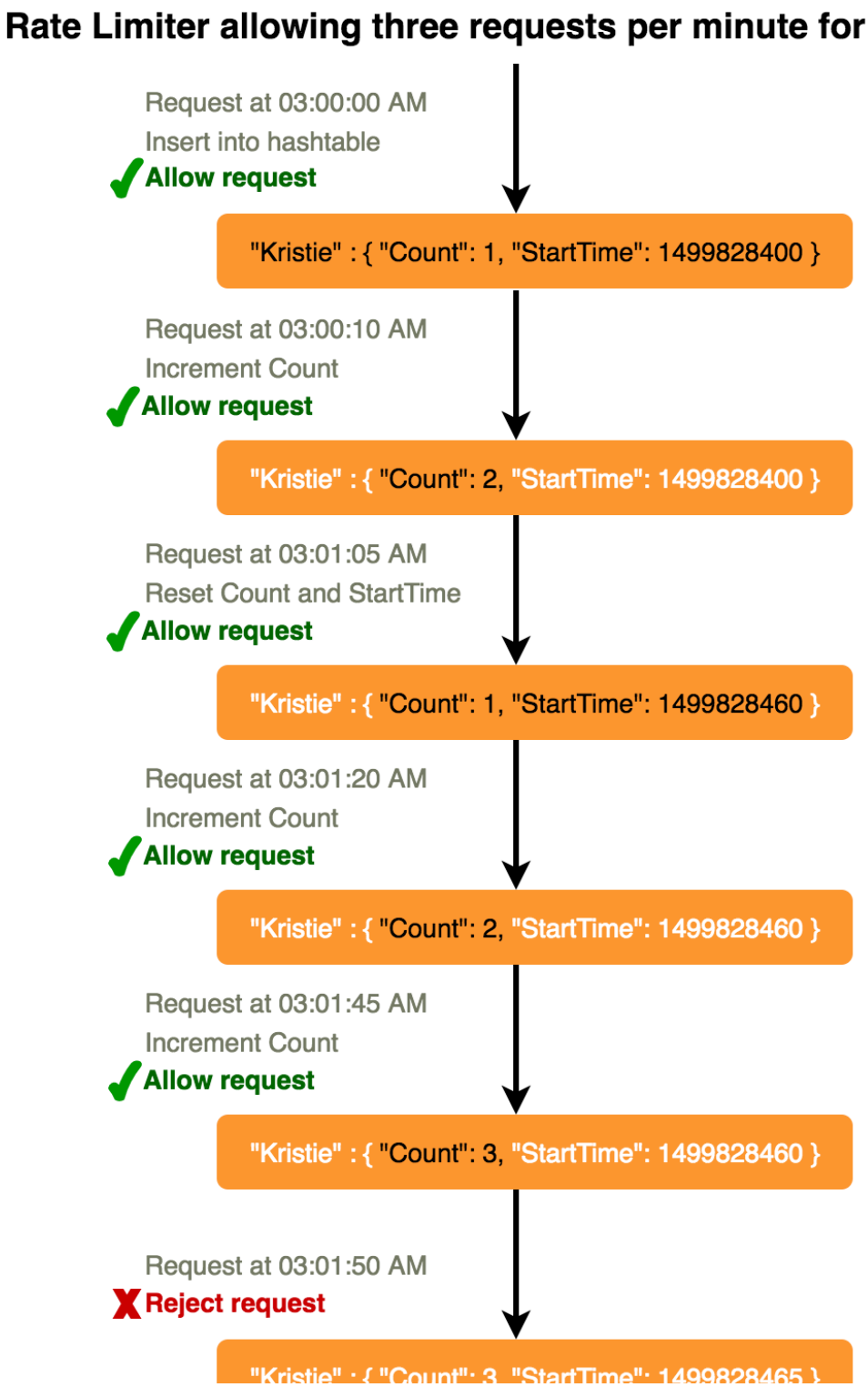
8. Basic System Design and Algorithm

Let's take the example where we want to limit the number of requests per user. Under this scenario, for each unique user, we would keep a count representing how many requests the user has made and a timestamp when we started counting the requests. We can keep it in a hashtable, where the 'key' would be the 'UserID' and 'value' would be a structure containing an integer for the 'Count' and an integer for the Epoch time:

	Key :	Value
E.g.,	UserID :	{ Count, StartTime }
	Kristie :	{ 3, 1499818564 }

Let's assume our rate limiter is allowing three requests per minute per user, so whenever a new request comes in, our rate limiter will perform following steps:

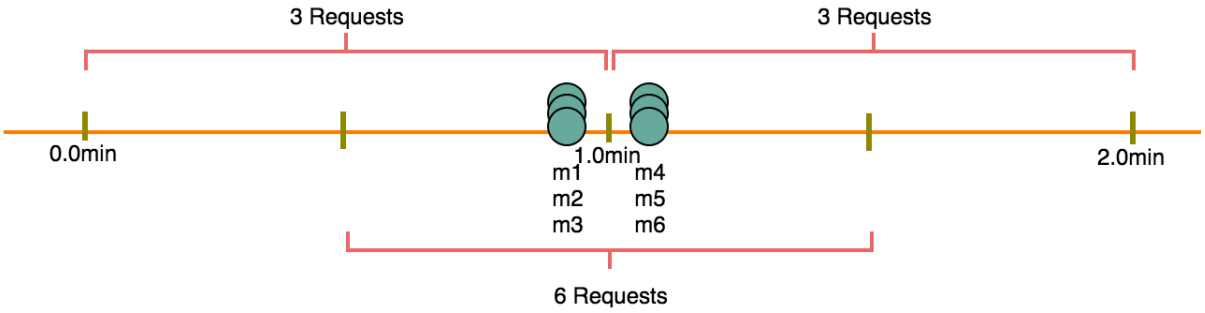
1. If the 'UserID' is not present in the hash-table, insert it and set the 'Count' to 1 and 'StarteTime' to the current time (normalized to a minute) , and allow the request.
2. Otherwise, find the record of the 'UserID' and if 'CurrentTime – StartTime >= 1 min', set the 'StartTime' to the current time and 'Count' to 1, and allow the request.
3. If 'CurrentTime - StartTime <= 1 min' and
    - If 'Count < 3', increment the Count and allow the request.
    - If 'Count >= 3', reject the request.



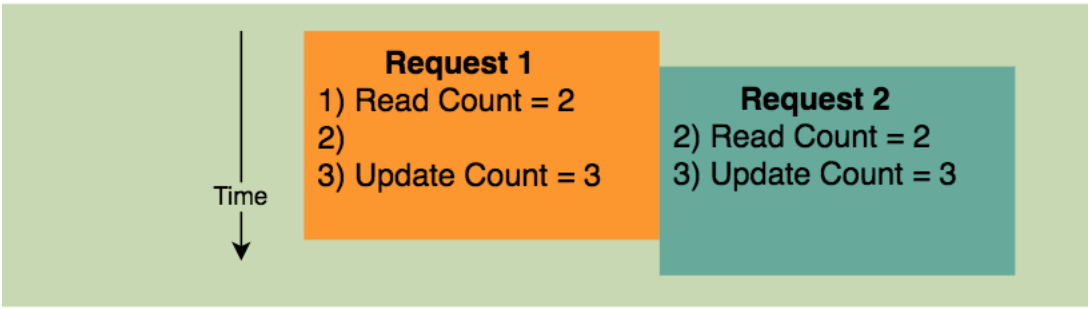
What are the some problems with our algorithm?

1. This is a **Fixed Window** algorithm, as we're resetting the 'StartTime' at the end of every minute, which means it can potentially allow twice the

number of requests per minute. Imagine if Kristie sends three requests at the last second of a minute, then she can immediately send three more requests at the very first second of the next minute, resulting in 6 requests in the span of two seconds. The solution to this problem would be a sliding window algorithm which we'll discuss later.



2. **Atomicity:** In a distributed environment, the "read-and-then-write" behavior can create a race condition. Imagine if Kristie's current 'Count' is "2" and that she issues two more requests. If two separate processes served each of these requests and concurrently read the Count before either of them updated it, each process would think the Kristie can have one more request and that she had not hit the rate limit.



If we are using Redis to store our key-value, one solution to resolve the atomicity problem is to use Redis lockfor the duration of the read-update operation. This, however, would come at the expense of slowing down concurrent requests from the same user and introducing another layer of complexity. We can use Memcached, but it would have comparable complications.

If we are using a simple hash-table, we can have a custom implementation for 'locking' each record to solve our atomicity problems.

How much memory would we need to store all of the user data? Let's assume the simple solution where we are keeping all of the data in a hash-table.

Let's assume 'UserID' takes 8 bytes. Let's also assume a 2 byte 'Count', which can count up to 65k, is sufficient for our use case. Although epoch time will need 4 bytes, we can choose to store only the minute and second part, which can fit into 2 bytes. Hence, we need total 12 bytes to store a user's data:

$$8 + 2 + 2 = 12 \text{ bytes}$$

Let's assume our hash-table has an overhead of 20 bytes for each record. If we need to track one million users at any time, the total memory we would need would be 32MB:

$$(12 + 20) \text{ bytes} * 1 \text{ million} => 32\text{MB}$$

If we assume that we would need a 4-byte number to lock each user's record to resolve our atomicity problems, we would require a total 36MB memory.

This can easily fit on a single server, however we would not like to route all of our traffic through a single machine. Also, if we assume a rate limit of 10 requests per second, this would translate into 10 million QPS for our rate limiter! This would be too much for a single server. Practically we can assume we would use Redis or Memcached kind of a solution in a distributed setup. We'll be storing all the data in the remote Redis servers, and all the Rate Limiter servers will read (and update) these servers before serving or throttling any request.

9. Sliding Window algorithm

We can maintain a sliding window if we can keep track of each request per user. We can store the timestamp of each request in a Redis Sorted Set in our 'value' field of hash-table.

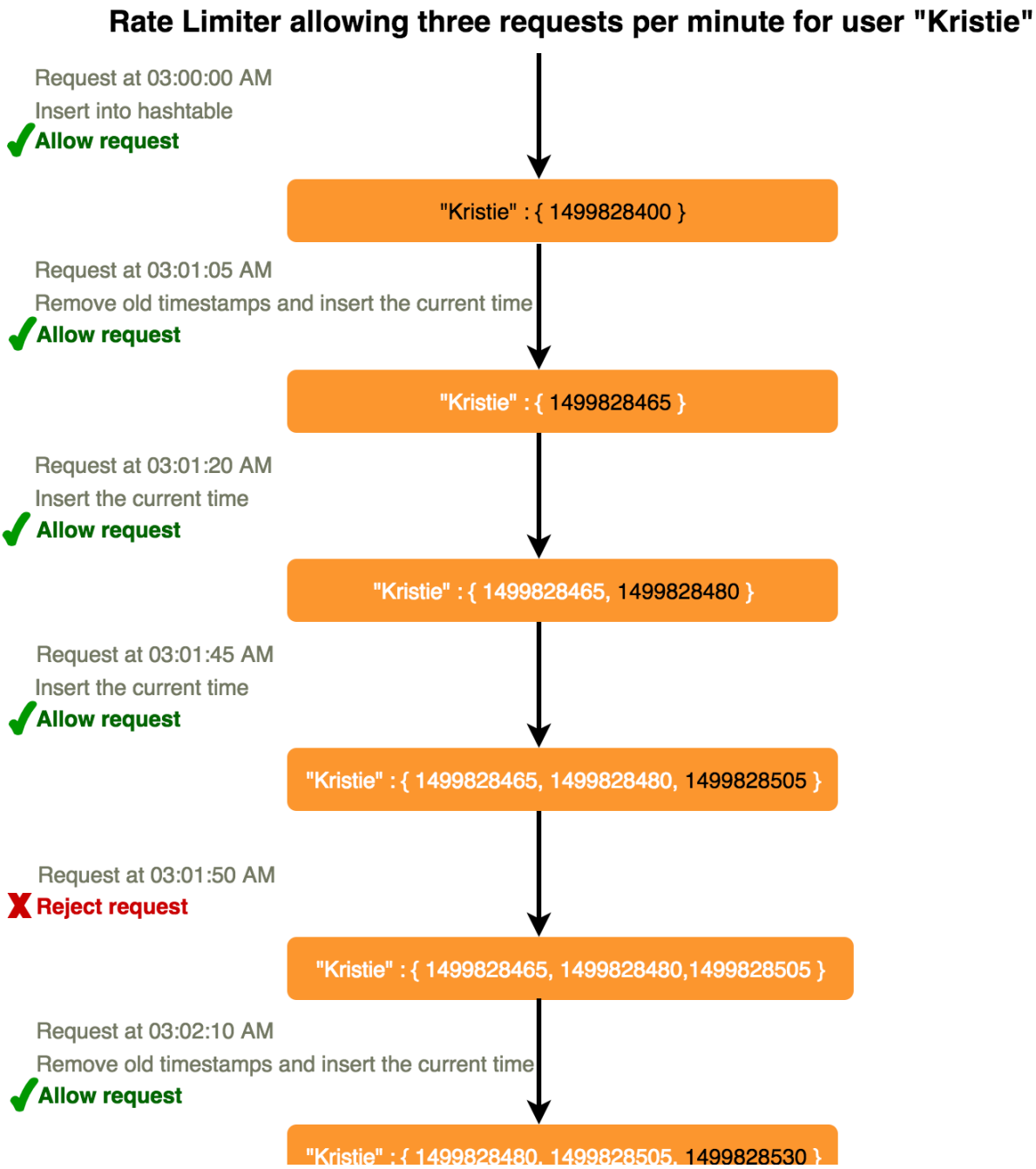
	Key :	Value
E.g.,	UserID :	{ Sorted Set <UnixTime> }
	Kristie :	{ 1499818000, 1499818500, 1499818860 }

Let’s assume our rate limiter is allowing three requests per minute per user, so whenever a new request comes in the Rate Limiter will perform following steps:

1. Remove all the timestamps from the Sorted Set that are older than “CurrentTime - 1 minute”.

2. Count the total number of elements in the sorted set. Reject the request if this count is greater than our throttling limit of “3”.

3. Insert the current time in the sorted set, and accept the request.



How much memory would we need to store all of the user data for sliding window? Let's assume 'UserID' takes 8 bytes. Each epoch time will require 4 bytes. Let's suppose we need a rate limiting of 500 requests per hour. Let's assume 20 bytes overhead for hash-table and 20 bytes overhead for the Sorted Set. At max, we would need a total of 12KB to store one user's data:

8 + (4 + 20 (sorted set overhead)) * 500 + 20 (hash-table overhead) = 12KB

Here are reserving 20 bytes overhead per element. In a sorted set, we can assume that we need at least two pointers to maintain order among elements. One pointer to the previous element and one to the next element. On a 64bit machine, each pointer will cost 8 bytes. So we will need 16 bytes for pointers. We added an extra word (4 bytes) for storing other overhead.

If we need to track one million users at any time, total memory we would need would be 12GB:

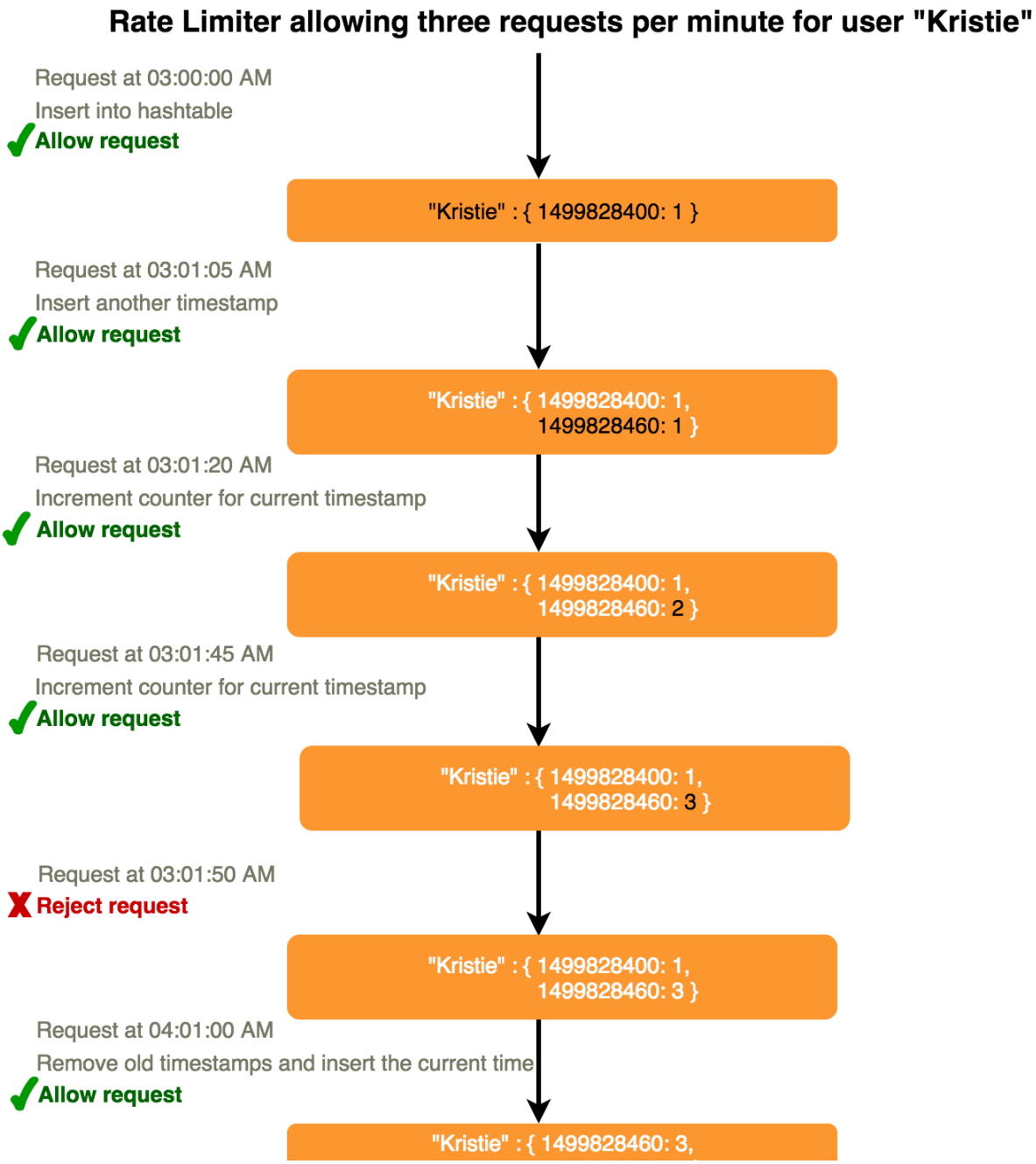
12KB * 1 million ~= 12GB

Sliding Window Algorithm is taking a lot of memory compared to the Fixed Window; this would be a scalability issue. What if we can combine the above two algorithms to optimize our memory usage?

10. Sliding Window with Counters

What if we keep track of request counts for each user using multiple fixed time windows, e.g., 1/60th the size of our rate limit's time window. For example, if we've an hourly rate limit, we can keep a count for each minute and calculate the sum of all counters in the past hour when we receive a new request to calculate the throttling limit. This would reduce our memory footprint. Let's take an example where we rate limit at 500 requests per hour with an additional limit of 10 requests per minute. This means that when the sum of the counters with timestamps in the past hour exceeds the request threshold (500), Kristie has exceeded the rate limit. In addition to that, she can't send more than ten requests per minute. This would be a reasonable and a practical consideration, as none of the real users would send frequent requests. Even if they do, they will see success with retries since their limits get reset every minute.

We can store our counters in a Redis Hash - as it offers extremely efficient storage for fewer than 100 keys. When each request increments a counter in the hash, it also sets the hash to expire an hour later. We'll normailze each 'time' to a minute.



How much memory we would need to store all the user data for sliding window with counters? Let's assume 'UserID' takes 8 bytes. Each epoch time will need 4 bytes and the Counter would need 2 bytes. Let's suppose we need a rate limiting of 500 requests per hour. Assume 20 bytes overhead for hash-table and 20 bytes for Redis hash. Since we'll keep a count for each minute, at max, we would need 60 entries for each user. We would need a total of 1.6KB to store one user's data:

8 + (4 + 2 + 20 (Redis hash overhead)) * 60 + 20 (hash-table overhead) = 1.6KB

If we need to track one million users at any time, total memory we would need would be 1.6GB:

1.6KB * 1 million ~= 1.6GB

So, our 'Sliding Window with Counters' algorithm uses 86% less memory than simple sliding window algorithm

11. Data Sharding and Caching

We can shard based on the 'UserID' to distribute user's data. For fault tolerance and replication we should use Consistent Hashing. If we want to have different throttling limits for different APIs, we can choose to shard per user per API. Take the example of URL Shortner, we can have different rate limiter for **creatURL()** and **deleteUR()** APIs for each user or IP.

If our APIs are partitioned, a practical consideration could be to have a separate (somewhat smaller) rate limiter for each API shard as well. Let's take the example of our URL Shortener, where we want to limit each user to not create more than 100 short URLs per hour. Assuming, we are using **Hash-Based Partitioning** for our **creatURL()**API, we can rate limit each partition to allow a user to create not more than three short URLs per minute, in addition to 100 short URLs per hour.

Our system can get huge benefits from caching recent active users. Application servers before hitting backend servers can quickly check if the cache has the desired record. Our rate limiter can greatly benefit from the **Write-back cache**, by updating all counters and timestamps in cache only. The write to the permanent storage can be done at fixed intervals. This way we can ensure minimum latency added to the user's requests by the rate limiter. The reads can always hit the cache first, this will be extremely useful once the user has hit their maximum limit and the rate limiter will only be reading data without any updates.

Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.

12. Should we rate limit by IP or by user?

Let's discuss pros and cons of using each one of these schemes:

IP: In this scheme, we throttle request per-IP, although it's not really optimal in terms of differentiating between 'good' and 'bad' actors, it's still better than not have rate limiting at all. The biggest problem with IP based throttling is when multiple users share a single public IP like in an internet cafe or smartphone

users that are using the same gateway. One bad user can cause throttling to other users. Another issue could arise while caching IP-based limits, as there are a huge number of IPv6 addresses available to a hacker from even one computer, it's trivial to make a server run out of memory tracking IPv6 addresses!

User: Rate limiting can be done on APIs after user authentication. Once authenticated, the user will be provided with a token which the user will pass with each request. This will ensure that we will rate limit against a particular API that has a valid authentication token. But what if we have to rate limit on the login API itself? The weakness of this rate-limiting would be that a hacker can perform a denial of service attack against a user by entering wrong credentials up to the limit, after that the actual user will not be able to login.

How about if we combine the above two schemes?

Hybrid: A right approach could be to do both per-IP and per-user rate limiting, as they both have weaknesses when implemented alone. Though, this will result in more cache entries with more details per entry hence requiring more memory and storage.

Designing Twitter Search

Twitter is one of the largest social networking service where users can share photos, news, and text-based messages. In this chapter, we will design a service that can store and search user tweets.

1. What is Twitter Search?

Twitter users can update their status whenever they like. Each status consists of plain text, and our goal is to design a system that allows searching over all the user statuses.

2. Requirements and Goals of the System

- Let's assume Twitter has 1.5 billion total users with 800 million daily active users.
- On the average Twitter gets 400 million status updates every day.
- Average size of a status is 300 bytes.
- Let's assume there will be 500M searches every day.
- The search query will consist of multiple words combined with AND/OR.

We need to design a system that can efficiently store and query user statuses.

3. Capacity Estimation and Constraints

Storage Capacity: Since we have 400 million new statuses every day and each status on average is 300 bytes, therefore total storage we need, will be:

400M * 300 => 112GB/day

Total storage per second:

112GB / 86400 sec ~= 1.3MB/second

4. System APIs

We can have SOAP or REST APIs to expose functionality of our service; following could be the definition of search API:

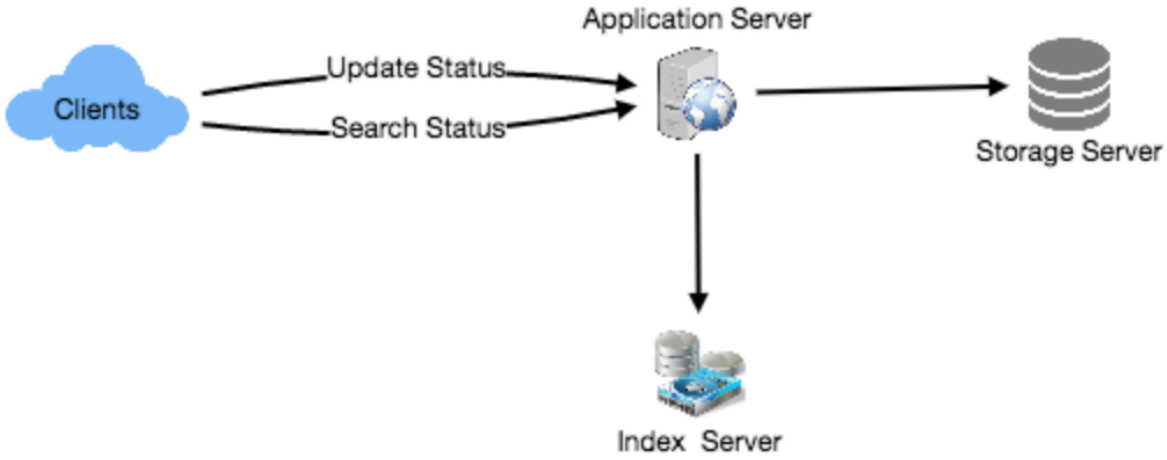
search(api_dev_key, search_terms, maximum_results_to_return, sort, page_token)

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.
maximum_results_to_return (number): Number of status messages 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.

Returns: (JSON)
A JSON containing information about a list of status messages matching the search query. Each result entry can have the user ID & name, status text, status ID, creation time, number of likes, etc.

5. High Level Design

At the high level, we need to store all the statues in a database, and also build an index that can keep track of which word appears in which status. This index will help us quickly find statuses that users are trying to search.



High level design for Twitter search

6. Detailed Component Design

1. Storage: We need to store 112GB of new data every day. Given this huge amount of data, we need to come up with a data partitioning scheme that will be

efficiently distributing it onto multiple servers. If we plan for next five years, we will need following storage:

112GB * 365days * 5 => 200 TB

If we never want to be more than 80% full, we would need 240TB. Let's assume that we want to keep an extra copy of all the statuses for fault tolerance, then our total storage requirement will be 480 TB. If we assume a modern server can store up to 4TB of data, then we would need 120 such servers to hold all of the required data for next five years.

Let's start with a simplistic design where we store our statuses in a MySQL database. We can assume to store our statuses in a table having two columns, StatusID and StatusText. Let's assume we partition our data based on StatusID. If our StatusIDs are system-wide unique, we can define a hash function that can map a StatusID to a storage server, where we can store that status object.

How can we create system wide unique StatusIDs? If we are getting 400M new statuses each day, then how many status objects we can expect in five years?

400M * 365 days * 5 years => 730 billion

This means we would need a five bytes number to identify StatusIDs uniquely. Let's assume we have a service that can generate a unique StatusID whenever we need to store an object (we will discuss this in detail later). We can feed the StatusID to our hash function to find the storage server and store our status object there.

2. Index: What should our index look like? Since our status queries will consist of words, therefore, let's build our index that can tell us which word comes in which status object. Let's first estimate how big our index will be. If we want to build an index for all the English words and some famous nouns like people names, city names, etc., and if we assume that we have around 300K English words and 200K nouns, then we will have 500k total words in our index. Let's assume that the average length of a word is five characters. If we are keeping our index in memory, we would need 2.5MB of memory to store all the words:

500K * 5 => 2.5 MB

Let's assume that we want to keep the index in memory for all the status objects for only past two years. Since we will be getting 730B status objects in 5 years, this will give us 292B status messages in two years. Given that, each StatusID will be 5 bytes, how much memory will we need to store all the StatusIDs?

292B * 5 => 1460 GB

So our index would be like a big distributed hash table, where 'key' would be the word, and 'value' will be a list of StatusIDs of all those status objects which contain that word. Assuming on the average we have 40 words in each status and since we will not be indexing prepositions and other small words like 'the', 'an', 'and' etc., let's assume we will have around 15 words in each status that need to be indexed. This means each StatusID will be stored 15 times in our index. So total memory will need to store our index:

$$(1460 * 15) + 2.5MB \sim= 21\ TB$$

Assuming a high-end server has 144GB of memory, we would need 152 such servers to hold our index.

We can shard our data based on two criteria:

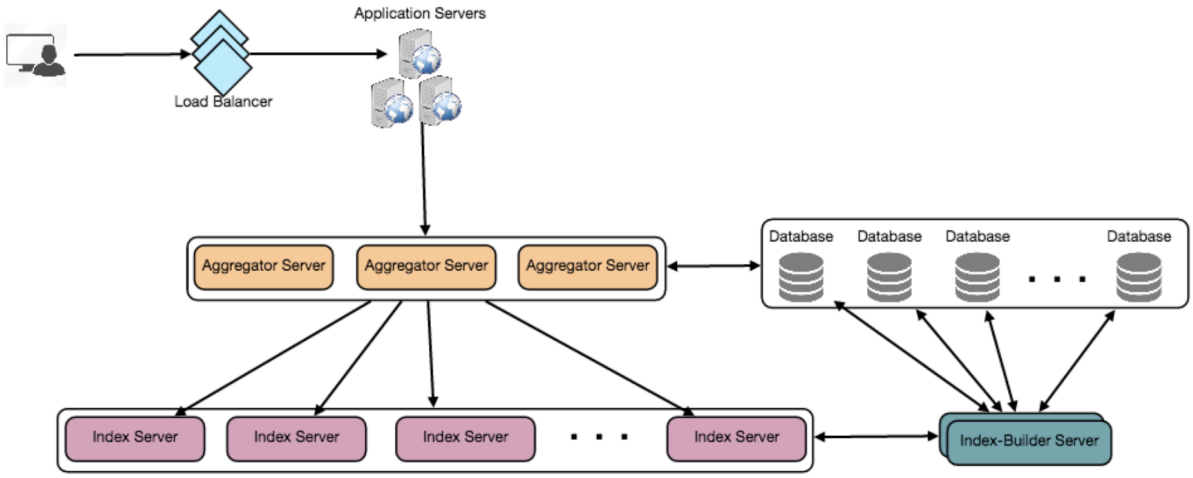
Sharding based on Words: While building our index, we will iterate through all the words of a status and calculate the hash of each word to find the server where it would be indexed. To find all statuses containing a specific word we have to query only that server which contains this word.

We have a couple of issues with this approach:

1. What if a word becomes hot? There would be a lot of queries on the server holding that word. This high load will affect the performance of our service.
2. Over time some words can end up storing a lot of StatusIDs compared to others, therefore, maintaining a uniform distribution of words while statuses are growing is quite difficult.

To recover from these situations either we have to repartition our data or use Consistent Hashing.

Sharding based on the status object: While storing, we will pass the StatusID to our hash function to find the server and index all the words of the status on that server. While querying for a particular word, we have to query all the servers, and each server will return a set of StatusIDs. A centralized server will aggregate these results to return them to the user.



Detailed component design

7. Fault Tolerance

What will happen when an index server dies? We can have a secondary replica of each server, and if the primary server dies it can take control after the failover. Both primary and secondary servers will have the same copy of the index.

What if both primary and secondary servers die at the same time? We have to allocate a new server and rebuild the same index on it. How can we do that? We don't know what words/statuses were kept on this server. If we were using 'Sharding based on the status object', the brute-force solution would be to iterate through the whole database and filter StatusIDs using our hash function to figure out all the required Statuses that will be stored on this server. This would be inefficient and also during the time when the server is being rebuilt we will not be able to serve any query from it, thus missing some Statuses that should have been seen by the user.

How can we efficiently retrieve a mapping between Statuses and index server? We have to build a reverse index that will map all the StatusID to their index server. Our Index-Builder server can hold this information. We will need to build a Hashtable, where the 'key' would be the index server number and the 'value' would be a HashSet containing all the StatusIDs being kept at that index server. Notice that we are keeping all the StatusIDs in a HashSet, this will enable us to add/remove Statuses from our index quickly. So now whenever an index server has to rebuild itself, it can simply ask the Index-Builder server for all the Statuses it needs to store, and then fetch those statuses to build the index. This approach will surely be quite fast. We should also have a replica of Index-Builder server for fault tolerance.

8. Cache

To deal with hot status objects, we can introduce a cache in front of our database. We can use Memcache , which can store all such hot status objects in memory. Application servers before hitting backend database can quickly check if the cache has that status object. 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.

9. Load Balancing

We can add Load balancing 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 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 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, it won't take 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 backend server about their load and adjusts traffic based on that.

10. Ranking

How about if we want to rank the search results by social graph distance, popularity, relevance, etc?

Let's assume we want to rank statuses on popularity, like, how many likes or comments a status is getting, etc. In such a case our ranking algorithm can calculate a 'popularity number' (based on the number of likes etc.), and store it with the index. Each partition can sort the results based on this popularity number before returning results to the aggregator server. The aggregator server combines all these results, sort them based on the popularity number and sends the top results to the user.

Designing a Web Crawler

Let's design a Web Crawler that will systematically browse and download the World Wide Web. Web crawlers are also known as web spiders, robots, worms, walkers, and bots.

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 needs 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, 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? Assuming 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, containing 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 weeks * 7 days * 86400 sec) ~= 6200 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 be 100KB. With each page if we are storing 500 bytes of metadata, total storage we would need:

15B * (100KB + 500) ~= 1.5 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 petabytes / 0.7 ~= 2.14 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. Establishing 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

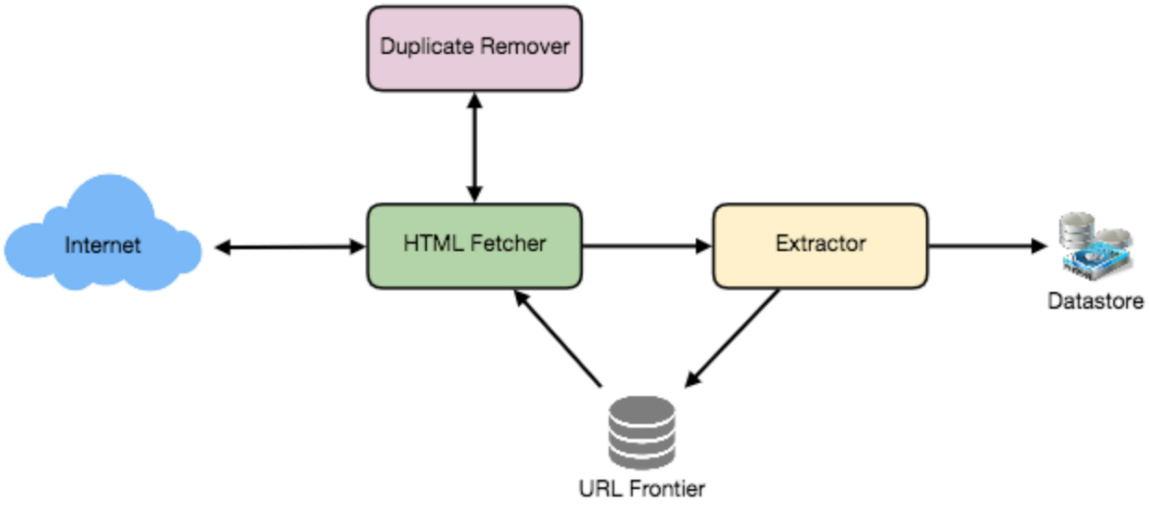
There are two important characteristics of the Web that makes Web crawling a very difficult task:

1. Large volume of Web pages: A large volume of web page 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 has been 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 same content is not extracted twice unintentionally.
5. Datastore: To store retrieve pages and URL and other metadata.



6. Detailed Component Design

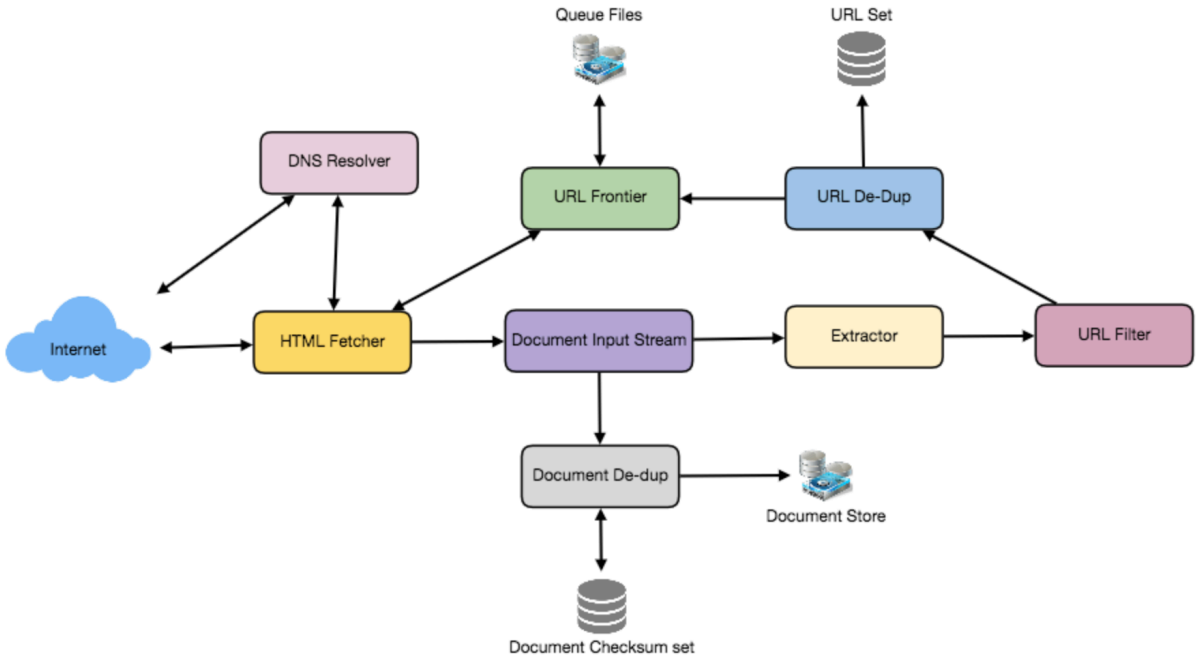
Let's assume our crawler is running on one server, and all the crawling is done by multiple working threads, where each working thread performs all the steps needed to download and process a document in a loop.

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 our URL frontier would be? The size would be in the hundreds of millions of URLs. Hence, we need to store our URLs on disk. We can implement our queues in such a way that they have separate buffers for enqueuing 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 robots 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 contents multiple times. To prevent processing 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 bytes => 120 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 in a 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 2 bytes for checksum, we would need:

$$15B * 2 \text{ bytes} => 30 \text{ GB}$$

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 to 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 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. Extended 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, extended 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 extended 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 into 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.

AOPIC algorithm (Adaptive Online Page Importance Computation), can help mitigating common types of bot-traps. AOPIC solves this problem by using a credit system.

1. Start with a set of N seed pages.
2. Before crawling starts, allocate a fixed X amount of credit to each page.

3. Select a page P with the highest amount of credit (or select a random page if all pages have the same amount of credit).
4. Crawl page P (let's say that P had 100 credits when it was crawled).
5. Extract all the links from page P (let's say there are 10 of them).
6. Set the credits of P to 0.
7. Take a 10% "tax" and allocate it to a Lambda page.
8. Allocate an equal amount of credits each link found on page P from P's original credit after subtracting the tax, so: (100 (P credits) - 10 (10% tax))/10 (links) = 9 credits per each link.
9. Repeat from step 3.

Since the Lambda page continuously collects the tax, eventually it will be the page with the largest amount of credit, and we'll have to "crawl" it. By crawling the Lambda page, we just take its credits and distribute them equally to all the pages in our database.

Since bot traps only give internal links credits and they rarely get credit from the outside, they will continually leak credits (from taxation) to the Lambda page. The Lambda page will distribute that credits out to all the pages in the database evenly, and upon each cycle, the bot trap page will lose more and more credits until it has so little credits that it almost never gets crawled again. This will not happen with good pages because they often get credits from backlinks found on other pages.

Designing Facebook's Newsfeed

Let's design Facebook's Newsfeed, which would contain posts, photos, videos and status updates from all the people and pages a user follows.

1. What is Facebook's newsfeed?

Newsfeed is the constantly updating list of stories in the middle of Facebook's homepage. It includes status updates, photos, videos, links, app activity and likes from people, pages, and groups that a user follows on Facebook. In other words, it's a compilation of a complete scrollable version of your and your friends' life story from photos, videos, locations, status updates and other activities.

Any social media site you design - Twitter, Instagram or Facebook, you will need some sort of newsfeed system to display updates from friends and followers.

2. Requirements and Goals of the System

Let's design a newsfeed for Facebook with the following requirements:

Functional requirements:

1. Newsfeed will be generated based on the posts from the people, pages, and groups that a user follows.
2. A user may have many friends and follow a large number of pages/groups.
3. Feeds may contain images, videos or just text.
4. Our service should support appending new posts, as they arrive, to the newsfeed for all active users.

Non-functional requirements:

1. Our system should be able to generate any user's newsfeed in real-time - maximum latency seen by the end user could be 2s.
2. A post shouldn't take more than 5s to make it to a user's feed assuming a new newsfeed request comes in.

3. Capacity Estimation and Constraints

Let's assume on average a user has 300 friends and follows 200 pages.

Traffic estimates: Let's assume 300M daily active users, with each user fetching their timeline an average of five times a day. This will result in 1.5B newsfeed requests per day or approximately 17,500 requests per second.

Storage estimates: On average, let's assume, we would need to have around 500 posts in every user's feed that we want to keep in memory for a quick fetch. Let's also assume that on average each post would be 1KB in size. This would mean that we need to store roughly 500KB of data per user. To store all this data for all the active users, we would need 150TB of memory. If a server can hold 100GB, we would need around 1500 machines to keep the top 500 posts in memory for all active users.

4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definition of the API for getting the newsfeed:

```
getUserFeed(api_dev_key, user_id, since_id, count, max_id, exclude_replies)
```

Parameters:
api_dev_key (string): The API developer key of a registered account. This can be used to, among other things, throttle users based on their allocated quota.
user_id (number): The ID of the user for whom the system will generate the newsfeed.
since_id (number): Optional; returns results with an ID greater than (that is, more recent than) the specified ID. **count (number):** Optional; specifies the number of feed items to try and retrieve, up to a maximum of 200 per distinct request.
max_id (number): Optional; returns results with an ID less than (that is, older than) or equal to the specified ID.
exclude_replies(boolean): Optional; this parameter will prevent replies from appearing in the returned timeline.

Returns: (JSON) Returns a JSON object containing a list of feed items.

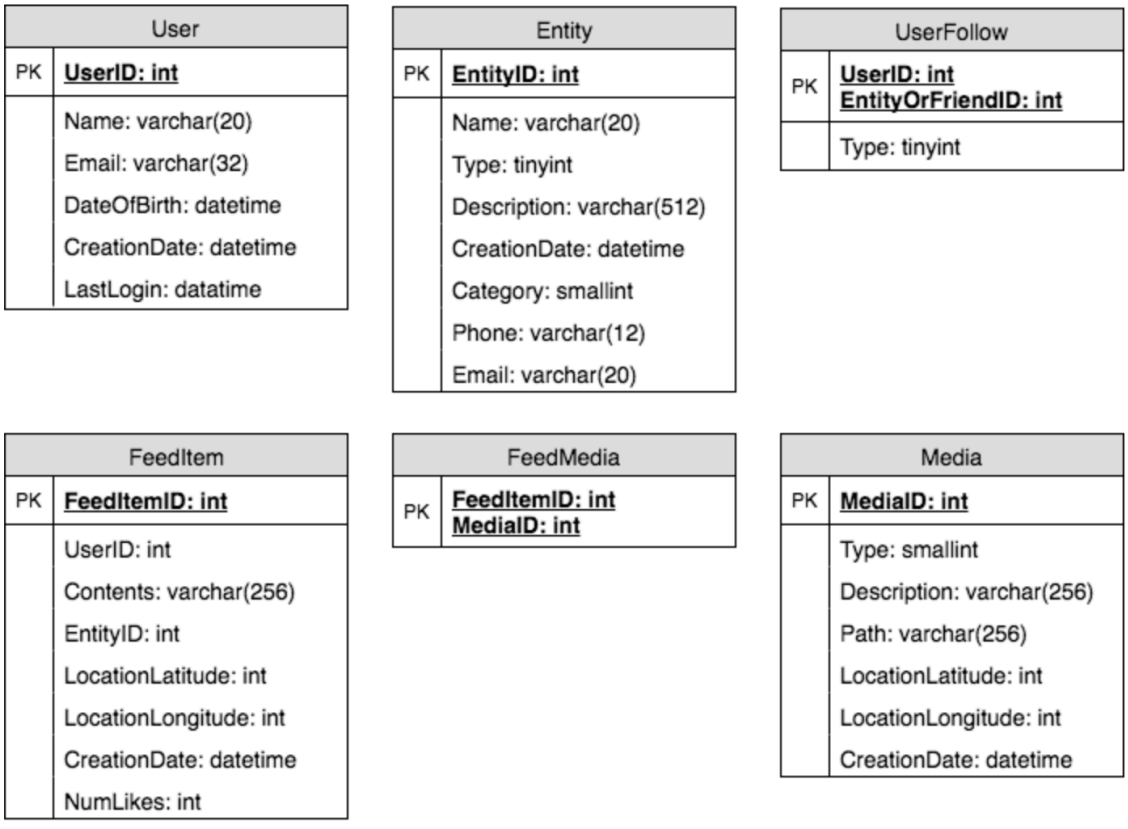
5. Database Design

There are three basic objects: User, Entity (e.g., page, group, etc.) and FeedItem (or Post). Here are some observations about the relationships between these entities:

- A User can follow entities and can become friends with other users.
- Both users and entities can post FeedItems which can contain text, images or videos.

- Each FeedItem will have a UserID which would point to the User who created it. For simplicity, let's assume that only users can create feed items, although on Facebook, Pages can post feed item too.
- Each FeedItem can optionally have an EntityID pointing to the page or the group where that post was created.

If we are using a relational database, we would need to model two relations: User-Entity relation and FeedItem-Media relation. Since each user can be friends with many people and follow a lot of entities, we can store this relation in a separate table. The "Type" column in "UserFollow" identifies if the entity being followed is a User or Entity. Similarly, we can have a table for FeedMedia relation.



6. High Level System Design

At a high level this problem can be divided into two parts:

Feed generation: Newsfeed is generated from the posts (or feed items) of users and entities (pages and groups) that a user follows. So, whenever our system

receives a request to generate the feed for a user (say Jane), we will perform following steps:

1. Retrieve IDs of all users and entities that Jane follows.
2. Retrieve latest, most popular and relevant posts for those IDs. These are the potential posts that we can show in Jane's newsfeed.
3. Rank these posts, based on the relevance to Jane. This represents Jane's current feed.
4. Store this feed in the cache and return top posts (say 20) to be rendered on Jane's feed.
5. On the front-end when Jane reaches the end of her current feed, she can fetch next 20 posts from the server and so on.

One thing to notice here is that we generated the feed once and stored it in cache. What about new incoming posts from people that Jane follows? If Jane is online, we should have a mechanism to rank and add those new posts to her feed. We can periodically (say every five minutes) perform the above steps to rank and add the newer posts to her feed. Jane can then be notified that there are newer items in her feed that she can fetch.

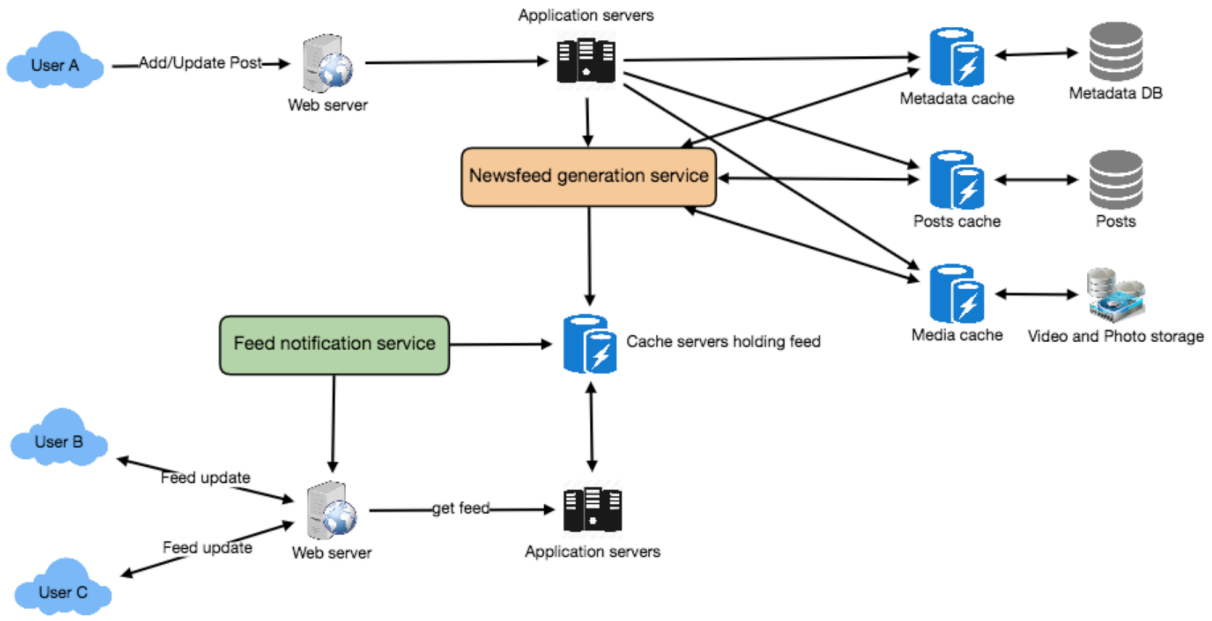
Feed publishing: Whenever Jane loads her newsfeed page, she has to request and pull feed items from the server. When she reaches the end of her current feed, she can pull more data from the server. For newer items either the server can notify Jane and then she can pull, or the server can push these new posts. We will discuss these options in detail later.

At a high level, we would need following components in our Newsfeed service:

1. **Web servers:** To maintain a connection with the user. This connection will be used to transfer data between the user and the server.
2. **Application server:** To execute the workflows of storing new posts in the database servers. We would also need some application servers to retrieve and push the newsfeed to the end user.
3. **Metadata database and cache:** To store the metadata about Users, Pages and Groups.
4. **Posts database and cache:** To store metadata about posts and their contents.
5. **Video and photo storage, and cache:** Blob storage, to store all the media included in the posts.
6. **Newsfeed generation service:** To gather and rank all the relevant posts for a user to generate newsfeed and store in the cache. This service would also receive live updates and will add these newer feed items to any user's timeline.

7. **Feed notification service:** To notify the user that there are newer items available for their newsfeed.

Following is the high-level architecture diagram of our system. User B and C are following User A.



Facebook Newsfeed Architecture

7. Detailed Component Design

Let's discuss different components of our system in detail.

a. Feed generation

Let's take the simple case of the newsfeed generation service fetching most recent posts from all the users and entities that Jane follows; the query would look like this:

```
SELECT FeedItemID FROM FeedItem WHERE SourceID in (
        SELECT EntityOrFriendID FROM UserFollow WHERE UserID = <current_user_id>
)
ORDER BY CreationDate DESC
LIMIT 100
```

Here are issues with this design for the feed generation service:

1. Crazy slow for users with a lot of friends/follows as we have to perform sorting/merging/ranking of a huge number of posts.
2. We generate the timeline when a user loads their page. This would be quite slow and have a high latency.
3. 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.
4. For live updates, the server pushing (or notifying about) newer posts to users could lead to very heavy loads, especially for people or pages that have a lot of followers. To improve the efficiency, we can pre-generate the timeline and store it in a memory.

Offline generation for newsfeed: We can have dedicated servers that are continuously generating users' newsfeed and storing them in memory. So, whenever a user requests for the new posts for their feed, we can simply serve it from the pre-generated, stored location. Using this scheme user's newsfeed is not compiled on load, but rather on a regular basis and returned to users whenever they request for it.

Whenever these servers need to generate the feed for a user, they would first query to see what was the last time the feed was generated for that user. Then, new feed data would be generated from that time onwards. We can store this data in a hash table, where the "key" would be UserID and "value" would be a STRUCT like this:

```
Struct {
   LinkedHashMap<FeedItemID> feedItems;
   DateTime lastGenerated;
}
```

We can store FeedItemIDs in a Linked HashMap, which will enable us to not only jump to any feed item but also iterate through the map easily. Whenever users want to fetch more feed items, they can send the last FeedItemID they currently see in their newsfeed, we can then jump to that FeedItemID in our linked hash map and return next batch / page of feed items from there.

How many feed items should we store in memory for a user's feed? Initially, we can decide to store 500 feed items per user, but this number can be adjusted later based on the usage pattern. For example, if we assume that one page of user's feed has 20 posts and most of the users never browse more than ten pages of their feed, we can decide to store only 200 posts per user. For any user, who wants to see more posts (more than what is stored in memory) we can always query backend servers.

Should we generate (and keep in memory) newsfeed for all users? There will be a lot of users that don't login frequently. Here are a few things we can do to handle this. A simpler approach could be to use an LRU based cache that can remove users from memory that haven't accessed their newsfeed for a long time. A smarter solution can figure out the login pattern of users to pre-generate their newsfeed, e.g., At what time of the day a user is active? Which days of the week a user accesses their newsfeed? etc.

Let's now discuss some solutions to our "live updates" problems in the following section.

b. Feed publishing

The process of pushing a post to all the followers is called a fanout. By analogy, the push approach is called fanout-on-write, while the pull approach is called fanout-on-load. Let's discuss different options of publishing feed data to users.

1. **"Pull" model or Fan-out-on-load:** This method involves keeping all the recent feed data in memory so that users can pull it from the server whenever they need it. Clients can pull the feed data on a regular basis or manually whenever they need it. Possible problems with this approach are a) New data might not be shown to the users until they issue a pull request, b) It's hard to find the right pull cadence, as most of the time pull requests will result in an empty response if there is no new data, causing waste of resources.

2. **"Push" model or Fan-out-on-write:** For a push system, once a user has published a post, we can immediately push this post to all her followers. The advantage is that when fetching feed, you don't need to go through your friends list and get feeds for each of them. It significantly reduces read operations. 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 that when a user has millions of followers (or a celebrity-user), the server has to push updates to a lot of people.

3. **Hybrid:** Another interesting approach to handle feed data could be to use a hybrid approach, i.e., to do a combination of fan-out-on-write and fan-out-on-load. Specifically, we can stop pushing posts from users with a high number of followers (a celebrity user) and only push data for those users who have a few hundred (or thousand) followers. For celebrity users, we can let the followers pull the updates. Since the push operation can be extremely costly for users who have a lot of friends or followers therefore, by disabling fanout for them, we can save a huge number of resources. Another alternate approach could be that once a user publishes a post; we can limit the fanout to only her online friends. Also, to get benefits of both

the approaches, a combination of push to notify and pull for serving end users is a great way to go. Purely push or pull model is less versatile.

How many feed items can we return to the client in each request? We should have a maximum limit for the number of items a user can fetch in one request (say 20). But we should let clients choose to specify how many feed items they want with each request, as the user may like to fetch a different number of posts depending on the device (mobile vs desktop).

Should we always notify users if there are new posts available for their newsfeed? It could be useful for users to get notified whenever new data is available. However, on mobile devices, where data usage is relatively expensive, it can consume unnecessary bandwidth. Hence, at least for mobile devices, we can choose not to push data, instead let users "Pull to Refresh" to get new posts.

8. Feed Ranking

The most straightforward way to rank posts in a newsfeed is by the creation time of the posts. But today's ranking algorithms are doing a lot more than that to ensure "important" posts are ranked higher. The high-level idea of ranking is to first select key "signals" that make a post important and then figure out how to combine them to calculate a final ranking score.

More specifically, we can select features that are relevant to the importance of any feed item, e.g. number of likes, comments, shares, time of the update, whether the post has images/videos, etc., and then, a score can be calculated using these features. This is generally enough for a simple ranking system. A better ranking system can significantly improve itself by constantly evaluating if we are making progress in user stickiness, retention, ads revenue, etc.

9. Data Partitioning

a. Sharding posts and metadata

Since we have a huge number of new posts every day and our read load is extremely high too, we need to distribute our data onto multiple machines such that we can read/write it efficiently. For sharding our databases that are storing posts and their metadata, we can have a similar design as discussed under Designing Twitter.

b. Sharding feed data

For feed data, which is being stored in memory, we can partition it based on UserID. We can try storing all the data of a user on one server. when storing, we can pass the UserID to our hash function that will map the user to a cache server

where we will store the user's feed objects. Also, for any given user, since we don't expect to store more than 500 FeedItmeIDs, we wouldn't run into a scenario where feed data for a user doesn't fit on a single server. To get the feed of a user, we would always have to query only one server. For future growth and replication, we must use Consistent Hashing.

Designing Yelp or Nearby Friends

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.

1. Why Yelp or Proximity Server?

Proximity servers are used to discover nearby attractions like places, events, etc. If you haven't used yelp.combefore, please try it before proceeding. You can search for nearby restaurants, theaters, etc., and spend some time understanding different options 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 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 location can have 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 => 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. 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 with, 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 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 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:

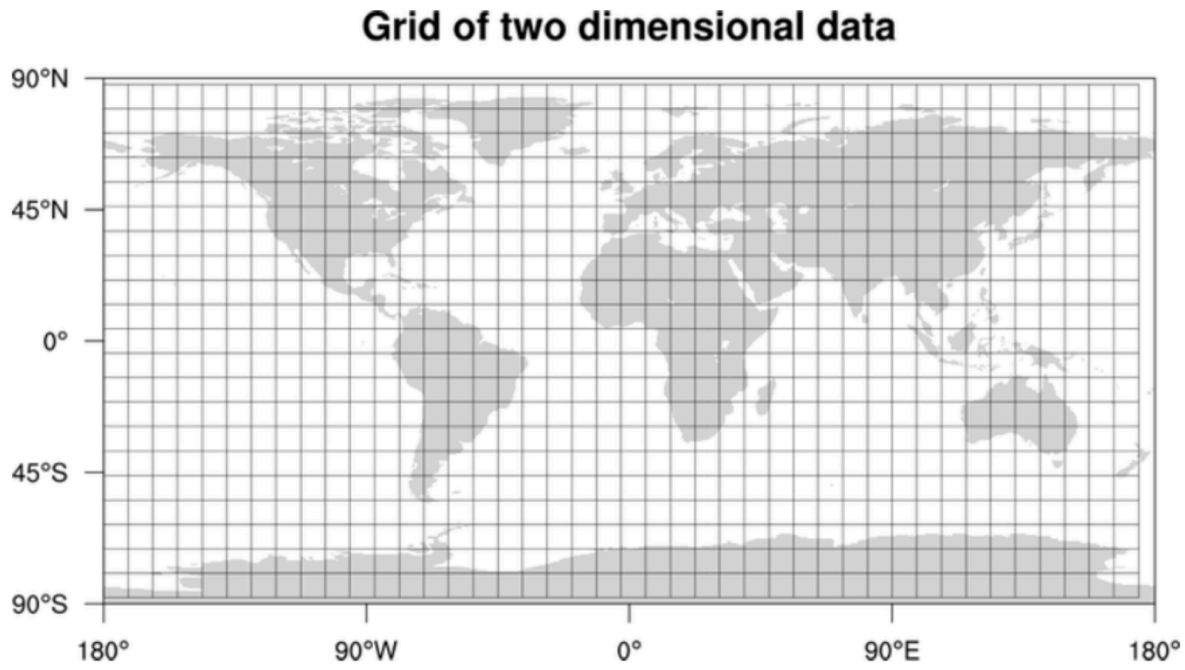
```
Select * from Places where Latitude between X-D and X+D and Longitude between Y-D and Y+D
```

How efficient this query would 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 certain range of longitude and latitude. This scheme would enable us to query only a few grids to find nearby places. Based on given location and radius, we can find all the nearby grids and then only query these grids to find nearby places.



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' would be the grid number and 'value' would be 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 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, therefore we would need 4GB of memory (ignoring hash table overhead) to store the index.

$$(4 * 20M) + (8 * 500M) \sim= 4 \text{ GB}$$

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.

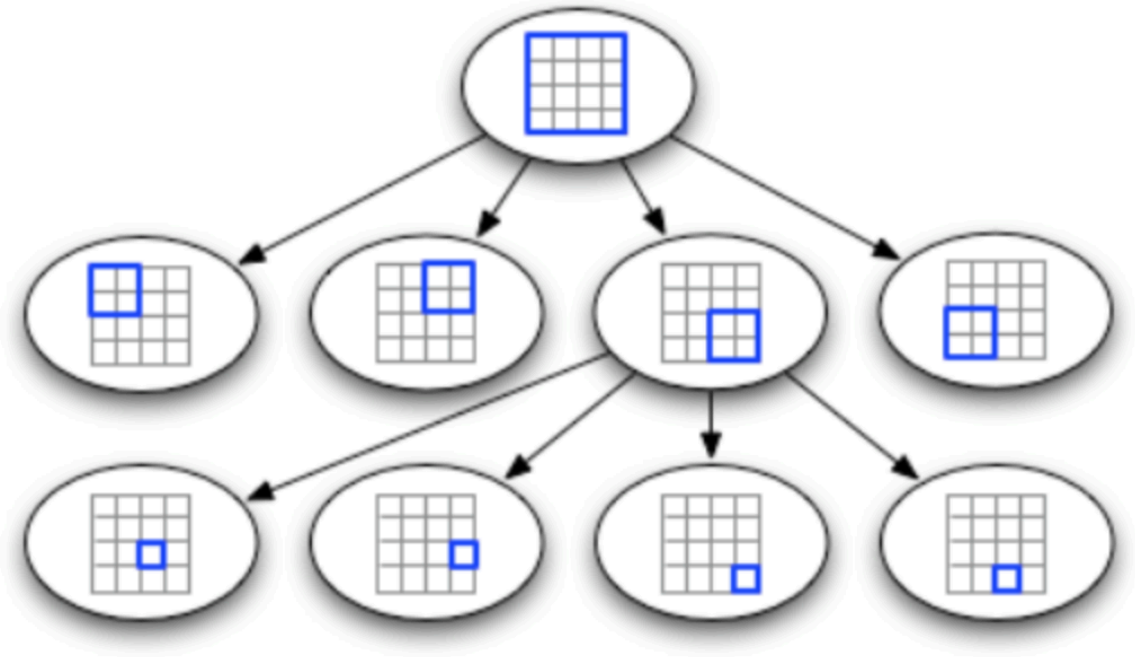
This problem can be solved if we can dynamically adjust our grid size, such that whenever we have a grid with a lot of places we break it down to create smaller grids. One challenge with this approach could be, how would we map these grids to locations? Also, how can we find all the neighboring grids 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



QuadTree

How will we build QuadTree? We will start with one node that would 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 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.

$$24 * 500M => 12 GB$$

Since each grid can have maximum 500 places and we have 500M locations, how many total grids we will have?

$$500M / 500 => 1M grids$$

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:

$$1M * 1/3 * 4 * 8 = 10 MB$$

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. While storing, we will find the region of each place to find the server and store the place there. Similarly, while querying for nearby places, we can ask the region server that contains user's location. This approach has a couple of issues:

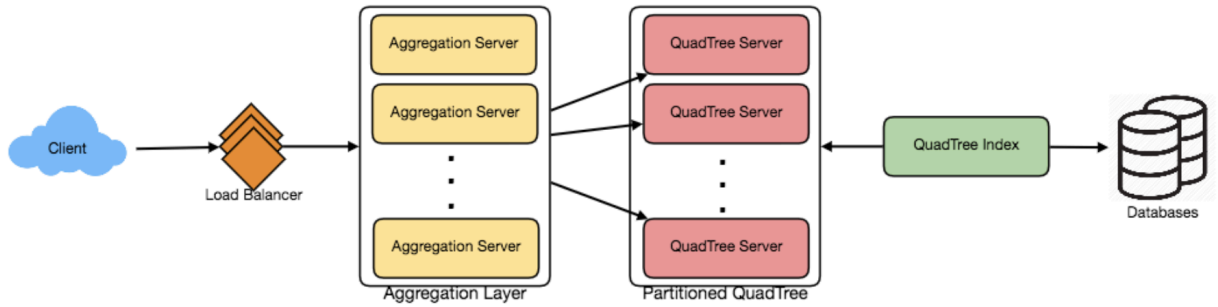
1. What if a region becomes hot? There would be a lot of queries on the server holding that region, making it perform slow. This will affect the performance of our service.
2. Over time some regions can end up storing a lot of places compared to others. Hence maintaining a uniform distribution of places, 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 nearby places of 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. Though, we do make sure that all servers have approximately 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.

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 and all write traffic will first go 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' would be the QuadTree server number and the 'value' would be a HashSet containing all the Places being kept on that QuadTree server. We need to store LocationID and Lat/Long with each place because through this information servers can build their QuadTrees. 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 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 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 top 100 places within a given radius, we can ask each partition of the QuadTree to return top 100 places having maximum popularity. Then the aggregator server

can determine 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 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 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.

Our next problem Designing Uber backend discusses dynamic updates of the QuadTree in detail.

Designing Uber backend

Let's design a ride-sharing service like Uber, which connects passengers who need a ride with drivers who have a car.

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 Uber app.

2. Requirements and Goals of the System

Let’s start with building a simpler version of Uber.

There are two kinds 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 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 customer 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 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 above-mentioned "Uber" use cases. The biggest difference we have is that our QuadTree was not built keeping in mind that there will 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 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, hence there will be 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 frequent? 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 total 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 1 million total drivers, we need the following memory (ignoring hash table overhead):

1 million * 35 bytes => 35 MB

How much bandwidth our service will 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 one million drivers, we will be getting 19MB per three seconds.

Do we need to distribute DriverLocationHT onto multiple servers? Although our memory and bandwidth requirements do not dictate that, since all this information can easily fit on one server, but for scalability, performance, and fault tolerance we should distribute the hash table onto multiple servers. We can distribute based on the DriverID to make the distribution completely random. Let's call the machines holding DriverLocationHT, Driver Location server. Other than storing 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 respective QuadTree server to refresh the driver's location. As discussed above, this can happen every 10 seconds.

How can we efficiently broadcast 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 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 driver's current position to the customer.

How much memory 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 the average let's assume that five customers subscribe 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.

$$(500K * 3) + (500K * 5 * 8) \sim= 21 MB$$

How much bandwidth we need to broadcast the driver's location to customers? For every active driver we have five subscribers, so 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:

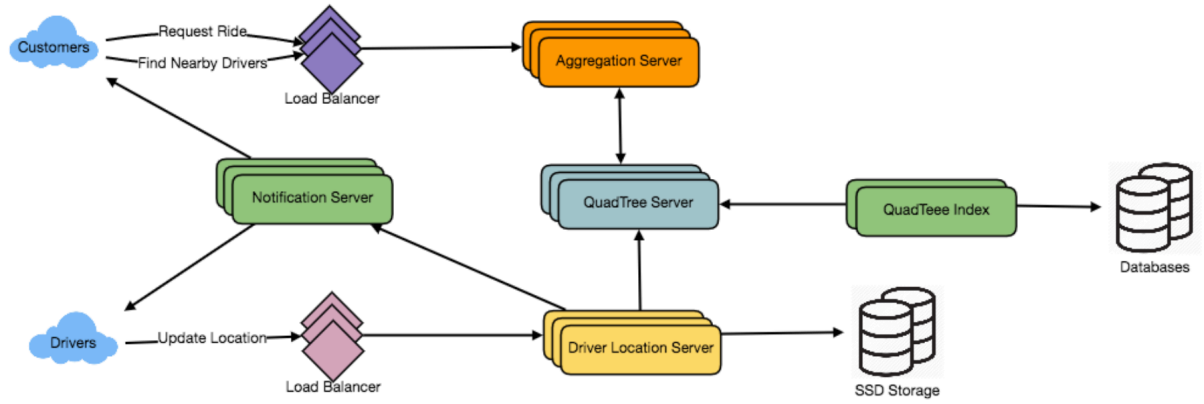
$$2.5M * 19 bytes => 47.5 MB/s$$

How can we efficiently implement Notification service? We can either use HTTP long polling or push notifications.

How the new publishers/drivers will get added for a current customer? As we have proposed above that 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, what 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 current positions of the drivers. Clients can query every five seconds to limit the number of round trips to the server. This solution looks quite 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 extra 10% before we partition/merge them. This should decrease the load for 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.
5. 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 a driver gets out of ten? While searching for top 10 drivers within a given radius, we can ask each partition of the QuadTree to return top 10 drivers with maximum rating. The aggregator server can then determine top 10 drivers among all the drivers returned by different partitions.

7. Advanced Issues

1. How to handle clients on slow and disconnecting networks?
2. What if a client gets disconnected when it was a part of a ride? How will we handle billing in such a scenario?
3. How about if clients pull all the information as compared to servers always pushing it?

Design BookMyShow

Let's design an online ticketing system that sells movie tickets like BookMyShow.

1. What is an online movie ticket booking system?

A movie ticket booking system provides its customers the ability to purchase theatre seats online. E-ticketing systems allow the customers to browse through movies currently being played and to book seats, anywhere and anytime.

2. Requirements and Goals of the System

Our ticket booking service should meet the following requirements:

Functional Requirements:

1. Our ticket booking service should be able to list down different cities where its affiliate cinemas are located.
2. Once the user selects the city, the service should display the movies released in that particular city.
3. Once the user selects the movie, the service should display the cinemas running that movie and its available shows.
4. The user should be able to select the show at a particular cinema and book their tickets.
5. 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 their preference.
6. The user should be able to distinguish available seats from the booked ones.
7. Users should be able to put a hold on the seats for five minutes before they make a payment to finalize the booking.
8. The user should be able to wait if there is a chance that seats might become available – e.g. when holds by other users expire.
9. Waiting customers should be serviced fairly in a first come first serve manner.

Non-Functional Requirements:

1. The system would need to be highly concurrent. There will be multiple booking requests for the same seat at any particular point in time. The service should handle this gracefully and fairly.
2. The core thing of the service is ticket booking which means financial transactions. This means that the system should be secure and the database ACID compliant.

3. Some Design Considerations

1. For simplicity, let’s assume our service doesn’t require user authentication.

2. The system will not handle partial ticket orders. Either user gets all the tickets they want, or they get nothing.
3. Fairness is mandatory for the system.
4. To stop system abuse, we can restrict users not to book more than ten seats.
5. We can assume that traffic would spike on popular/much-awaited movie releases, and the seats fill up pretty fast. The system should be scalable, highly available to cope up with the surge in traffic.

4. Capacity Estimation

Traffic estimates: Let's assume that our service has 3 billion page views per month and sells 10 million tickets a month.

Storage estimates: Let's assume that we have 500 cities and on average each city has ten cinemas. If there are 2000 seats in each cinema and on average, there are two shows every day.

Let's assume each seat booking needs 50 bytes (IDs, NumberOfSeats, ShowID, MovieID, SeatNumbers, SeatStatus, Timestamp, etc.) to store in the database. We would also need to store information about movies and cinemas, let's assume it'll take 50 bytes. So, to store all the data about all shows of all cinemas of all cities for a day:

500 cities * 10 cinemas * 2000 seats * 2 shows * (50+50) bytes = 2GB / day bytes = 1GB

To store five years of this data, we would need around 3.6PB.

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definitions of the APIs to search movie shows and reserve seats.

SearchMovies(api_dev_key, keyword, city, lat_long, radius, start_datetime, end_datetime, postal_code,
includeSpellcheck, results_per_page, sorting_order)

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.
keyword (string): Keyword to search on.
city (string): City to filter movies by.
lat_long (string): Latitude and longitude to filter by. **radius (number):** Radius of the area in which we want to search for events.
start_datetime (string): Filter movies with a start datetime after this datetime.
end_datetime (string): Filter movies with a start datetime before this datetime.
postal_code (string): Filter movies by postal code / zipcode.
includeSpellcheck (Enum: "yes" or " no"): Yes, to include spell check

suggestions in the response.
results_per_page (number): How many results to return per page. Maximum is 30.
sorting_order (string): Sorting order of the search result. Some allowable values : ‘name,asc’, ‘name,desc’, ‘date,asc’, ‘date,desc’, ‘distance,asc’, ‘name,date,asc’, ‘name,date,desc’, ‘date,name,asc’, ‘date,name,desc’.

Returns: (JSON)
Here is a sample list of movies and their shows:


```
[
  {
    "MovieID": 1,
    "ShowID": 1,
    "Title": "Cars 2",
    "Description": "About cars",
    "Duration": 120,
    "Genre": "Animation",
    "Language": "English",
    "ReleaseDate": "8th Oct. 2014",
    "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": 1,
    "ShowID": 2,
    "Title": "Cars 2",
    "Description": "About cars",
    "Duration": 120,
    "Genre": "Animation",
    "Language": "English",
    "ReleaseDate": "8th Oct. 2014",
    "Country": USA,
    "StartTime": "16:30",
    "EndTime": "18:30",
    "Seats":
```

```
  [
    {
      "Type": "Regular"
      "Price": 14.99
      "Status: "Full"
    },
    {
      "Type": "Premium"
     "Price": 24.99
     "Status: "Almost Full"
    }
  ]
 },
]
```

ReserveSeats(api_dev_key, session_id, movie_id, show_id, seats_to_reserve[])

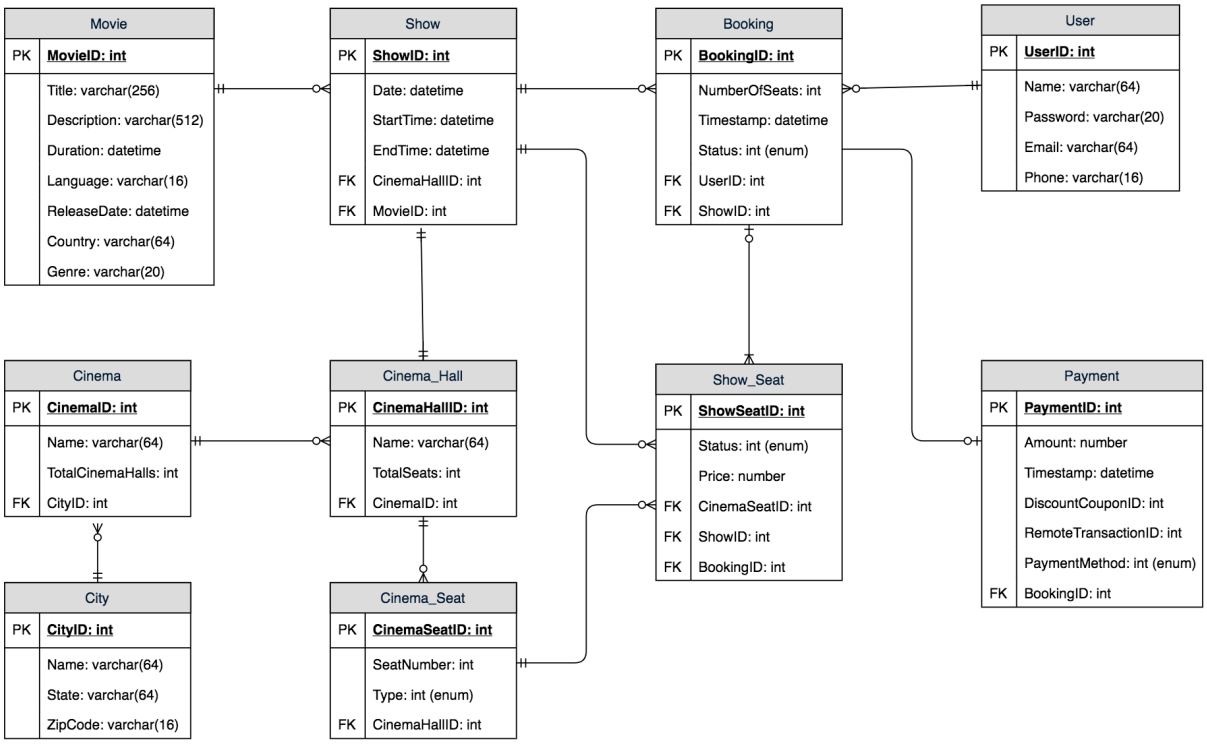
Parameters:
api_dev_key (string): same as above
session_id (string): User's session ID to track this reservation. Once the reservation time expires, user's reservation on the server will be removed using this ID.
movie_id (string): Movie to reserve.
show_id (string): Show to reserve.
seats_to_reserve (number): An array containing seat IDs to reserve.

Returns: (JSON)
Returns 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".

6. Database Design

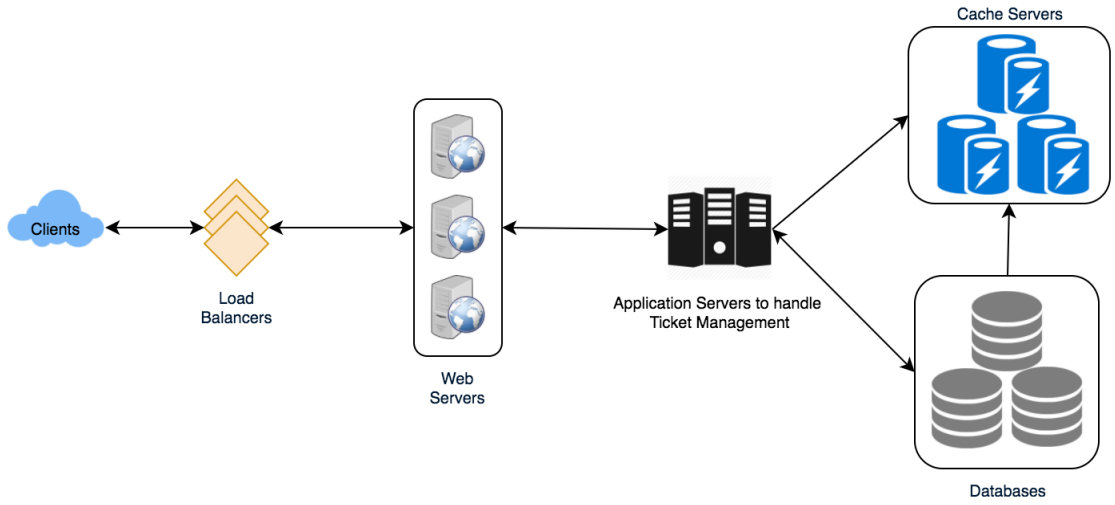
Here are a few observations about the data we are going to store:

1. Each City can have multiple Cinemas.
2. Each Cinema will have multiple halls.
3. Each Movie will have many Shows, and each Show will have multiple Bookings.
4. A user can have multiple bookings.



7. High Level Design

At a high level, our web servers will manage Users' sessions, and application servers will handle all the ticket management, storing data in the databases and work with cache servers to process reservations.



8. Detailed Component Design

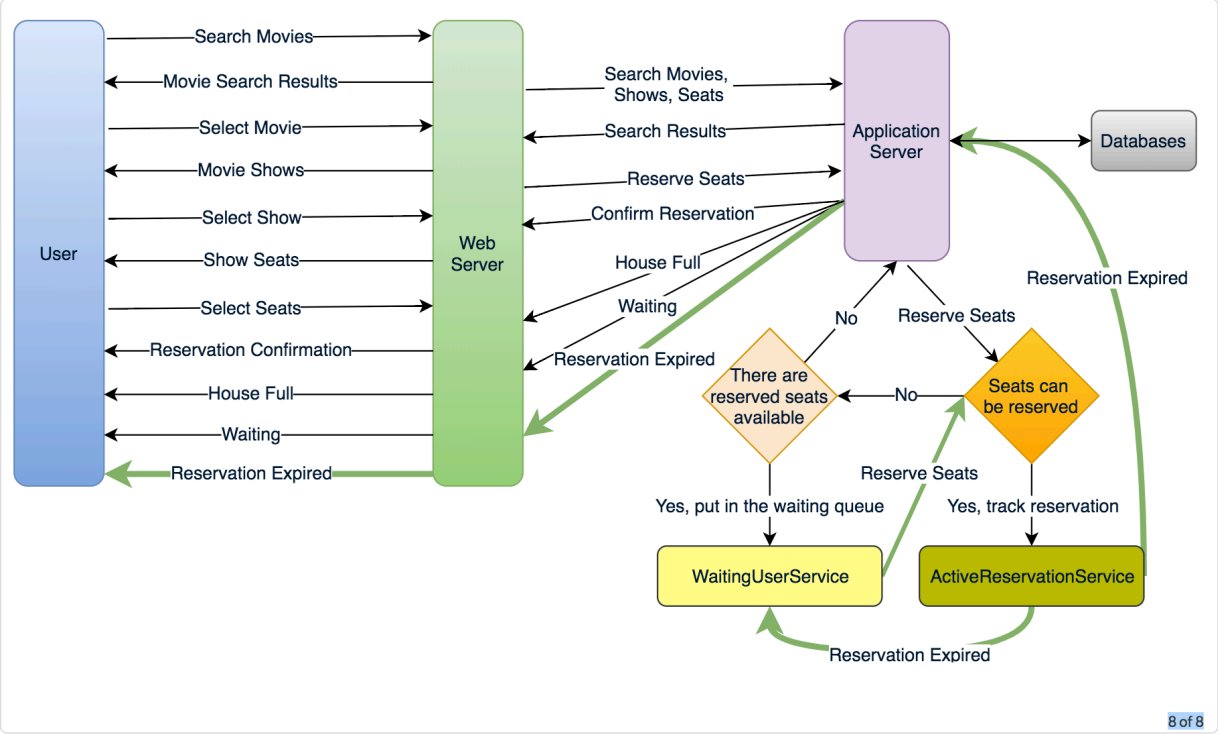
First, let's try to build our service assuming if it is being served from a single server.

Ticket Booking Workflow: Following would be a typical ticket booking workflow:

1. User searches for a movie.
2. User selects a movie.
3. User is shown the available shows of the movie.
4. User selects a show.
5. User selects the number of seats to be reserved.
6. If the required number of seats are available, the user is shown a map of the theater to select seats. If not, the user is taken to 'step 8' below.
7. Once the user selects the seat, the system will try to reserve those selected seats.
8. If seats can't be reserved, we have following options:

- Show is full; the user is shown the error message.
- The seats user wants to reserve are no longer available, but there are other seats available, so the user is taken back to the theater map to choose different seats.
- There are no seats available to reserve, but all the seats are not booked yet as there are some seats that other users are holding in reservation pool and have not booked yet. The user will be taken to a waiting page where they can wait until required seats get freed from the reservation pool. This waiting could result in following options:
    - If the required number of seats become available, the user is taken to the theater map page where they can choose the seats.
    - While waiting if all seats get booked, or there are fewer seats in the reservation pool than the user intend to book, the user is shown the error message.
    - User cancels the waiting and is taken back to the movie search page.
    - At maximum, a user can wait one hour, after that user's session gets expired and the user is taken back to the movie search page.

9. If seats are reserved successfully, the user has five minutes to pay for the reservation. After payment, booking is marked complete. If the user is not able to pay within five minutes, all their reserved seats are freed to become available to other users.



How would the server keep track of all the active reservation that haven't been booked yet? And how would the server keep track of all the waiting customers?

We need two daemon services, one to keep track of all active reservations and to remove any expired reservation from the system, let's call it **ActiveReservationService**. The other service would be keeping track of all the waiting user requests, and as soon as the required number of seats become available, it'll notify the (the longest waiting) user to choose the seats, let's call it **WaitingUserService**.

a. ActiveReservationsService

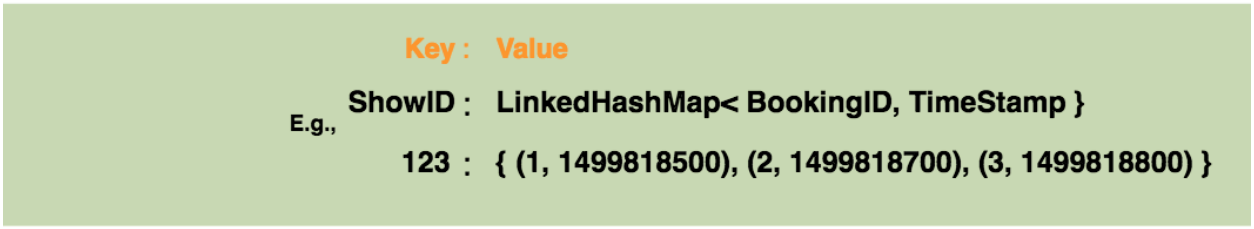
We can keep all the reservations of a 'show' in memory in a Linked HashMap in addition to keeping all the data in the database. We would need a Linked HashMap so that we can jump to any reservation to remove it when the booking is complete. Also, since we will have expiry time associated with each reservation, the head of the linked HashTable will always point to the oldest reservation record, so that 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' would be 'ShowID' and the 'value' would be the Linked HashMap containing 'BookingID' and creation 'Timestamp'.

In the database, we'll store the reservation in the 'Booking' table, and the 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, the system will update the

'Status' to 'Booked (2)' and remove the reservation record from the Linked HashMap of the relevant show. When the reservation is expired, we can either remove it from the Booking table or mark it 'Expired (3)' in addition to removing it from memory.

ActiveReservationsService will also work with the external financial service to process user payments. Whenever a booking is completed, or a reservation gets expired, WaitingUsersService will get a signal, so that any waiting customer can be served.



ActiveReservationsService keeping track of all active reservations

b. WaitingUsersService

Just like ActiveReservationsService, we can keep all the waiting users of a show in memory in a Linked HashMap. We need a Linked HashMap so that we can jump to any user to remove them from the HashMap when the user cancels their request. Also, since we are serving in a first-come-first-serve manner, the head of the Linked HashMap would always be pointing to the longest waiting user, so that whenever seats become available, we can serve users in a fair manner.

We will have a HashTable to store all the waiting users for every Show. The 'key' would be 'ShowID, and the 'value' would be a Linked HashMap containing 'UserIDs' and their wait-start-time.

Clients can use Long Polling for keeping themselve updated for their reservation status. Whenever seats become available, the server can use this request to notify the user.

Reservation Expiration

On the server, ActiveReservationsService keeps track of expiry (based on reservation time) of active reservations. As the client will be shown a timer (for the expiration time) which could be a little out of sync with the server, we can add a buffer of five seconds on the server to safeguard from a broken experience - such that – the client never times out after the server, preventing a successful purchase.

9. Concurrency

How to handle concurrency; such that no two users are able to book same seat? We can use transactions in SQL databases to avoid any clashes. For example, if we are using SQL server we can utilize Transaction Isolation Levels to lock the rows before we can update them. Here is the sample code:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

    -- Suppose we intend to reserve three seats (IDs: 54, 55, 56) for ShowID=99
    Select * From Show_Seat where ShowID=99 && ShowSeatID in (54, 55, 56) && Status=0 -- free

    -- if the number of rows returned by the above statement is three, we can update to
    -- return success otherwise return failure to the user.
    update Show_Seat ...
    update Booking ...

COMMIT TRANSACTION;
```

'Serializable' is the highest isolation level and guarantees safety from Dirty, Nonrepeatable and Phantoms reads. One thing to note here, within a transaction if we read rows we get a write lock on them so that they can't be updated by anyone else.

Once the above database transaction is successful, we can start tracking the reservation in ActiveReservationService.

10. Fault Tolerance

What happens when ActiveReservationsService or WaitingUsersService crashes?
Whenever ActiveReservationsService crashes, we can read all the active reservations from the 'Booking' table. Remember that we keep the 'Status' column as 'Reserved (1)' until a reservation gets booked. Another option is to have master-slave configuration so that when master crashes slave can take over. Since we are not storing the waiting users in the database, so when WaitingUsersService crashes, we don't have any means to recover that data unless we have a master-slave setup.

Similarly, we'll have a master-slave setup for databases to make them fault tolerant.

11. Data Partitioning

Database partitioning: If we partition by 'MovieID' then all the Shows of a movie will be on a single server. For a very 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.

ActiveReservationService and WaitingUserService partitioning: Our web servers will manage all the active users' sessions and handle all the communication with the users. We can use Consistent Hashing to allocate application servers for both ActiveReservationService and WaitingUserService based upon the 'ShowID'. This way, all reservations and waiting users of a particular show will be handled by a certain set of servers. Let's assume for load balancing our Consistent Hashing allocates three servers for any Show, so whenever a reservation is expired, the server holding that reservation will do following things:

1. Update database to remove the Booking (or mark it expired) and update the seats' Status in 'Show_Seats' table.
2. Remove the reservation from the Linked HashMap.
3. Notify the user that their reservation has expired.
4. Broadcast a message to all WaitingUserService servers that are holding waiting users of that Show to figure out the longest waiting user. Consistent Hashing scheme will tell what servers are holding these users.
5. Send a message to the WaitingUserService server holding the longest waiting user to process their request if required seats have become available.

Whenever a reservation is successful, following things will happen:

1. The server holding that reservation sends a message to all servers holding waiting users of that Show so that they can expire all those waiting users that need more seats than the available seats.
2. Upon receiving above message, all servers holding the waiting users will query the database to find how many free seats are available now. Database cache would greatly help here to run this query only once.
3. Expire all waiting users who want to reserve more seats than the available seats. For this, WaitingUserService has to iterate through the Linked HashMap of all the waiting users.