# System Design Interviews: A step by step guide

**We'll cover the following**

* + [Step 1: Requirements clarifications](https://www.educative.io/courses/grokking-the-system-design-interview/B8nMkqBWONo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px-step-1-requirements-clarificationsdiv)
  + [Step 2: Back-of-the-envelope estimation](https://www.educative.io/courses/grokking-the-system-design-interview/B8nMkqBWONo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px-step-2-back-of-the-envelope-estimationdiv)
  + [Step 3: System interface definition](https://www.educative.io/courses/grokking-the-system-design-interview/B8nMkqBWONo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px-step-3-system-interface-definitiondiv)
  + [Step 4: Defining data model](https://www.educative.io/courses/grokking-the-system-design-interview/B8nMkqBWONo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px-step-4-defining-data-modeldiv)
  + [Step 5: High-level design](https://www.educative.io/courses/grokking-the-system-design-interview/B8nMkqBWONo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px-step-5-high-level-designdiv)
  + [Step 6: Detailed design](https://www.educative.io/courses/grokking-the-system-design-interview/B8nMkqBWONo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px-step-6-detailed-designdiv)
  + [Step 7: Identifying and resolving bottlenecks](https://www.educative.io/courses/grokking-the-system-design-interview/B8nMkqBWONo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px-step-7-identifying-and-resolving-bottlenecksdiv)
  + [Summary](https://www.educative.io/courses/grokking-the-system-design-interview/B8nMkqBWONo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5pxsummarydiv)

A lot of software engineers struggle with system design interviews (SDIs) primarily because of three reasons:

* The unstructured nature of SDIs, where the candidates are asked to work on an open-ended design problem that doesn’t have a standard answer.
* Candidates lack experience in developing complex and large-scale systems.
* Candidates did not spend enough time to prepare for SDIs.

Like coding interviews, candidates who haven’t put a deliberate effort to prepare for SDIs, mostly perform poorly, especially at top companies like Google, Facebook, Amazon, Microsoft, etc. In these companies, candidates who do not perform above average 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 shows the candidate’s ability to handle a complex system.

In this course, we’ll follow a step by step approach to solve multiple design problems. First, let’s go through these steps:

## Step 1: Requirements clarifications

It is always a good idea to ask questions about the exact scope of the problem we are trying to solve. 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 define the end goals of the system always have a better chance to be successful in the interview. Also, since we only have 35-40 minutes to design a (supposedly) large system, we should clarify what parts of the system we will be focusing on.

Let’s expand this with an actual example of designing a Twitter-like service. Here are some questions for designing Twitter that should be answered before moving on to the next steps:

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

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

## Step 2: Back-of-the-envelope estimation

It is always a good idea to estimate the scale of the system we’re going to design. This will also help later when we focus on scaling, partitioning, load balancing, and caching.

* What scale is expected from the system (e.g., number of new tweets, number of tweet views, number of timeline generations per sec., etc.)?
* How much storage will we need? We will have different storage requirements if users can have photos and videos in their tweets.
* What network bandwidth usage are we expecting? This will be crucial in deciding how we will manage traffic and balance load between servers.

## Step 3: System interface definition

Define what APIs are expected from the system. This will establish the exact contract expected from the system and ensure if we haven’t gotten any requirements wrong. Some examples of APIs for our Twitter-like service will 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 4: Defining data model

Defining the data model in the early part of the interview will clarify how data will flow between different system components. Later, it will guide for data partitioning and management. The candidate should identify various system entities, how they will interact with each other, and different aspects of data management like storage, transportation, encryption, etc. Here are some entities for our Twitter-like service:

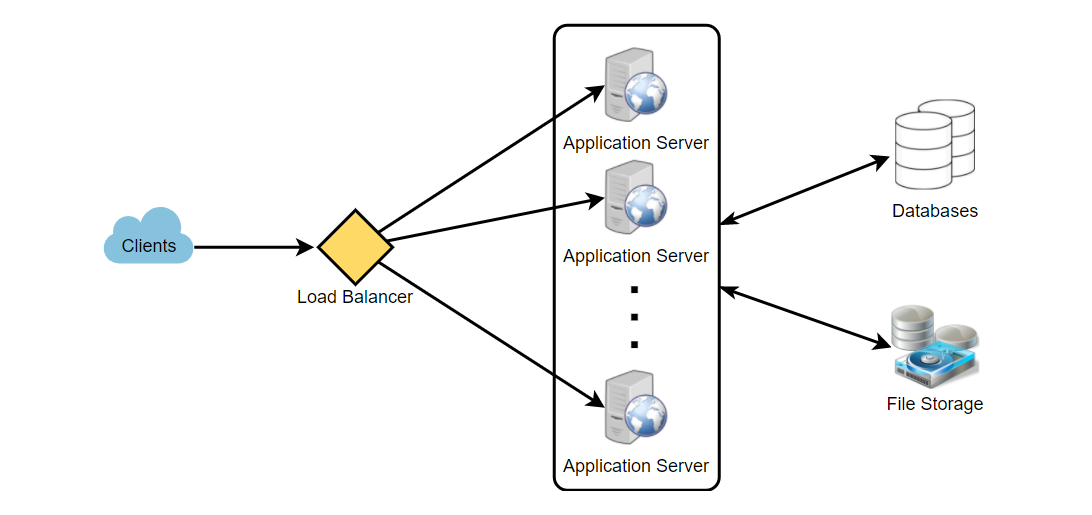
**User:** UserID, Name, Email, DoB, CreationDate, LastLogin, etc.  
**Tweet:** TweetID, Content, TweetLocation, NumberOfLikes, TimeStamp, etc.  
**UserFollow:** UserID1, UserID2  
**FavoriteTweets:** UserID, TweetID, TimeStamp

Which database system should we use? Will NoSQL like [Cassandra](https://en.wikipedia.org/wiki/Apache_Cassandra) best fit our needs, or should we use a 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 the core components of our system. We should identify enough components that are needed to solve the actual problem from end to end.

For Twitter, at a high level, we will 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 will have a lot more read traffic (compared to write), we can decide to have separate servers to handle these scenarios. On the back-end, we need an efficient database that can store all the tweets and support a large number of reads. We will also need a distributed file storage system for storing photos and videos.



## Step 6: Detailed design

Dig deeper into two or three major components; the interviewer’s feedback should always guide us to what parts of the system need further discussion. We should present different approaches, their pros and cons, and explain why we will prefer one approach over the other. Remember, there is no single answer; the only important thing is to consider tradeoffs between different options while keeping system constraints in mind.

* Since we will be storing a massive 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 could it cause?
* How will we handle hot users who tweet a lot or follow lots of people?
* Since users’ timeline will contain the most recent (and relevant) tweets, should we try to store our data so that it is optimized for scanning the 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 have enough replicas of the data so that we can still serve our users if we lose a few servers?
* Similarly, do we have enough copies of different services running such that a few failures will not cause a total system shutdown?
* How are we monitoring the performance of our service? Do we get alerts whenever critical components fail or their performance degrades?

## Summary

In short, preparation and being organized during the interview are the keys to success in system design interviews. The steps mentioned above should guide you to remain on track and cover all the different aspects while designing a system.

Let’s apply the above guidelines to design a few systems that are asked in SDIs.

# Designing a URL Shortening service like TinyURL

Let's design a URL shortening service like TinyURL. This service will provide short aliases redirecting to long URLs. Similar services: bit.ly, ow.ly, short.io Difficulty Level: Easy

**We'll cover the following**

* + [1. Why do we need URL shortening?](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px1-why-do-we-need-url-shorteningdiv)
  + [2. Requirements and Goals of the System](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px2-requirements-and-goals-of-the-systemdiv)
  + [3. Capacity Estimation and Constraints](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px3-capacity-estimation-and-constraintsdiv)
  + [4. System APIs](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px4-system-apisdiv)
  + [5. Database Design](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px5-database-designdiv)
    - [Database Schema:](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#database-schema)
  + [6. Basic System Design and Algorithm](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px6-basic-system-design-and-algorithmdiv)
  + [a. Encoding actual URL](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#a-encoding-actual-url)
  + [b. Generating keys offline](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#b-generating-keys-offline)
  + [7. Data Partitioning and Replication](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px7-data-partitioning-and-replicationdiv)
  + [8. Cache](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px8-cachediv)
  + [9. Load Balancer (LB)](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px9-load-balancer-lbdiv)
  + [10. Purging or DB cleanup](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px10-purging-or-db-cleanupdiv)
  + [11. Telemetry](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px11-telemetrydiv)
  + [12. Security and Permissions](https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px12-security-and-permissionsdiv)

## 1. Why do we need URL shortening?

URL shortening is used to create shorter aliases for long URLs. We call these shortened aliases “short links.” Users are redirected to the original URL when they hit these short links. Short links save a lot of space when displayed, printed, messaged, or tweeted. Additionally, users are less likely to mistype shorter URLs.

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 one-third the size of the actual URL.

URL shortening is used to optimize links across devices, track individual links to analyze audience, measure ad campaigns’ performance, or hide affiliated original URLs.

If you haven’t used [tinyurl.com](http://tinyurl.com/) before, please try creating a new shortened URL and spend some time going through the various options their service offers. This will help you a lot in understanding this chapter.

## 2. Requirements and Goals of the System

💡     ***You should always clarify requirements at the beginning of the interview. Be sure to ask questions to find the exact scope of the system that the interviewer has in mind.***

Our URL shortening system should meet the following requirements:

**Functional Requirements:**

1. Given a URL, our service should generate a shorter and unique alias of it. This is called a short link. This link should be short enough to be easily copied and pasted into applications.
2. When users access a short link, our service should redirect them to the original link.
3. Users should optionally be able to pick a custom short link for their URL.
4. Links will expire after a standard default timespan. Users should be able to specify the 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 minimal 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 will be read-heavy. There will be lots of redirection requests compared to new URL shortenings. Let’s assume a 100:1 ratio between read and write.

**Traffic estimates:** Assuming, we will have 500M new URL shortenings per month, with 100:1 read/write ratio, we can expect 50B redirections during the same period:

100 \* 500M => 50B

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

Considering 100:1 read/write ratio, URLs redirections per second will be:

100 \* 200 URLs/s = 20K/s

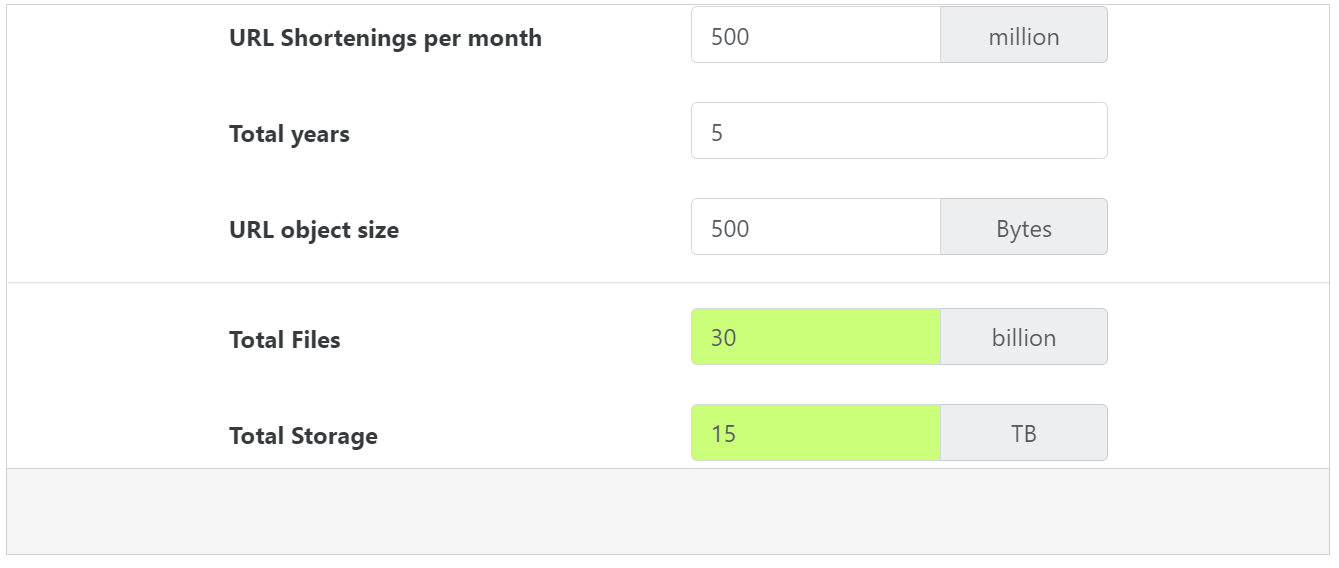
**Storage estimates:** Let’s assume we store every URL shortening request (and associated shortened link) for 5 years. Since we expect to have 500M new URLs every month, the total number of objects we expect to store will be 30 billion:

500 million \* 5 years \* 12 months = 30 billion

Let’s assume that each stored object will be approximately 500 bytes (just a ballpark estimate–we will dig into it later). We will need 15TB of total storage:

30 billion \* 500 bytes = 15 TB

In the following table, we can change our assumptions to see how the estimates change:



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

200 \* 500 bytes = 100 KB/s

For read requests, since every second we expect ~20K URLs redirections, total outgoing data for our service would be 10MB per second:

20K \* 500 bytes = ~10 MB/s

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

Since we have 20K requests per second, we will be getting 1.7 billion requests per day:

20K \* 3600 seconds \* 24 hours = ~1.7 billion

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

0.2 \* 1.7 billion \* 500 bytes = ~170GB

One thing to note here is that since there will be many duplicate requests (of the same URL), our actual memory usage will be less than 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 | 20K/s |
| Incoming data | 100KB/s |
| Outgoing data | 10MB/s |
| Storage for 5 years | 15TB |
| Memory for cache | 170GB |

## 4. System APIs

💡     ***Once we've finalized the requirements, it's always a good idea to define the system APIs. This should explicitly state what is expected from the system.***

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:

createURL(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 the encoding.  
expire\_date (string): Optional expiration date for the shortened URL.

**Returns:** (string)  
A successful insertion returns the shortened URL; otherwise, it 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?** A malicious user can put us out of business by consuming all URL keys in the current design. To prevent abuse, we can limit users via their api\_dev\_key. Each api\_dev\_key can be limited to a certain number of URL creations and redirections per some time period (which may be set to a different duration per developer key).

## 5. Database Design

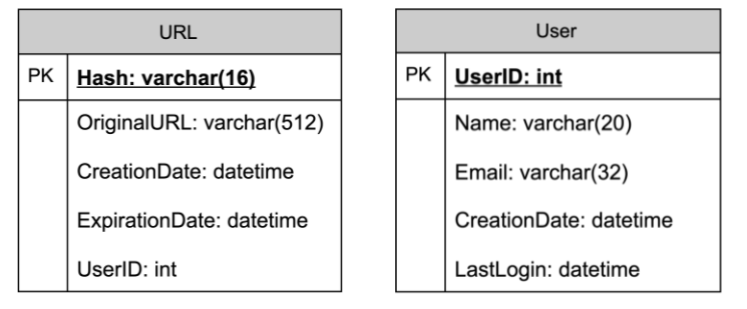
💡     ***Defining the DB schema in the early stages of the interview would help to understand the data flow among various components and later would guide towards data partitioning.***

A few observations about the nature of the data we will store:

1. We need to store billions of records.
2. Each object we store is small (less than 1K).
3. There are no relationships between records—other than storing which user created a URL.
4. Our service is read-heavy.

### Database Schema:

We would need two tables: one for storing information about the URL mappings and one for the user’s data who created the short link.



**What kind of database should we use?** Since we anticipate storing billions of rows, and we don’t need to use relationships between objects – a NoSQL store like [DynamoDB](https://en.wikipedia.org/wiki/Amazon_DynamoDB), [Cassandra](https://en.wikipedia.org/wiki/Apache_Cassandra) or [Riak](https://en.wikipedia.org/wiki/Riak" \t "_blank) is a better choice. A NoSQL choice would also be easier to scale. Please see [SQL vs NoSQL](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5728116278296576/) for more details.

## 6. Basic System Design and Algorithm

The problem we are solving here is how to generate a short and unique key for a given URL.

In the TinyURL example in Section 1, the shortened URL is “[http://tinyurl.com/jlg8zpc”](http://tinyurl.com/jlg8zpc%E2%80%9D). The last seven 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](https://en.wikipedia.org/wiki/MD5) or [SHA256](https://en.wikipedia.org/wiki/SHA-2), etc.) of the given URL. The hash can then be encoded for display. 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](https://en.wikipedia.org/wiki/Base64#Base64_table) 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 letters long key would result in 64^6 = ~68.7 billion possible strings.  
Using base64 encoding, an 8 letters long key would result in 64^8 = ~281 trillion possible strings.

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

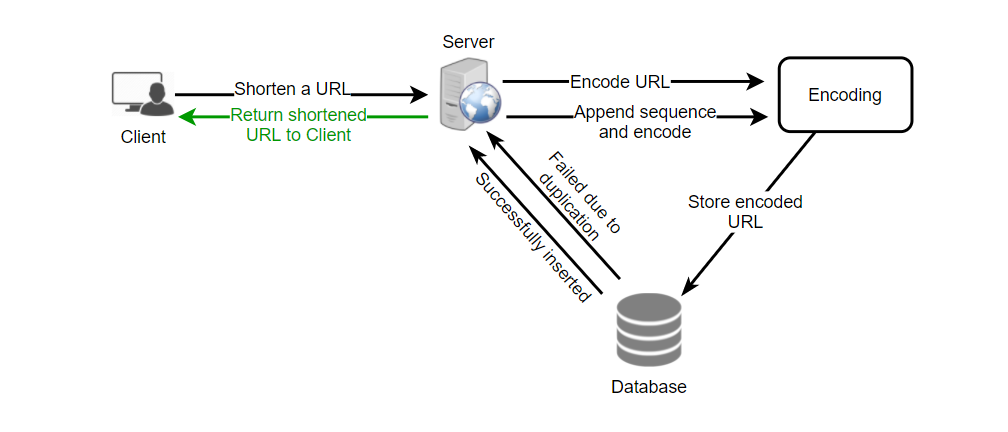
If we use the MD5 algorithm as our hash function, it will produce a 128-bit hash value. After base64 encoding, we’ll get a string having more than 21 characters (since each base64 character encodes 6 bits of the hash value). Now we only have space for 6 (or 8) characters per short key; how will we choose our key then? We can take the first 6 (or 8) letters for the key. This could result in key duplication; to resolve that, we can choose some other characters out of the encoding string or swap some characters.

**What are the 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 its hash. We don’t need to store this sequence number in the databases, though. Possible problems with this approach could be an ever-increasing sequence number. Can it overflow? Appending an increasing sequence number will also impact the performance of the service.

Another solution could be to append the user id (which should be unique) to the input URL. However, if the user has not signed in, we would have to 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

**1** of 9

## b. 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 take one of the already-generated keys and use it. This approach will make things quite simple and fast. Not only are we not encoding the URL, but we won’t have to worry about duplications or collisions. KGS will make sure all the keys inserted into key-DB are unique

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

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

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

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

**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:

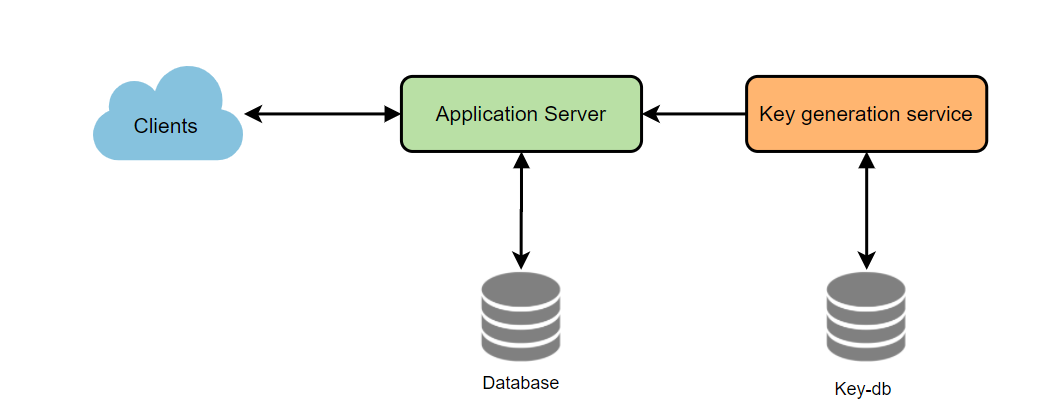
6 (characters per key) \* 68.7B (unique keys) = 412 GB.

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

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

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

**Should we impose size limits on custom aliases?** Our service supports custom aliases. Users can pick any ‘key’ they like, but providing a custom alias is not mandatory. However, it is reasonable (and often desirable) to impose a size limit on a custom alias to ensure we have a consistent URL database. Let’s assume users can specify a maximum of 16 characters per customer key (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 URLs. Therefore, we need to develop a partitioning scheme that would divide and store our data into different DB servers.

**a. Range Based Partitioning:** We can store URLs in separate partitions based on the hash key’s first letter. Hence we save all the URLs starting with the letter ‘A’ (and ‘a’) in one partition, save those that start with the letter ‘B’ in 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. Thus, we should develop a static partitioning scheme to always store/find a URL in a predictable manner.

The main problem with this approach is that it can lead to unbalanced DB servers. For example, we decide to put all URLs starting with the letter ‘E’ into a DB partition, but later we realize that we have too many URLs that start with the letter ‘E.’

**b. Hash-Based Partitioning:** In this scheme, we take a hash of the object we are storing. We then calculate which partition to use based upon the hash. In our case, we can take the hash of the ‘key’ or the short link to determine the partition in which we store the data object.

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]). This number would represent the partition in which we store our object.

This approach can still lead to overloaded partitions, which can be solved using [Consistent Hashing](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5709068098338816/).

## 8. Cache

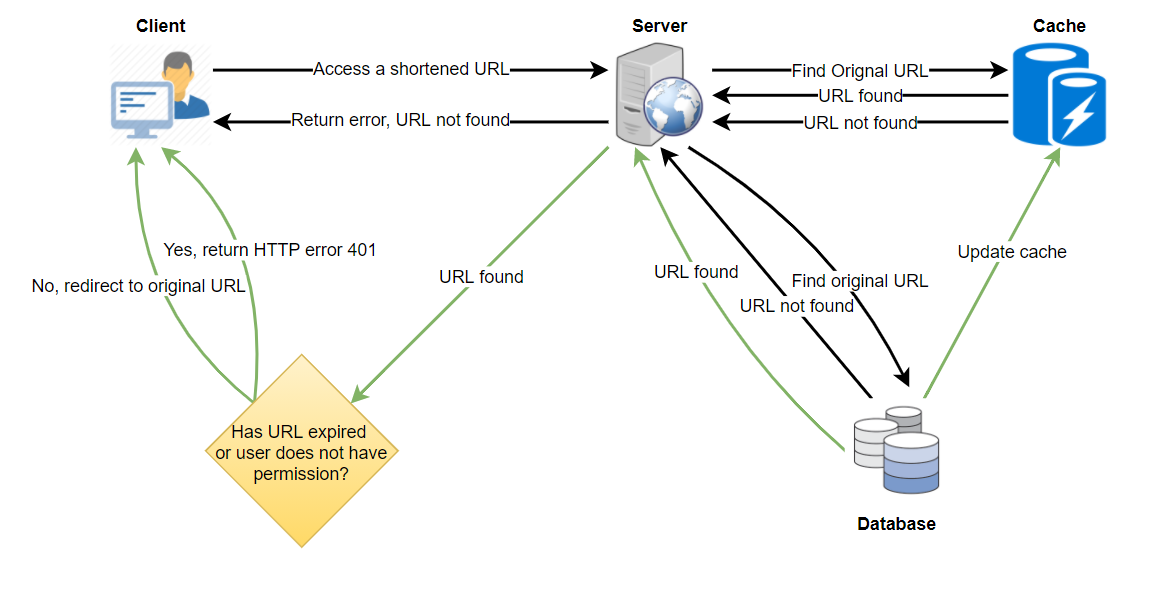
We can cache URLs that are frequently accessed. We can use any off-the-shelf solution like [Memcached](https://en.wikipedia.org/wiki/Memcached), which can store full URLs with their respective hashes. Thus, the application servers, before hitting the backend storage, can quickly check if the cache has the desired URL.

**How much cache memory should we have?** We can start with 20% of daily traffic and, based on clients’ usage patterns, we can adjust how many cache servers we need. As estimated above, we need 170GB of memory to cache 20% of daily traffic. Since a modern-day server can have 256GB of memory, we can easily fit all the cache into one machine. Alternatively, we can use a couple of smaller servers to store all these hot URLs.

**Which cache eviction policy would best fit our needs?** When the cache is full, and we want to replace a link with a newer/hotter URL, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system. Under this policy, we discard the least recently used URL first. We can use a [Linked Hash Map](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html) or a similar data structure to store our URLs and Hashes, which will also keep track of the URLs that have been accessed recently.

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

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



Request flow for accessing a shortened URL

**1** of 11

## 9. Load Balancer (LB)

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

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

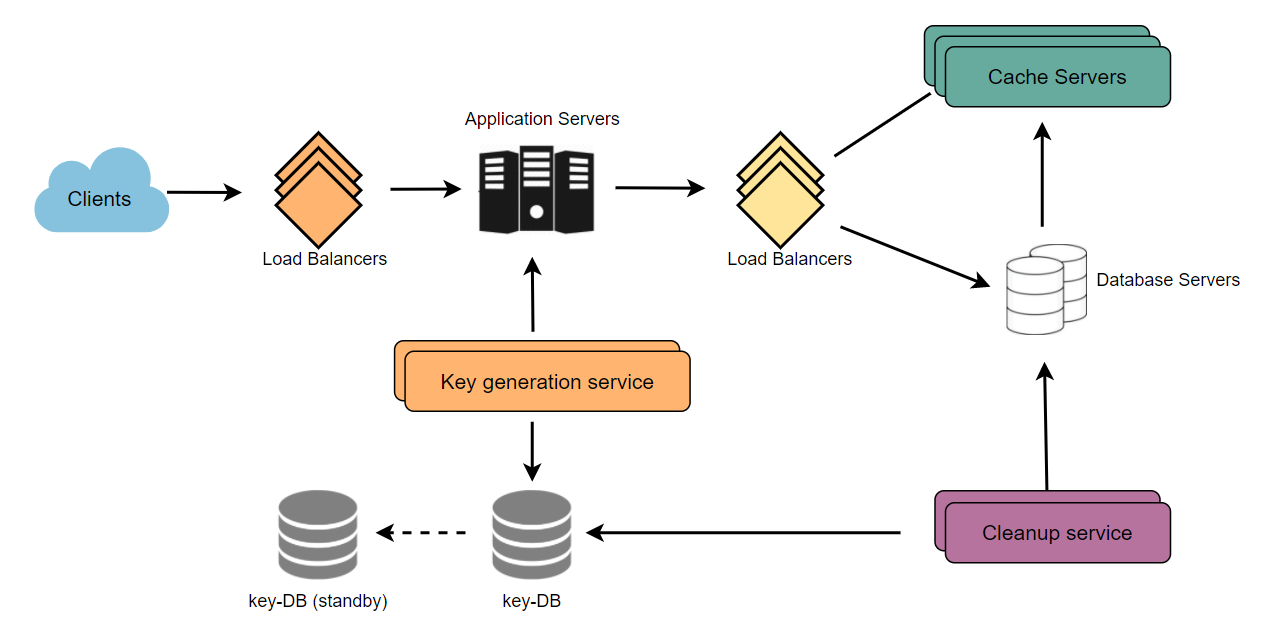
Initially, we could use a simple Round Robin approach that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is that if a server is dead, LB will take it out of the rotation and stop sending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

A problem with Round Robin LB is that we do not consider the server load. As a result, if a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load and adjusts traffic based on that.

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

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

Detailed component design for URL shortening

## 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?

Some statistics worth tracking: country of the visitor, date and time of access, web page that referred the click, browser, or platform from where the page was accessed.

## 12. Security and Permissions

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

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

# Designing Pastebin

Let's design a Pastebin like web service, where users can store plain text. Users of the service will enter a piece of text and get a randomly generated URL to access it. Similar Services: pastebin.com, controlc.com, hastebin.com, privatebin.net Difficulty Level: Easy

**We'll cover the following**

* + [1. What is Pastebin?](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px1-what-is-pastebindiv)
  + [2. Requirements and Goals of the System](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px2-requirements-and-goals-of-the-system-div)
  + [3. Some Design Considerations](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px3-some-design-considerationsdiv)
  + [4. Capacity Estimation and Constraints](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px4-capacity-estimation-and-constraintsdiv)
  + [5. System APIs](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px5-system-apisdiv)
  + [6. Database Design](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px6-database-designdiv)
    - [Database Schema:](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#database-schema)
  + [7. High Level Design](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px7-high-level-designdiv)
  + [8. Component Design](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px8-component-designdiv)
  + [a. Application layer](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#a-application-layer)
  + [b. Datastore layer](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#b-datastore-layer)
  + [9. Purging or DB Cleanup](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px9-purging-or-db-cleanupdiv)
  + [10. Data Partitioning and Replication](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px10-data-partitioning-and-replicationdiv)
  + [11. Cache and Load Balancer](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px11-cache-and-load-balancerdiv)
  + [12. Security and Permissions](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px12-security-and-permissionsdiv)

## 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](http://pastebin.com/) before, please try creating a new ‘Paste’ there and spend some time going through the different options their service offers. This will help you a lot in understanding this chapter.

## 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 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](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904), 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 will be read-heavy; there will be more read requests compared to new Paste creation. We can assume a 5:1 ratio between the 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 hours \* 3600 seconds) ~= 12 pastes/sec

Paste reads per second:

5M / (24 hours \* 3600 seconds) ~= 58 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 GB/day

If we want to store this data for ten years we would need a 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 to 51.4TB.

**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 the 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 the 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, it will return 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’.

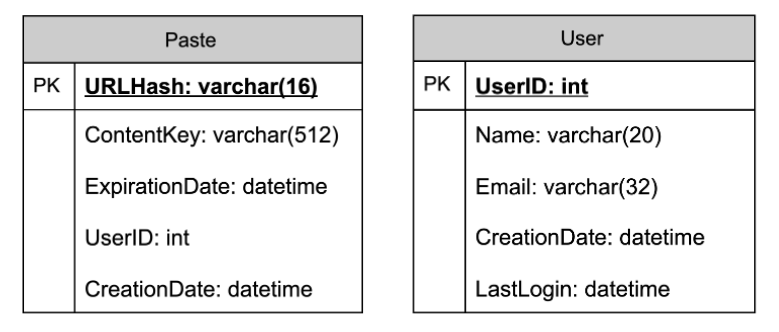
## 6. Database Design

A few observations about the nature of the data we are storing:

1. We need to store billions of records.
2. Each metadata object we are storing would be small (less than 1KB).
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: [**#**](https://www.educative.io/courses/grokking-the-system-design-interview/3jyvQ3pg6KO#database-schema)

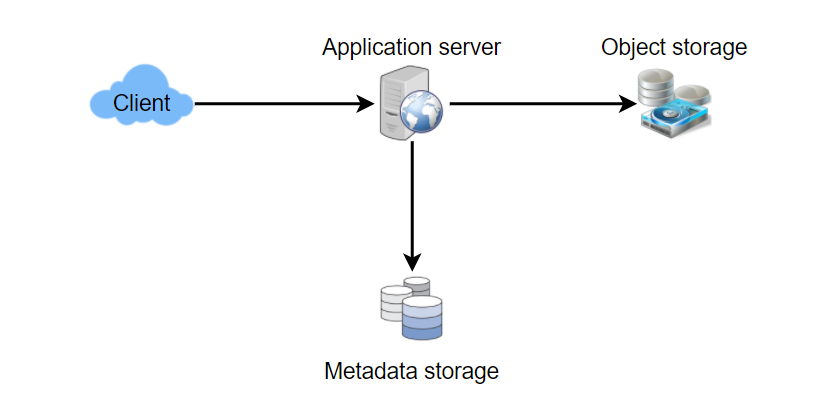
We would need two tables, one for storing information about the Pastes and the other for users’ data.



Here, ‘URlHash’ is the URL equivalent of the TinyURL, and ‘ContentKey’ is a reference to an external object storing the contents of the paste; we’ll discuss the external storage of the paste contents later in the chapter.

## 7. 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 the paste contents in some object storage (like [Amazon S3](https://en.wikipedia.org/wiki/Amazon_S3)). This division of data will also allow us to scale them individually.



## 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 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 for 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 gives some keys to an application server, it can move these to the used keys table. KGS can always keep some keys in memory so that whenever a server needs them, it can quickly provide them. As soon as KGS loads some keys in memory, it can move them to the used keys table; this way we can make sure each server gets unique keys. If KGS dies before using all the keys loaded in memory, we will be wasting those keys. We can ignore these keys given that we have a huge number of them.

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

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

**How does it 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, it 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. Object storage: We can store our contents in an Object Storage like Amazon’s S3. Whenever we feel like hitting our full capacity on content storage, we can easily increase it by adding more servers.

## 9. Purging or DB Cleanup

Please see [Designing a URL Shortening service](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904).

## 10. Data Partitioning and Replication

Please see [Designing a URL Shortening service](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904).

## 11. Cache and Load Balancer

Please see [Designing a URL Shortening service](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904).

## 12. Security and Permissions

Please see [Designing a URL Shortening service](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904).

# Designing Instagram

Let's design a photo-sharing service like Instagram, where users can upload photos to share them with other users. Similar Services: Flickr, Picasa Difficulty Level: Medium

**We'll cover the following**

* + [1. What is Instagram?](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px-1-what-is-instagramdiv)
  + [2. Requirements and Goals of the System](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px2-requirements-and-goals-of-the-systemdiv)
  + [3. Some Design Considerations](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px3-some-design-considerationsdiv)
  + [4. Capacity Estimation and Constraints](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px4-capacity-estimation-and-constraintsdiv)
  + [5. High Level System Design](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px5-high-level-system-designdiv)
  + [6. Database Schema](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px6-database-schema)
  + [7. Data Size Estimation](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px7-data-size-estimationdiv)
  + [8. Component Design](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px8-component-designdiv)
  + [9. Reliability and Redundancy](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px9-reliability-and-redundancydiv)
  + [10. Data Sharding](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px10-data-shardingdiv)
  + [11. Ranking and News Feed Generation](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px11-ranking-and-news-feed-generationdiv)
  + [12. News Feed Creation with Sharded Data](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px12-news-feed-creation-with-sharded-datadiv)
  + [13. Cache and Load balancing](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px13-cache-and-load-balancingdiv)

## 1. What is Instagram?

Instagram is a social networking service that enables its users to upload and share their photos and videos with other users. Instagram users can choose to share information either publicly or privately. Anything shared publicly can be seen by any other user, whereas privately shared content can only be accessed by the specified set of people. Instagram also enables its users to share through many other social networking platforms, such as Facebook, Twitter, Flickr, and Tumblr.

We plan to design a simpler version of Instagram for this design problem, where a user can share photos and follow other users. The ‘News Feed’ for each user will consist of top photos of all the people the user follows.

## 2. Requirements and Goals of the System

We’ll focus on the following set of requirements while designing Instagram:

**Functional Requirements**

1. Users should be able to upload/download/view photos.
2. Users can perform searches based on photo/video titles.
3. Users can follow other users.
4. The system should generate and display a user’s News Feed consisting of top photos from all the people the user follows.

**Non-functional Requirements**

1. Our service needs to be highly available.
2. The acceptable latency of the system is 200ms for News Feed generation.
3. Consistency can take a hit (in the interest of availability) if a user doesn’t see a photo for a while; it should be fine.
4. The system should be highly reliable; any uploaded photo or video should never be lost.

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

## 3. Some Design Considerations

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

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

## 4. Capacity Estimation and Constraints

* Let’s assume we have 500M total users, with 1M daily active users.
* 2M new photos every day, 23 new photos every second.
* Average photo file size => 200KB
* Total space required for 1 day of photos

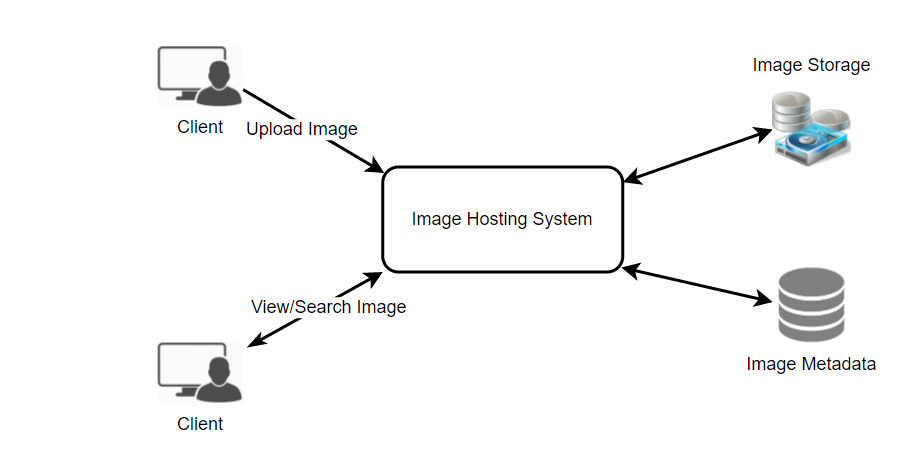
2M \* 200KB => 400 GB

* Total space required for 10 years:

400GB \* 365 (days a year) \* 10 (years) ~= 1425TB

## 5. High Level System Design

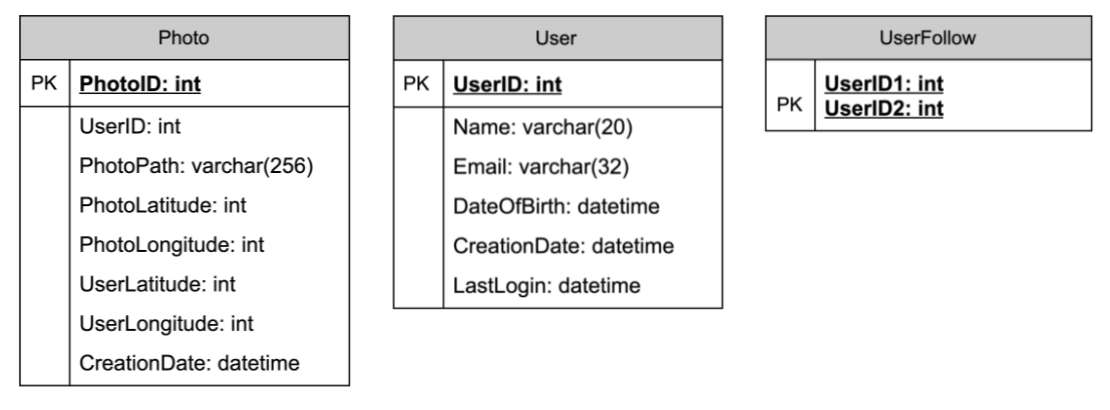
At a high-level, we need to support two scenarios, one to upload photos and the other to view/search photos. Our service would need some [object storage](https://en.wikipedia.org/wiki/Object_storage) servers to store photos and some database servers to store metadata information about the photos.



## 6. Database Schema

💡     ***Defining the DB schema in the early stages of the interview would help to understand the data flow among various components and later would guide towards data partitioning.***

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



A straightforward approach for storing the above schema would be to use an RDBMS like MySQL since we require joins. But relational databases come with their challenges, especially when we need to scale them. For details, please take a look at [SQL vs. NoSQL](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5728116278296576/).

We can store photos in a distributed file storage like [HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop) or [S3](https://en.wikipedia.org/wiki/Amazon_S3).

We can store the above schema in a distributed key-value store to enjoy the benefits offered by NoSQL. All the metadata related to photos can go to a table where the ‘key’ would be the ‘PhotoID’ and the ‘value’ would be an object containing PhotoLocation, UserLocation, CreationTimestamp, etc.

If we go with a NoSQL database, we need an additional table to store the relationships between users and photos to know who owns which photo. Let’s call this table ‘UserPhoto’. We also need to store the list of people a user follows. Let’s call it ‘UserFollow’. For both of these tables, we can use a wide-column datastore like [Cassandra](https://en.wikipedia.org/wiki/Apache_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. Data Size Estimation

Let’s estimate how much data will be going into each table and how much total storage we will need for 10 years.

**User:** Assuming each “int” and “dateTime” is four bytes, each row in the User’s table will be of 68 bytes:

UserID (4 bytes) + Name (20 bytes) + Email (32 bytes) + DateOfBirth (4 bytes) + CreationDate (4 bytes) + LastLogin (4 bytes) = 68 bytes

If we have 500 million users, we will need 32GB of total storage.

500 million \* 68 ~= 32GB

**Photo:** Each row in Photo’s table will be of 284 bytes:

PhotoID (4 bytes) + UserID (4 bytes) + PhotoPath (256 bytes) + PhotoLatitude (4 bytes) + PhotoLongitude(4 bytes) + UserLatitude (4 bytes) + UserLongitude (4 bytes) + CreationDate (4 bytes) = 284 bytes

If 2M new photos get uploaded every day, we will need 0.5GB of storage for one day:

2M \* 284 bytes ~= 0.5GB per day

For 10 years we will need 1.88TB of storage.

**UserFollow:** Each row in the UserFollow table will consist of 8 bytes. If we have 500 million users and on average each user follows 500 users. We would need 1.82TB of storage for the UserFollow table:

500 million users \* 500 followers \* 8 bytes ~= 1.82TB

Total space required for all tables for 10 years will be 3.7TB:

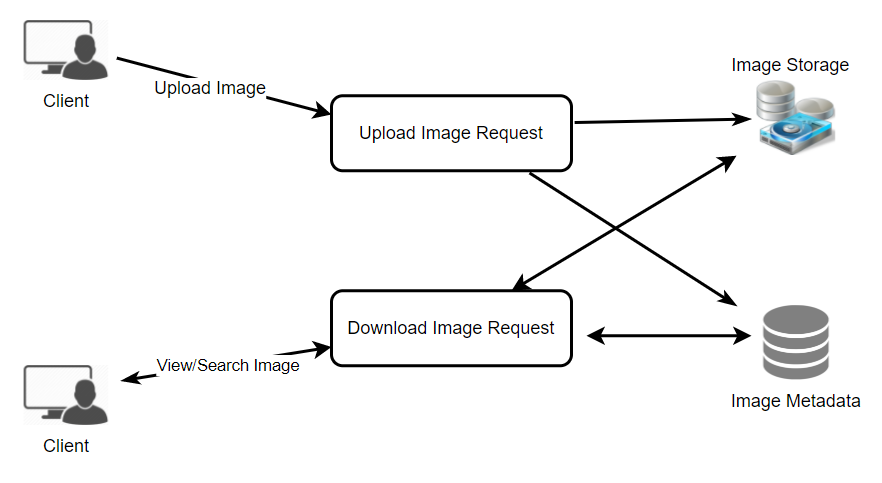
32GB + 1.88TB + 1.82TB ~= 3.7TB

## 8. Component Design

Photo uploads (or writes) can be slow as they have to go to the disk, whereas reads will be faster, especially if they are being served from cache.

Uploading users can consume all the available connections, as uploading is a slow process. This means that ‘reads’ cannot be served if the system gets busy with all the ‘write’ requests. We should keep in mind that web servers have a connection limit before designing our system. If we assume that a web server can have a maximum of 500 connections at any time, then it can’t have more than 500 concurrent uploads or reads. To handle this bottleneck, we can split reads and writes into separate services. We will have dedicated servers for reads and different servers for writes to ensure that uploads don’t hog the system.

Separating photos’ read and write requests will also allow us to scale and optimize each of these operations independently.



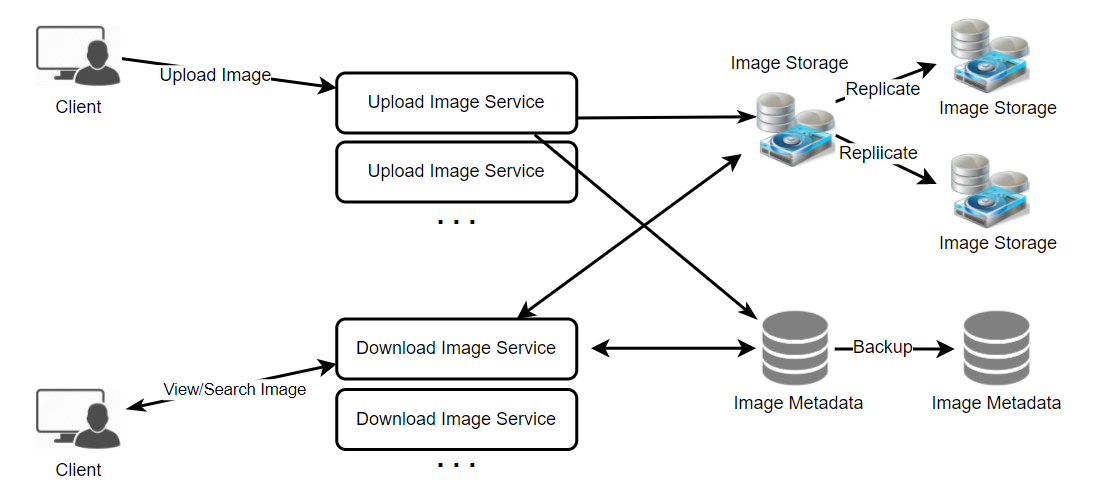
## 9. Reliability and Redundancy

Losing files is not an option for our service. Therefore, we will store multiple copies of each file so that if one storage server dies, we can retrieve the photo from the other copy present on a different storage server.

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

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

Creating redundancy in a system can remove single points of failure and provide a backup or spare functionality if needed in a crisis. For example, if there are two instances of the same service running in production and one fails or degrades, the system can failover to the healthy copy. Failover can happen automatically or require manual intervention.



## 10. Data Sharding

Let’s discuss different schemes for metadata sharding:

**a. Partitioning based on UserID** Let’s assume we shard based on the ‘UserID’ so that we can keep all photos of a user on the same shard. If one DB shard is 1TB, we will need four shards to store 3.7TB of data. Let’s assume, for better performance and scalability, we keep 10 shards.

So we’ll find the shard number by **UserID % 10** and then store the data there. To uniquely identify any photo in our system, we can append the shard number with each PhotoID.

**How can we generate 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 the different issues with this partitioning scheme?**

1. How would we handle hot users? Several people follow such hot users, and a lot of other people see any photo they upload.
2. Some users will have a lot of photos compared to others, thus making a non-uniform distribution of storage.
3. What if we cannot store all pictures of a user on one shard? If we distribute photos of a user onto multiple shards, will it cause higher latencies?
4. Storing all photos of a user on one shard can cause issues like unavailability of all of the user’s data if that shard is down or higher latency if it is serving high load etc.

**b. Partitioning based on PhotoID** If we can generate unique PhotoIDs first and then find a shard number through “PhotoID % 10”, the above problems will have been solved. 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 because we need to know 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 would be. A workaround for that could be to define two such databases, one generating even-numbered IDs and the other odd-numbered. For MySQL, the following script can define such sequences:

KeyGeneratingServer1:  
auto-increment-increment = 2  
auto-increment-offset = 1  
  
KeyGeneratingServer2:  
auto-increment-increment = 2  
auto-increment-offset = 2

We can put a load balancer in front of both of these databases to round-robin between them and to deal with downtime. Both these servers could be out of sync, with one generating more keys than the other, but this will not cause any issue in our system. We can extend this design by defining separate ID tables for Users, Photo-Comments, or other objects present in our system.

**Alternately,** we can implement a ‘key’ generation scheme similar to what we have discussed in [Designing a URL Shortening service like TinyURL](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904).

**How can we plan for the future growth of our system?** We can have a large number of logical partitions to accommodate future data growth, such that in the beginning, multiple logical partitions reside on a single physical database server. Since each database server can have multiple database instances running on it, we can have separate databases for each logical partition on any server. So whenever we feel that a particular database server has a lot of data, we can migrate some logical partitions from it to another server. We can maintain a config file (or a separate database) that can map our logical partitions to database servers; this will enable us to move partitions around easily. Whenever we want to move a partition, we only have to update the config file to announce the change.

## 11. Ranking and News Feed Generation

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

For simplicity, let’s assume we need to fetch the top 100 photos for a user’s News Feed. Our application server will first get a list of people the user follows and then fetch metadata info of each user’s latest 100 photos. In the final step, the server will submit all these photos to our ranking algorithm, which will determine the top 100 photos (based on recency, likeness, etc.) and return them to the user. A possible problem with this approach would be higher latency as we have to query multiple tables and perform sorting/merging/ranking on the results. To improve the efficiency, we can pre-generate the News Feed and store it in a separate table.

**Pre-generating the News Feed:** We can have dedicated servers that are continuously generating users’ News Feeds and storing them in a ‘UserNewsFeed’ table. So whenever any user needs the latest photos for their News-Feed, we will simply query this table and return the results to the user.

Whenever these servers need to generate the News Feed of a user, they will first query the UserNewsFeed table to find the last time the News Feed was generated for that user. Then, new News-Feed data will be generated from that time onwards (following the steps mentioned above).

**What are the different approaches for sending News Feed contents to the users?**

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

**2. Push:** Servers can push new data to the users as soon as it is available. To efficiently manage this, users have to maintain a [Long Poll](https://en.wikipedia.org/wiki/Push_technology#Long_polling) request with the server for receiving the updates. A possible problem with this approach is a user who follows a lot of people or a celebrity user who has millions of followers; in this case, the server has to push updates quite frequently.

**3. Hybrid:** We can adopt a hybrid approach. We can move all the users who have a high number of followers to a pull-based model and only push data to those who have a few hundred (or thousand) follows. Another approach could be that the server pushes updates to all the users not more than a certain frequency and letting users with a lot of followers/updates to pull data regularly.

For a detailed discussion about News-Feed generation, take a look at [Designing Facebook’s Newsfeed](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5641332169113600).

## 12. News Feed Creation with Sharded Data

One of the most important requirements to create the News Feed for any given user is to fetch the latest photos from all people the user follows. For this, we need to have a mechanism to sort photos on their time of creation. To efficiently do this, we can make photo creation time part of the PhotoID. As we will have a primary index on PhotoID, it will be quite quick to find the latest PhotoIDs.

We can use epoch time for this. Let’s say our PhotoID will have two parts; the first part will be representing epoch time, and the second part will be an auto-incrementing sequence. So to make a new PhotoID, we can take the current epoch time and append an auto-incrementing ID from our key-generating DB. We can figure out the shard number from this PhotoID ( PhotoID % 10) and store the photo there.

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

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

We would need 31 bits to store this number. Since, on average, we are expecting 23 new photos per second, we can allocate 9 additional bits to store the auto-incremented sequence. So every second, we can store (2^9 => 512)(2​9​​=>512) new photos. We are allocating 9 bits for the sequence number which is more than what we require; we are doing this to get a full byte number (as 40 bits = 5 bytes40*bits*=5*bytes*). We can reset our auto-incrementing sequence every second.

We will discuss this technique under ‘Data Sharding’ in [Designing Twitter](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5741031244955648).

## 13. Cache and Load balancing

## [**#**](https://www.educative.io/courses/grokking-the-system-design-interview/m2yDVZnQ8lG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px13-cache-and-load-balancingdiv)

Our service would need a massive-scale photo delivery system to serve globally distributed users. Our service should push its content closer to the user using a large number of geographically distributed photo cache servers and use CDNs (for details, see [Caching](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5643440998055936/)).

We can introduce a cache for metadata servers to cache hot database rows. We can use Memcache to cache the data, and Application servers before hitting the database, can quickly check if the cache has desired rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system. Under this policy, we discard the least recently viewed row first.

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

# 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

**We'll cover the following**

* + [1. Why Cloud Storage?](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px1-why-cloud-storagediv)
  + [2. Requirements and Goals of the System](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px2-requirements-and-goals-of-the-systemdiv)
  + [3. Some Design Considerations](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px3-some-design-considerationsdiv)
  + [4. Capacity Estimation and Constraints](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px4-capacity-estimation-and-constraintsdiv)
  + [5. High Level Design](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px5-high-level-designdiv)
  + [6. Component Design](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px6-component-designdiv)
    - [a. Client](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#a-client)
    - [b. Metadata Database](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#b-metadata-database)
    - [c. Synchronization Service](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#c-synchronization-service)
    - [d. Message Queuing Service](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#d-message-queuing-service)
    - [e. Cloud/Block Storage](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#e-cloudblock-storage)
  + [7. File Processing Workflow](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px7-file-processing-workflowdiv)
  + [8. Data Deduplication](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px8-data-deduplicationdiv)
  + [9. Metadata Partitioning](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px9-metadata-partitioningdiv)
  + [10. Caching](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5pxpadding5px10-cachingdiv)
  + [11. Load Balancer (LB)](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5pxpadding5px11-load-balancer-lbdiv)
  + [12. Security, Permissions and File Sharing](https://www.educative.io/courses/grokking-the-system-design-interview/m22Gymjp4mG#div-stylecolorblack-background-colore2f4c7-border-radius5pxpadding5px12-security-permissions-and-file-sharingdiv)

## 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 each with portable access from various geographical locations at any time, is believed to be accountable for the huge popularity of cloud storage services. Following are some of the top benefits of such services:

**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](http://dropbox.com/) before, we would highly recommend creating an account there and uploading/editing a file and also going through the different options their service offers. This will help you a lot in understanding this chapter.

## 2. Requirements and Goals of the System

💡 **You should always clarify requirements at the beginning of the interview. Be sure to ask questions to find the exact scope of the system that the interviewer has in mind.**

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 i.e. 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.

## 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 the 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 for 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 sizes 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 Clients?** Keeping a local copy of metadata not only enables 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 with 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 a 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 an empty response most of the time, but will also be keeping the server busy. Pulling information in this manner is not scalable.

A solution to the 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.

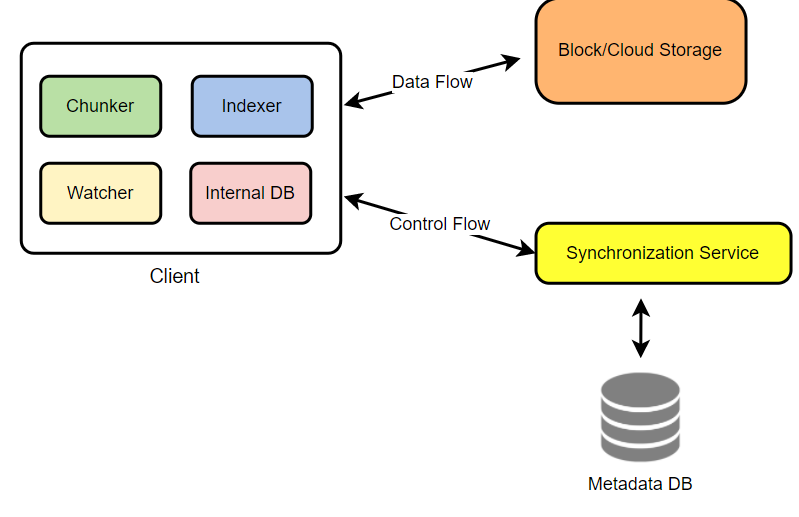
Based on the above considerations, we can divide our client into four parts:

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 the 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, 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 is working 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 database. However, using a relational database can simplify the implementation of the Synchronization Service as they natively support ACID properties.

The 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 they come 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.

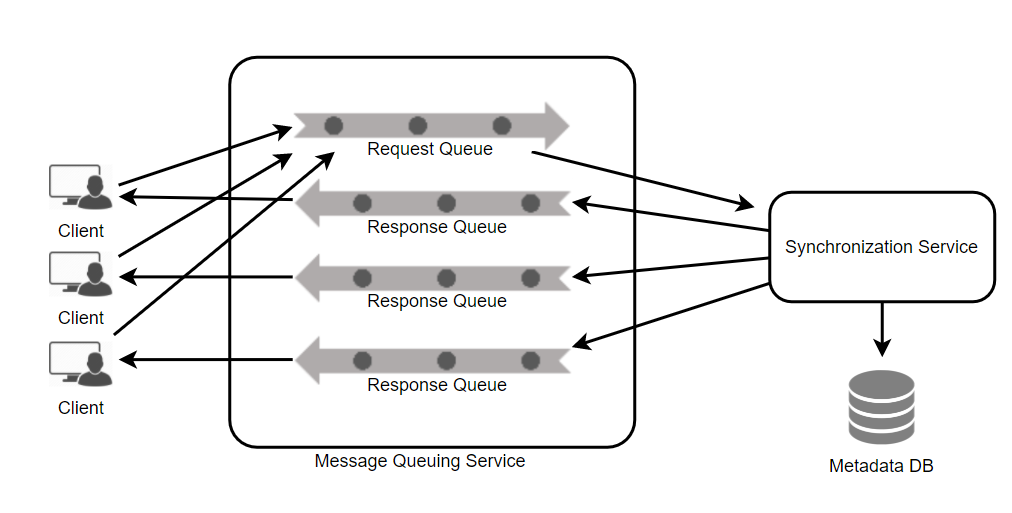
The Synchronization Service should be designed to transmit less data between clients and the Cloud Storage to achieve a better response time. To meet this design goal, the Synchronization Service can employ a differencing algorithm to reduce the amount of data that needs to be synchronized. Instead of 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 the 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 notifications 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](https://en.wikipedia.org/wiki/Message_queue), and the communication middleware will be able to balance its load.

### d. Message Queuing Service

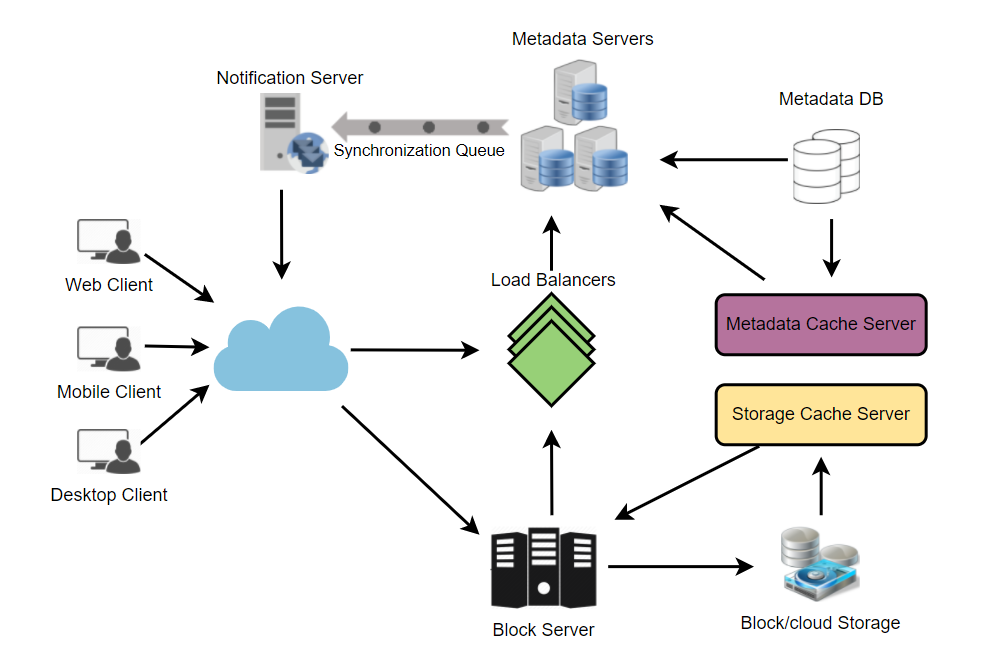
An important part of our architecture is a messaging middleware that should be able to handle a substantial number of requests. A scalable Message Queuing Service that supports asynchronous message-based communication between clients and the Synchronization Service best fits the requirements of our application. 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.

The Message Queuing Service will implement two types of queues in our system. The Request Queue is a global queue and all clients will share it. Clients’ requests to update the Metadata Database will be sent to the Request Queue first; from there, the 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](https://cloudacademy.com/blog/object-storage-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 the cloud or in-house.



## 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 are not online at the time of the update, the Message Queuing Service keeps the update notifications in separate response queues for them until they come 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 the 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 that it has already stored, only a reference to the existing chunk will be added in the metadata, rather than a 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 in 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 storing such a 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? In that case, we save all the files starting with the letter ‘A’ in one partition and those that start with the 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 the letter ‘E’ into a DB partition, and later we realize that we have too many files that start with the 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](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5709068098338816).

## 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 [Memcached](https://en.wikipedia.org/wiki/Memcached) that can store whole chunks with its respective IDs/Hashes and Block servers before hitting Block storage can quickly check if the cache has desired chunk. Based on clients’ usage patterns we can determine how many cache servers we need. A high-end commercial server can have 144GB of memory; 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 the 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 servers about their load and adjusts traffic based on that.

## 12. Security, Permissions and File Sharing

One of the primary concerns users will have while storing their files in the cloud is 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 them with everyone. To handle this, we will be storing the 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. Difficulty Level: Medium

**We'll cover the following**

* + [1. What is Facebook Messenger?](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-styleblackwhite-background-colore2f4c7-border-radius5px-padding5px1-what-is-facebook-messengerdiv)
  + [2. Requirements and Goals of the System](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px2-requirements-and-goals-of-the-systemdiv)
  + [3. Capacity Estimation and Constraints](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px3-capacity-estimation-and-constraintsdiv)
  + [4. High Level Design](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px4-high-level-designdiv)
  + [5. Detailed Component Design](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px5-detailed-component-designdiv)
  + [a. Messages Handling](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#a-messages-handling)
  + [b. Storing and retrieving the messages from the database](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#b-storing-and-retrieving-the-messages-from-the-database)
  + [c. Managing user’s status](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#c-managing-users-status)
  + [6. Data partitioning](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px6-data-partitioningdiv)
  + [7. Cache](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px7-cachediv)
  + [8. Load balancing](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px8-load-balancingdiv)
  + [9. Fault tolerance and Replication](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px9-fault-tolerance-and-replicationdiv)
  + [10. Extended Requirements](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px10-extended-requirementsdiv)
  + [a. Group chat](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#a-group-chat)
  + [b. Push notifications](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#b-push-notifications)

## 1. What is Facebook Messenger?

Facebook Messenger is a software application that provides text-based instant messaging services to its users. Messenger users can chat with their Facebook friends both from cell phones and Facebook’s 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 the online/offline statuses of its users.
3. Messenger should support the persistent storage of chat history.

**Non-functional Requirements:**

1. Users should have a real-time chatting experience with minimum latency.
2. Our system should be highly consistent; users should 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

To store five years of chat history, we would need 3.6 petabytes of storage.

2 TB \* 365 days \* 5 years ~= 3.6 PB

Besides chat messages, we also need to store users’ information, messages’ metadata (ID, Timestamp, etc.). Not to mention, the above calculation doesn’t take data compression and replication into 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

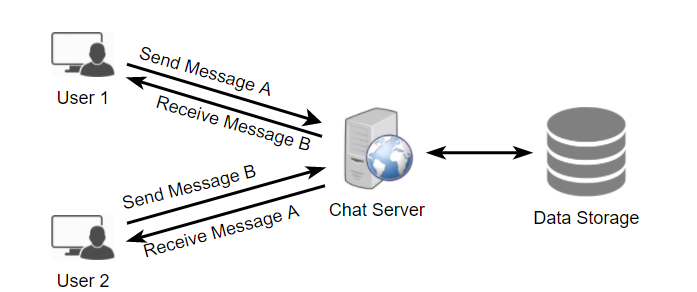
Since each incoming message needs to go out to another user, we will need the same amount of bandwidth 25MB/s for both upload and download.

**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 |

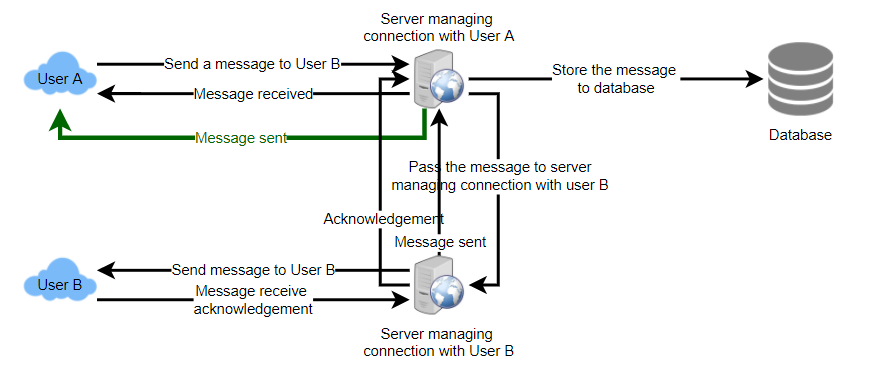
## 4. High Level Design

At a high level, we will need a chat server that will be the central piece orchestrating all the communications between users. For example, 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.

**1** of 8

## 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 the 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.

In the first approach, 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 messages. 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 the pending messages, and we will have minimum latency, as the messages are delivered instantly on the opened connection.

**How will clients maintain an open connection with the server?** We can use HTTP [Long Polling](https://www.educative.io/courses/grokking-the-system-design-interview/gx7wZzWn5Vj#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5pxhttp-long-pollingdiv) or [WebSockets](https://www.educative.io/courses/grokking-the-system-design-interview/gx7wZzWn5Vj" \l "div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5pxwebsocketsdiv" \t "_blank). 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. However, the long polling request can timeout or receive a disconnect from the server; in that case, the client has to open a new request.

**How can the server keep track of all the opened connections 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. However, 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 do 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 do we 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 the following things upon receiving a new message: 1) Store the message in the database, 2) Send the message to the receiver, and 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 to store 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 is the time when the server receives the message. However, this will still not ensure the correct ordering of messages for clients. The scenario where the server timestamp cannot determine the exact order 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 the 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 the message sequence, but this view will be consistent for them on all devices.

## b. Storing and retrieving the messages from the database [**#**](https://www.educative.io/courses/grokking-the-system-design-interview/B8R22v0wqJo#b-storing-and-retrieving-the-messages-from-the-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 the database connection pool.
2. How to retry failed requests.
3. Where to log those requests that failed even after some retries.
4. How to retry these logged requests (that failed after the retry) when all the issues have been resolved.

**Which storage system should we use?** We need to have a database that can support a very high rate of small updates and also fetch a range of records quickly. This is required because we have a huge number of small messages that need to be inserted in the database and, while querying, a user is mostly interested in sequentially accessing the messages.

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 not only make the basic operations of our service run with high latency but also create a huge load on databases.

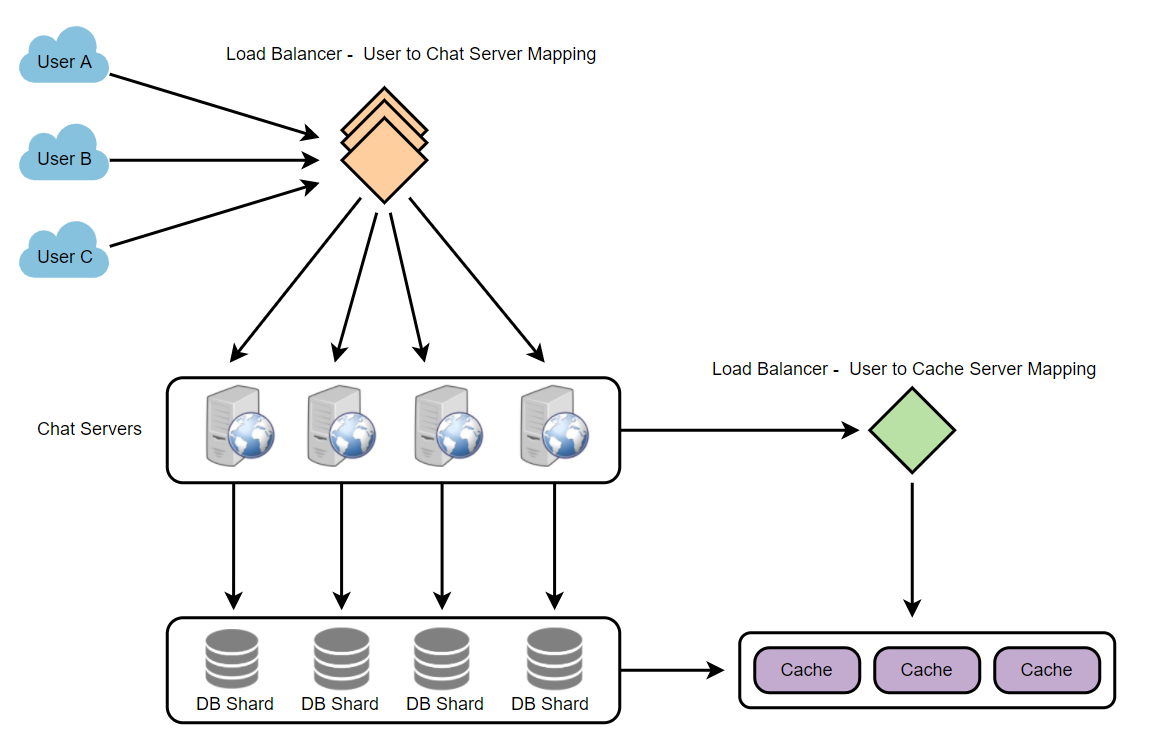
Both of our requirements can be easily met with a wide-column database solution like [HBase](https://en.wikipedia.org/wiki/Apache_HBase). HBase is a column-oriented key-value NoSQL database that can store multiple values against one key into multiple columns. HBase is modeled after Google’s [BigTable](https://en.wikipedia.org/wiki/Bigtable" \t "_blank) and runs on top of Hadoop Distributed File System ([HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop)). 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 to store 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-sized 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 fewer messages/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 the 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 the 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 a few seconds to see if the user does not go offline immediately.
4. Clients 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.

**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. So, what will 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 unlimited history of messages, we can start with a large number of logical partitions that will be mapped to fewer physical servers. Then, 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 shards, 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 a user’s viewport (say last 5). Since we decided to store all of the user’s messages on one shard, the cache for a user should entirely 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 store only one copy of the 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, users can only send messages to online users; 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.

For Push notifications, each user can opt-in from their device (or a web browser) to get notifications whenever there is a new message or event. Each manufacturer maintains a set of servers that handles pushing these notifications to the user.

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 the manufacturer’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

**We'll cover the following**

* + [1. What is Twitter?](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px1-what-is-twitter)
  + [2. Requirements and Goals of the System](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px2-requirements-and-goals-of-the-system)
  + [3. Capacity Estimation and Constraints](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px3-capacity-estimation-and-constraints)
  + [4. System APIs](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5pxpadding5px4-system-apisdiv)
  + [5. High Level System Design](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px5-high-level-system-design)
  + [6. Database Schema](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px6-database-schema)
  + [7. Data Sharding](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px7-data-sharding)
  + [8. Cache](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px8-cache)
  + [9. Timeline Generation](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px9-timeline-generation)
  + [10. Replication and Fault Tolerance](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px10-replication-and-fault-tolerance)
  + [11. Load Balancing](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px11-load-balancing)
  + [12. Monitoring](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px12-monitoring)
  + [13. Extended Requirements](https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px13-extended-requirements)

## 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 the 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 as favorites.
4. The service should be able to create and display a 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 for tweets.
2. Replying 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 assume 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 will our system generate?** Let’s assume on average a user visits their timeline two times a day and visits five other people’s pages. On each page if a user sees 20 tweets, then our system will generate 28B/day total tweet-views:

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 our storage needs be for five years? How much storage we would need for users’ data, follows, favorites? We will leave this for the 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

💡     ***Once we've finalized the requirements, it's always a good idea to define the system APIs. This should explicitly state what is expected from the system.***

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)

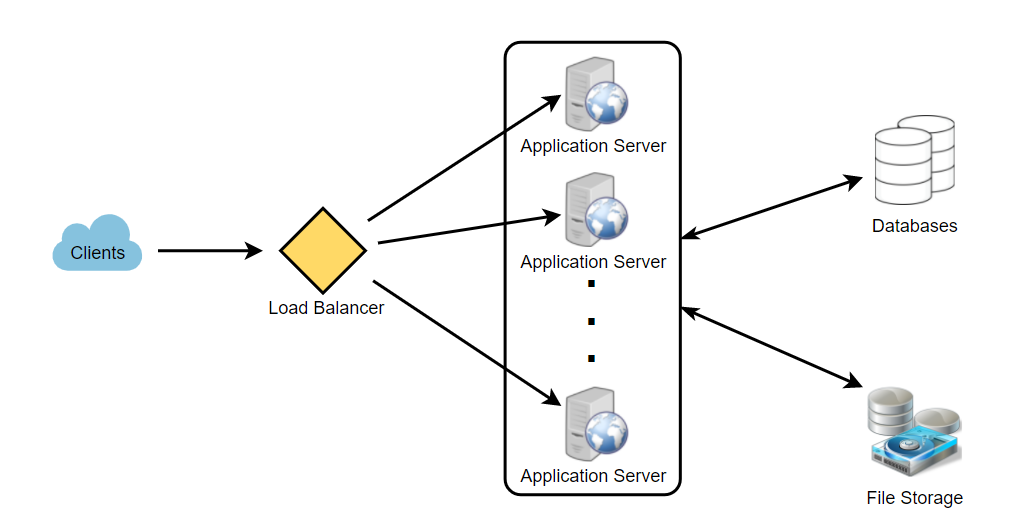
**Parameters:**  
api\_dev\_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.  
tweet\_data (string): The text of the tweet, typically up to 140 characters.  
tweet\_location (string): Optional location (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.

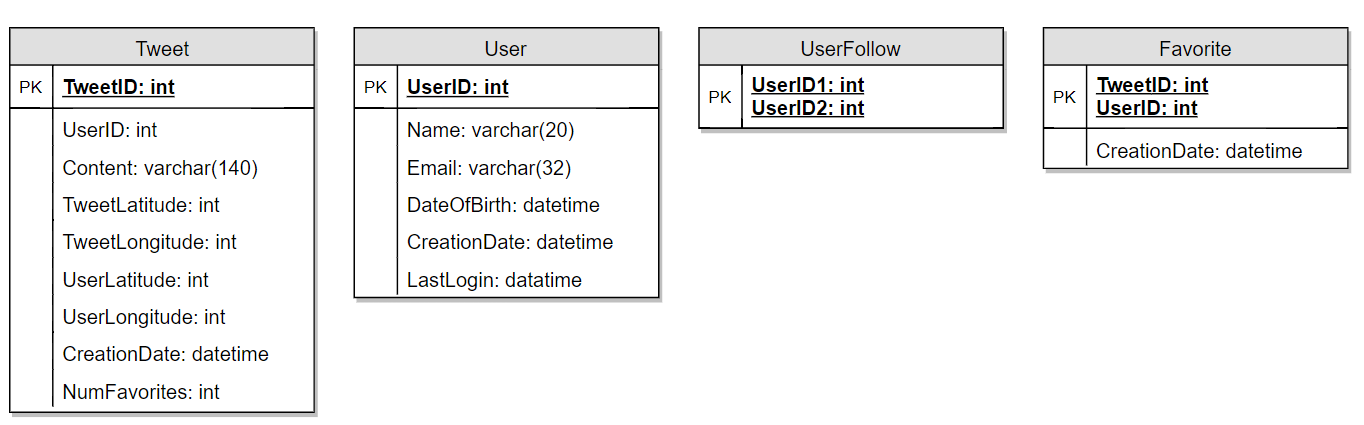
At a high level, we need multiple application servers to serve all these requests with load balancers in front of them for traffic distributions. On the backend, we need an efficient database that can store all the new tweets and can support a huge number of reads. We also need some file storage to store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



For choosing between SQL and NoSQL databases to store the above schema, please see ‘Database schema’ under [Designing Instagram](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5673385510043648).

## 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 having a lot of follows compared to others. Maintaining a uniform distribution of growing user 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 for 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.

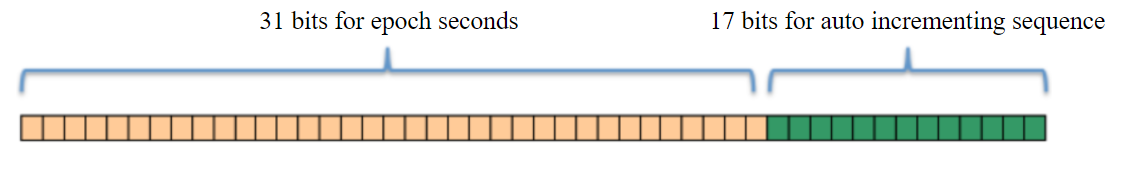
**Sharding based on Tweet creation time:** Storing tweets based on creation time will give us the advantage of fetching all the top tweets quickly and we only have to query a very small set of servers. The problem here is that the traffic load will not be distributed, e.g., while writing, all new tweets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

**What if we can combine sharding by TweetID 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 the latest Tweets. For this, we must make each TweetID universally unique in our system and each TweetID should contain a 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 the 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 the 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 the next 100 years and also store them for mili-seconds granularity.

In the above approach, we still have to query all the servers for timeline generation, but our reads (and writes) will be substantially quicker.

1. Since we don’t have any secondary index (on creation time) this will reduce our write latency.
2. While reading, we don’t need to filter on creation-time as our primary key has epoch time included in it.

## 8. Cache

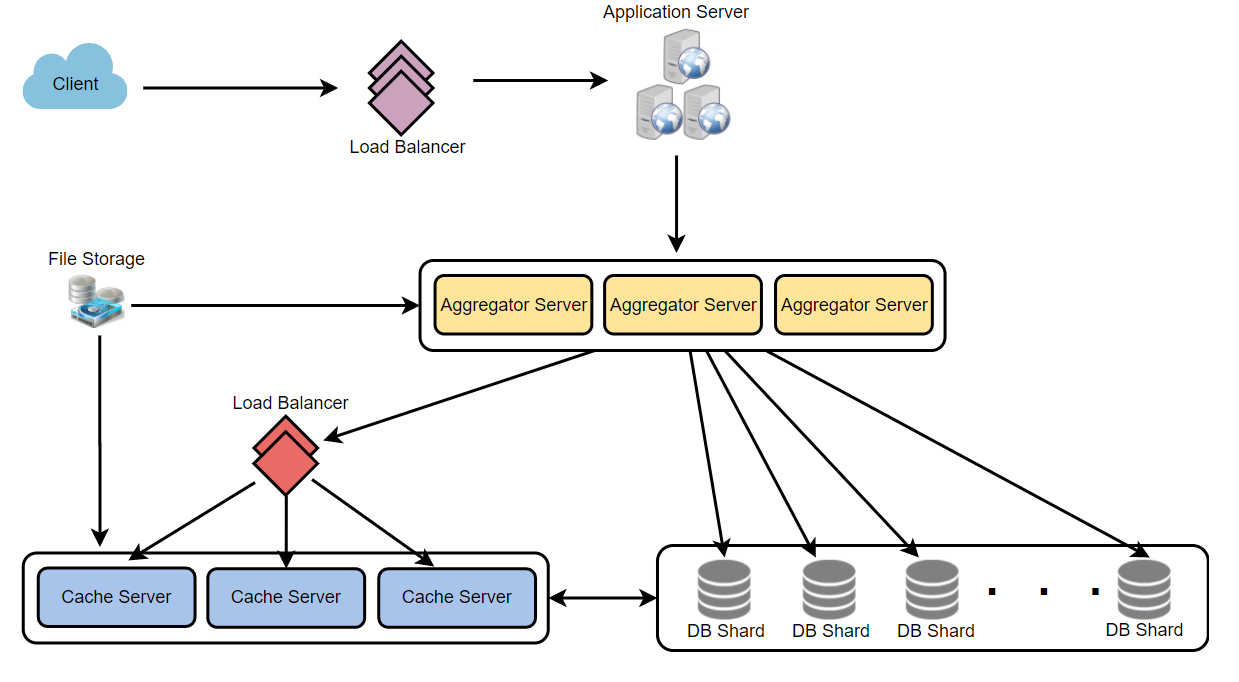
We can introduce a cache for database servers to cache hot tweets and users. We can use an off-the-shelf solution like Memcache that can store the whole tweet objects. Application servers, before hitting database, can quickly check if the cache has desired tweets. Based on clients’ usage patterns we 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 a more intelligent cache?** If we go with 80-20 rule, that is 20% of tweets generating 80% of read traffic which means that certain tweets are so popular that a 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 the past three days only; we can try to cache all the tweets from the past three days. Let’s say we have dedicated cache servers that cache all the tweets from all the users from the 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 will 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 the backend server to fetch that data. On a similar design, we can try caching photos and videos from the 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 the past three days. Since we want to retrieve the 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](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5641332169113600).

## 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, since 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 that if a server is dead, LB will take it out of the rotation and will stop sending any traffic to it. A problem with Round Robin LB is that it won’t take servers 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, load balancing, or caching.

## 13. Extended Requirements

**How do we 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, since on a 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](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5673385510043648).

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

**Trending Topics:** We can cache most frequently occurring hashtags or search 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 levels 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, financial, 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](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5738600293466112).

# 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

**We'll cover the following**

* + [1. Why Youtube?](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px1-why-youtubediv)
  + [2. Requirements and Goals of the System](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px2-requirements-and-goals-of-the-systemdiv)
  + [3. Capacity Estimation and Constraints](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px3-capacity-estimation-and-constraintsdiv)
  + [4. System APIs](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5pxpadding5px4-system-apisdiv)
  + [5. High Level Design](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px5-high-level-designdiv)
  + [6. Database Schema](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px6-database-schemadiv)
  + [7. Detailed Component Design](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px7-detailed-component-designdiv)
  + [8. Metadata Sharding](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px8-metadata-shardingdiv)
  + [9. Video Deduplication](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px9-video-deduplicationdiv)
  + [10. Load Balancing](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px10-load-balancingdiv)
  + [11. Cache](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px11-cachediv)
  + [12. Content Delivery Network (CDN)](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px12-content-delivery-network-cdndiv)
  + [13. Fault Tolerance](https://www.educative.io/courses/grokking-the-system-design-interview/xV26VjZ7yMl#div-stylecolorblack-background-colore2f4c7-border-radius5px-padding5px13-fault-tolerancediv)

## 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 should be able to 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 a real-time experience while watching videos and should not feel any lag.

**Not in scope:** Video recommendations, most popular videos, channels, 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 average, a user views five videos per day then the 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 videos 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), the total storage needed for videos uploaded in a minute would be:

500 hours \* 60 min \* 50MB => 1500 GB/min (25 GB/sec)

These are estimated numbers ignoring video compression and replication, which would change real numbers.

**Bandwidth estimates:** With 500 hours of video uploads per minute (which is 30000 mins of video uploads per minute), assuming uploading each minute of the video takes 10MB of the bandwidth, 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. The following could be the definitions of the APIs for uploading and searching videos:

uploadVideo(api\_dev\_key, video\_title, video\_description, tags[], category\_id, default\_language, recording\_details, video\_contents)

**Parameters:**  
api\_dev\_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.  
video\_title (string): Title of the video.  
video\_description (string): Optional description of the video.  
tags (string[]): Optional tags for the video.  
category\_id (string): Category of the video, e.g., Film, Song, People, etc.  
default\_language (string): For example English, Mandarin, Hindi, etc.  
recording\_details (string): Location where the video was recorded.  
video\_contents (stream): Video to be uploaded.

**Returns:** (string)  
A successful upload will return HTTP 202 (request accepted) 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 a view count.

streamVideo(api\_dev\_key, video\_id, offset, codec, resolution)

**Parameters:**  
api\_dev\_key (string): The API developer key of a registered account of our service.  
video\_id (string): A string to identify the video.  
offset (number): We should be able to stream video from any offset; this offset would be a time in seconds from the beginning of the video. If we support playing/pausing a video from multiple devices, we will need to store the offset on the server. This will enable the users to start watching a video on any device from the same point where they left off.  
codec (string) & resolution(string): We should send the codec and resolution info in the API from the client to support play/pause from multiple devices. Imagine you are watching a video on your TV’s Netflix app, paused it, and started watching it on your phone’s Netflix app. In this case, you would need codec and resolution, as both these devices have a different resolution and use a different codec.

**Returns:** (STREAM)  
A media stream (a video chunk) from the given offset.

## 5. High Level Design

At a high-level we would need the 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:** To generate a few thumbnails for each video.
4. **Video and Thumbnail storage:** To store video and thumbnail files in some distributed file storage.
5. **User Database:** To store user’s information, e.g., name, email, address, etc.
6. **Video metadata storage:** A metadata database to store all the information about videos like title, file path in the system, uploading user, total views, likes, dislikes, etc. It will also be used to store all the video comments.

## 6. Database Schema

**Video metadata storage - MySql**  
Videos metadata can be stored in a SQL database. The 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 to be 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](https://en.wikipedia.org/wiki/Apache_Hadoop#HDFS) or [GlusterFS](https://en.wikipedia.org/wiki/GlusterFS" \t "_blank).

**How should we efficiently manage read traffic?** We should segregate our read traffic from write traffic. Since we will have multiple copies of each video, we can distribute our read traffic on different servers. For metadata, we can have primary-secondary configurations where writes will go to primary first and then get applied at all the secondaries. Such configurations can cause some staleness in data, e.g., when a new video is added, its metadata would be inserted in the primary first, and before it gets applied to the secondary, our secondaries would not be able to see it; and therefore, it 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 would 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 huge read traffic. There will be two consideration before deciding which storage system should be used for thumbnails:

1. Thumbnails are small files, say, a maximum of 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 with 20 thumbnails of other videos.

Let’s evaluate storing all the thumbnails on a disk. Given that we have a huge number of files, we have to perform many seeks to different locations on the disk to read these files. This is quite inefficient and will result in higher latencies.

[Bigtable](https://en.wikipedia.org/wiki/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 most significant requirements for our service. Keeping hot thumbnails in the cache will also help improve 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, the uploader will be notified, and the video is made available for view/sharing.

## 8. Metadata Sharding

Since we have a huge number of new videos every day and our read load is extremely high, therefore, 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 the 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; this could create a performance bottleneck. This will also 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 data is quite tricky.

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 a 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, contain overlays or additional borders, or 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: Duplicate videos will also increase 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 could 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](https://en.wikipedia.org/wiki/Block-matching_algorithm), [Phase Correlation](https://en.wikipedia.org/wiki/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 continue the upload and 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 the parts that are missing.

## 10. Load Balancing

We should use [Consistent Hashing](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5709068098338816/) 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 an uneven load on the logical replicas due to each video’s different popularity. 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 will 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 the 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 a more intelligent cache?** If we go with the 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 user’s geographic locations, the origin of the web page, and a content delivery server. Take a look at the ‘CDN’ section in [Caching](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5643440998055936) chapter.

Our service can move 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](https://www.educative.io/collection/page/5668639101419520/5649050225344512/5709068098338816/) for distribution among database servers. Consistent hashing will not only help in replacing a dead server but also help in distributing load among servers.