**Caching**

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

1. Application server cache

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

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

2. Distributed cache

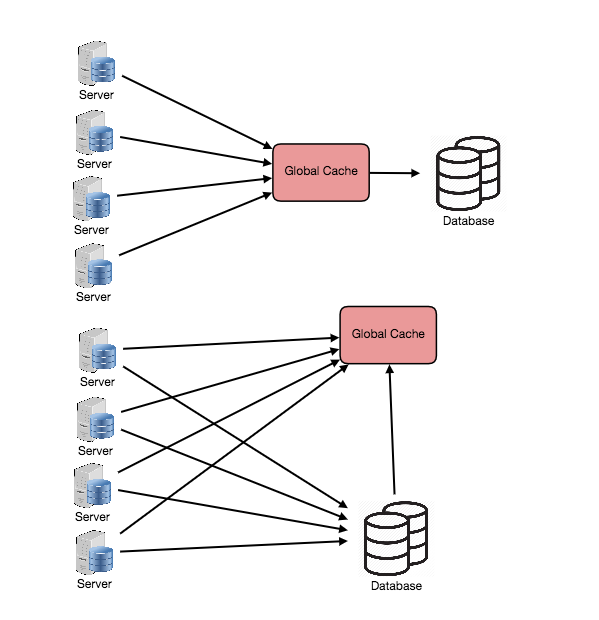
In a distributed cache, each of its nodes own part of the cached data. Typically, the cache is divided up using a consistent hashing function, such that if a request node is looking for a certain piece of data, it can quickly know where to look within the distributed cache to determine if that data is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

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

3. Global Cache

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

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



No Graph available

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

4. Content Distribution Network (CDN)

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

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

Cache Invalidation

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

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

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

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

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

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

Cache eviction policies

Following are some of the most common cache eviction policies:

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

Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.

Least Recently Used (LRU): Discards the least recently used items first.

Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first.

Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first.

Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.

**Data Partitioning**

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

1. Partitioning Methods

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

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

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

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

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

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

2. Partitioning Criteria

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

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

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

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

3. Common Problems of Sharding

On a sharded database, there are certain extra constraints on the different operations that can be performed. Most of these constraints are due to the fact that, operations across multiple tables or multiple rows in the same table, will no longer run on the same server. Below are some of the constraints and additional complexities introduced by sharding:

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

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

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

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

The data distribution is not uniform, e.g., there are a lot of places for a particular ZIP code, that cannot fit into one database partition.

There are a lot of load on a shard, e.g., there are too many requests being handled by the DB shard dedicated to user photos.

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

**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

1. Why Instagram?

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

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

2. Requirements and Goals of the System

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

Functional Requirements

Users should be able to upload/download/view photos.

Users can perform searches based on photo/video titles.

Users can follow other users.

The system should be able to generate and display a user’s timeline consisting of top photos from all the people the user follows.

Non-functional Requirements

Our service needs to be highly available.

The acceptable latency of the system is 200ms for timeline generation.

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.

The system should be highly reliable, any photo/video uploaded should not be lost.

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

3. Some Design Considerations

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

Practically users can upload as many photos as they like. Efficient management of storage should be a crucial factor while designing this system.

Low latency is expected while reading images.

Data should be 100% reliable. If a user uploads an image, the system will guarantee that it will never be lost.

4. Capacity Estimation and Constraints

Let’s assume we have 300M total users, with 1M daily active users.

2M new photos every day, 23 new photos every second.

Average photo file size => 200KB

Total space required for 1 day of photos

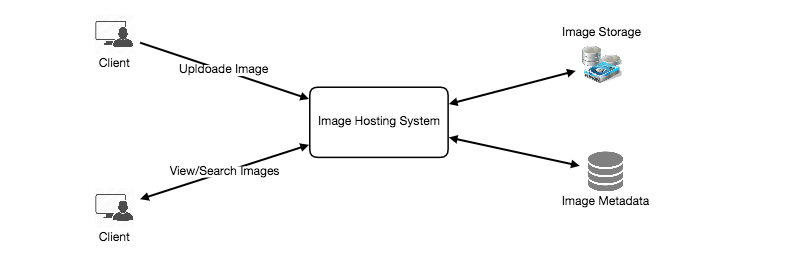
2M \* 200KB => 400 GB

Total space required for 5 years:

400GB \* 365 (days a year) \* 5 (years) ~= 712 TB

5. High Level System Design

At a high-level, we need to support two scenarios, one to upload photos and the other to view/search photos. Our service would need some block storage servers to store photos and also some database servers to store metadata information about the photos.



6. Database Schema

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 the data partitioning.

5752142325350400

One simple approach for storing the above schema would be to use an RDBMS like MySQL since we require joins. But relational databases come with their challenges, especially when we need to scale them. For details, please take a look at SQL vs. NoSQL.

We can store photos in a distributed file storage like HDFS or S3.

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

We also need to store relationships between users and photos, to know who owns which photo. Another relationship we would need to store is the list of people a user follows. For both of these tables, we can use a wide-column datastore like Cassandra. For the ‘UserPhoto’ table, the ‘key’ would be ‘UserID’ and the ‘value’ would be the list of ‘PhotoIDs’ the user owns, stored in different columns. We will have a similar scheme for the ‘UserFollow’ table.

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

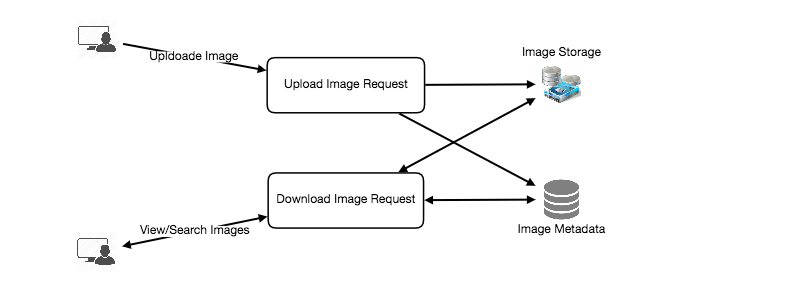
7. Component Design

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

Uploading users can consume all the connections, as uploading would be a slower process. This means reads cannot be served if the system gets busy with all the write requests. To handle this bottleneck we can split out read and writes into separate services.

Since most of the web servers have connection limit, we should keep this thing in mind before designing our system. Synchronous connection for uploads, but downloads can be asynchronous. Let’s assume if a web server can have maximum 500 connections at any time, and it can’t have more than 500 concurrent uploads simultaneously. Since reads can be asynchronous, the web server can serve a lot more than 500 users at any time, as it can switch between users quickly. This guides us to have separate dedicated servers for reads and writes so that uploads don’t hog the system.

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



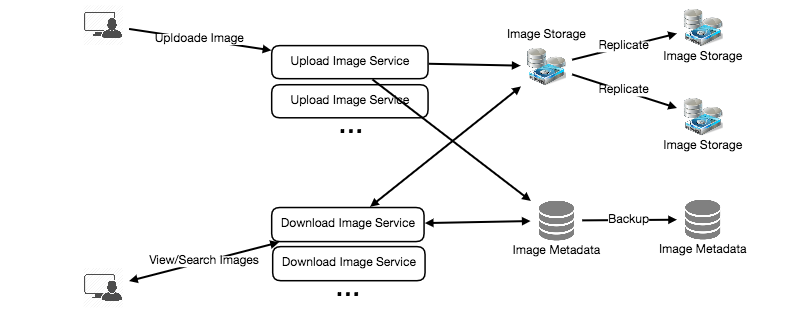
8. Reliability and Redundancy

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

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

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

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



9. Data Sharding

a. Partitioning based on UserID Let’s assume we shard based on the UserID so that we can keep all photos of a user on the same shard. If one DB shard is 4TB, we will have 712/4 => 178 shards. Let’s assume for future growths we keep 200 shards.

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

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

What are different issues with this partitioning scheme?

How would we handle hot users? Several people follow such hot users, and any photo they upload is seen by a lot of other people.

Some users will have a lot of photos compared to others, thus making a non-uniform distribution of storage.

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?

Storing all pictures of a user on one shard can cause issues like unavailability of all of the user’s data if that shard is down or higher latency if it is serving high load etc.

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

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

Wouldn’t this key generating DB be a single point of failure? Yes, it will be. A workaround for that could be, we can define two such databases, with one generating even numbered IDs and the other odd numbered. For MySQL following script can define such sequences:

KeyGeneratingServer1:

auto-increment-increment = 2

auto-increment-offset = 1

​

KeyGeneratingServer2:

auto-increment-increment = 2

auto-increment-offset = 2

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

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

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

10. Ranking and Timeline Generation

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

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

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

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

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

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

2. Push: Servers can push new data to the users as soon as it is available. To efficiently manage this, users have to maintain a Long Poll request with the server for receiving the updates. A possible problem with this approach is when a user has a lot of follows or a celebrity user who has millions of followers; in this case, the server has to push updates quite frequently.

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

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

11. Timeline Creation with Sharded Data

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

We can use epoch time for this. Let’s say our PhotoID will have two parts; the first part will be representing epoch seconds and the second part will be an auto-incrementing sequence. So to make a new PhotoID, we can take the current epoch time and append an auto incrementing ID from our key generating DB. We can figure out shard number from this PhotoID ( PhotoID % 200) and store the photo there.

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

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

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

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

12. Cache and Load balancing

To serve globally distributed users, our service needs a massive-scale photo delivery system. Our service should push its content closer to the user using a large number of geographically distributed photo cache servers and use CDNs (for details see Caching).

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

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

Designing 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, goo.gl, 2020.fm etc. Difficulty Level: Easy

1. Why do we need URL shortening?

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

For example, if we shorten this page through TinyURL:

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

We would get:

<http://tinyurl.com/jlg8zpc>

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

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

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

2. Requirements and Goals of the System

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

Our URL shortener system should meet the following requirements:

Functional Requirements:

Given a URL, our service should generate a shorter and unique alias of it.

When users access a shorter URL, our service should redirect them to the original link.

Users should optionally be able to pick a custom alias for their URL.

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

Non-Functional Requirements:

The system should be highly available. This is required because if our service is down, all the URL redirections will start failing.

URL redirection should happen in real-time with minimum latency.

Shortened links should not be guessable (not predictable).

Extended Requirements:

Analytics, e.g., how many times a redirection happened?

Our service should also be accessible through REST APIs by other services.

3. Capacity Estimation and Constraints

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

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

New URLs shortenings per second:

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

URLs redirections per second:

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

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

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

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

30 billion \* 500 bytes = 15 TB

URL Shortenings per month 500 million

Total years 5

URL object size 500 Bytes

Total Files 30 billion

Total Storage 15 TB

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

200 \* 500 bytes = 100 KB/s

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

19K \* 500 bytes ~= 9 MB/s

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

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

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

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

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

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

New URLs 200/s

URL redirections 19K/s

Incoming data 100KB/s

Outgoing data 9MB/s

Storage for 5 years 15TB

Memory for cache 170GB

1. System APIs

Once we've finalized the requirements, it's always a good idea to define the system APIs. This would 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:

creatURL(api\_dev\_key, original\_url, custom\_alias=None user\_name=None, expire\_date=None)

Parameters:

api\_dev\_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

original\_url (string): Original URL to be shortened.

custom\_alias (string): Optional custom key for the URL.

user\_name (string): Optional user name to be used in encoding.

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

Returns: (string)

A successful insertion returns the shortened URL, otherwise, returns an error code.

deleteURL(api\_dev\_key, url\_key)

Where “url\_key” is a string representing the shortened URL to be retrieved. A successful deletion returns ‘URL Removed’.

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

1. 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 the data partitioning.

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

We need to store billions of records.

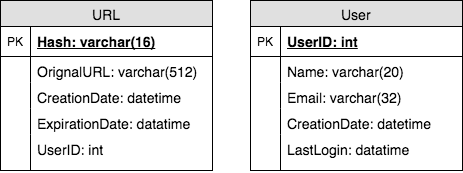
Each object we are going to store is small (less than 1K).

There are no relationships between records, except if we want to store which user created what URL.

Our service is read-heavy.

Database Schema:

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



What kind of database should we use? Since we are likely going to store billions of rows and we don’t need to use relationships between objects – a NoSQL key-value store like Dynamo or Cassandra is a better choice, which would also be easier to scale. Please see SQL vs NoSQL for more details. If we choose NoSQL, we cannot store UserID in the URL table (as there are no foreign keys in NoSQL), for that we would need a third table which will store the mapping between URL and the user.

6. Basic System Design and Algorithm

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

a. Encoding actual URL

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

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

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

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

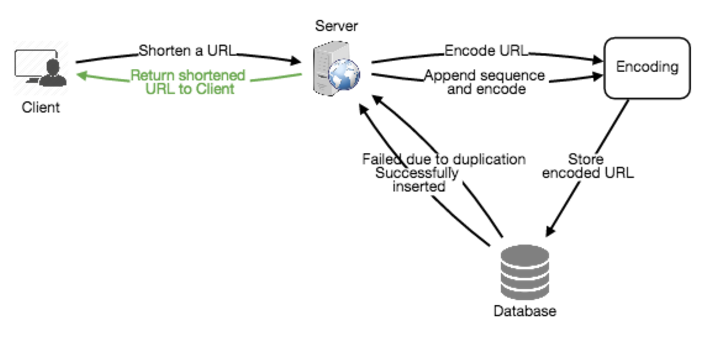
What are different issues with our solution? We have the following couple of problems with our encoding scheme:

If multiple users enter the same URL, they can get the same shortened URL, which is not acceptable.

What if parts of the URL are URL-encoded? e.g., http://www.educative.io/distributed.php?id=design, and http://www.educative.io/distributed.php%3Fid%3Ddesign are identical except for the URL encoding.

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

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



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 just take one of the already generated keys and use it. This approach will make things quite simple and fast since we will not be encoding the URL or worrying about duplications or collisions. KGS will make sure all the keys inserted in key-DB are unique.

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

Servers can use KGS to read/mark keys in the database. KGS can use two tables to store keys, one for keys that are not used yet and one for all the used keys. As soon as KGS gives keys to one of the servers, it can move them to the used keys table. KGS can always keep some keys in memory so that whenever a server needs them, it can quickly provide them. For simplicity, as soon as KGS loads some keys in memory, it can move them to used keys table. This way we can make sure each server gets unique keys. If KGS dies before assigning all the loaded keys to some server, we will be wasting those keys, which we can ignore given a huge number of keys we have. KGS also has to make sure not to give the same key to multiple servers. For that, it must synchronize (or get a lock to) the data structure holding the keys before removing keys from it and giving them to a server.

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

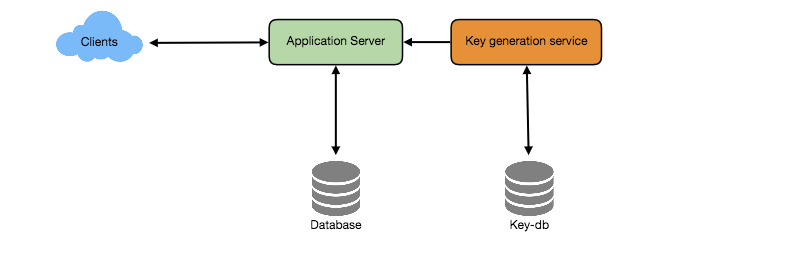
6 (characters per key) \* 68.7B (unique keys) => 412 GB.

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

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

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

Should we impose size limits on custom aliases? Since our service supports custom aliases, users can pick any ‘key’ they like, but providing a custom alias is not mandatory. However, it is reasonable (and often desirable) to impose a size limit on a custom alias, so that we have a consistent URL database. Let’s assume users can specify maximum 16 characters long customer key (as reflected in the above database schema).



7. Data Partitioning and Replication

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

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

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

b. Hash-Based Partitioning: b. Hash-Based Partitioning: In this scheme, we take a hash of the object we are storing, and based on this hash we figure out the DB partition to which this object should go. In our case, we can take the hash of the ‘key’ or the actual URL to determine the partition to store the file. Our hashing function will randomly distribute URLs into different partitions, e.g., our hashing function can always map any key to a number between [1…256], and this number would represent the partition to store our object.

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

8. Cache

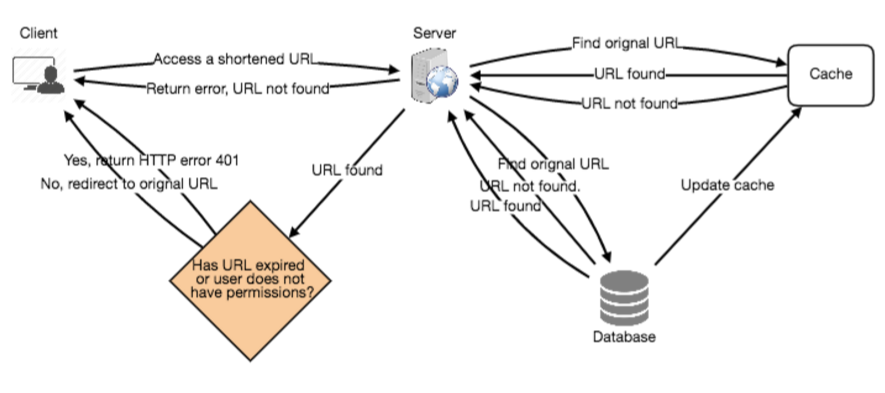
We can cache URLs that are frequently accessed. We can use some off-the-shelf solution like Memcache, that can store full URLs with their respective hashes. The application servers, before hitting backend storage, can quickly check if the cache has desired URL.

How much cache should we have? We can start with 20% of daily traffic and based on clients’ usage pattern we can adjust how many cache servers we need. As we estimated above we need 15GB memory to cache 20% of daily traffic that can easily fit into one server.

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

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

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



9. Load Balancer (LB)

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

Between Clients and Application servers

Between Application Servers and database servers

Between Application Servers and Cache servers

Initially, a simple Round Robin approach can be adopted; that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is if a server is dead, LB will take it out of the rotation and will stop sending any traffic to it. A problem with Round Robin LB is, it won’t take server load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries backend server about its load and adjusts traffic based on that.

10. Purging or DB cleanup

Should entries stick around forever or should they be purged? If a user-specified expiration time is reached, what should happen to the link? If we chose to actively search for expired links to remove them, it would put a lot of pressure on our database. We can slowly remove expired links and do a lazy cleanup too. Our service will make sure that only expired links will be deleted, although some expired links can live longer but will never be returned to users.

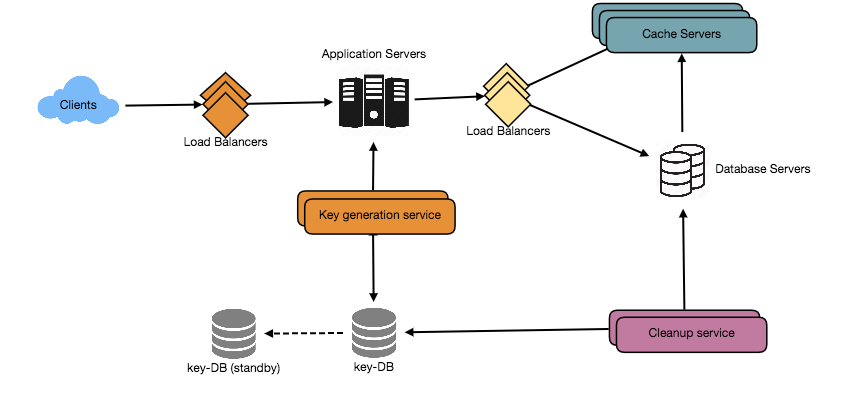
Whenever a user tries to access an expired link, we can delete the link and return an error to the user.

A separate Cleanup service can run periodically to remove expired links from our storage and cache. This service should be very lightweight and can be scheduled to run only when the user traffic is expected to be low.

We can have a default expiration time for each link, e.g., two years.

After removing an expired link, we can put the key back in the key-DB to be reused.

Should we remove links that haven’t been visited in some length of time, say six months? This could be tricky. Since storage is getting cheap, we can decide to keep links forever.



11. Telemetry

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

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

12. Security and Permissions

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

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