Capacity Estimation and Constraints

Let us 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 us 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 us 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 numbers are estimated with ignoring video compression and replication, which would change our estimates.

Bandwidth estimates: With 500 hours of video uploads per minute and assuming each video upload takes a bandwidth of 10MB/min, we would

be getting 300GB of uploads every minute.

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

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

System APIs

uploadVideo(api_dev_key, video_title, vide_description, tags[], cate gory_id, default_language, recording_details, video_contents)

searchVideo(api_dev_key, search_query, user_location, maximum_videos _to_return, page_token)comment video

streamVideo(api_dev_key, video_id, offset, codec, resolution

Database Design

User

Field	**Type**
Userid	string
Name	String
Email	String
Age	Number
Password	String
Last login	timestamp

Video

Field	**Type**
Video id	Integer
Description	String
Title	String
Size	Integer
Thumbnail	String
Total likes	Integer
Total dislikes	Integer
Total Views	integer
Created date	Timestamp

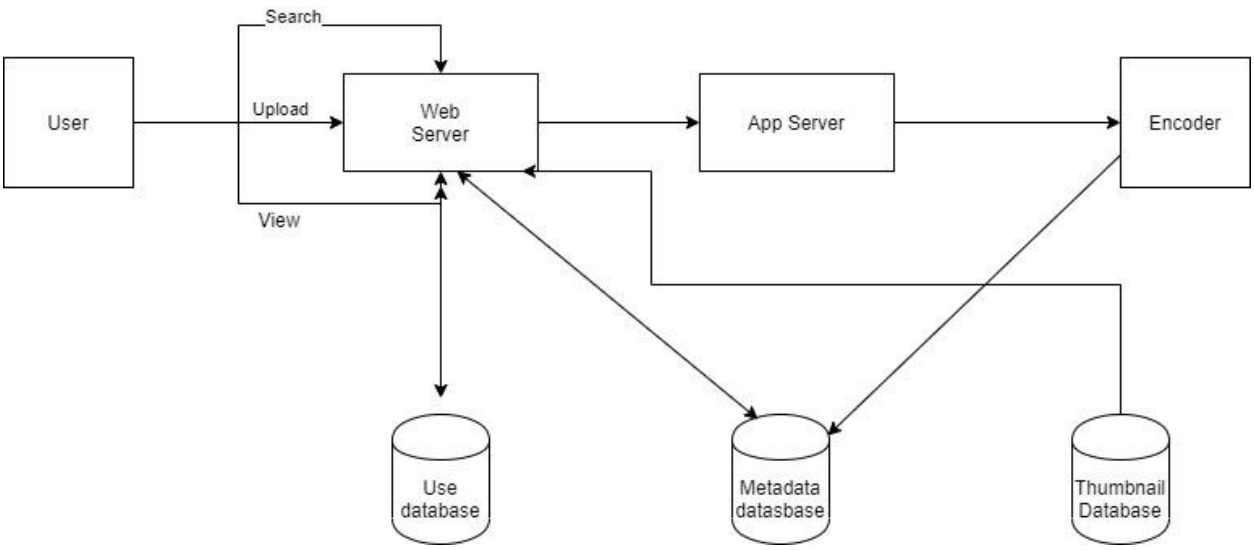
Video Comments

Field	Type
Video id	Integer
User id	Integer
Comment id	Integer
Comment	string
Created date	Timestamp

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.

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

Basic System Design and Algorithm



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 master-slave configurations where writes will go to master first

and then gets applied at all the slaves. Such configurations can cause some staleness in data, e.g., when a new video is added, its metadata would be inserted in the master first and before it gets applied at the slave our slaves 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.

Storing Thumbnail

There will be a lot more thumbnails than videos. If we assume that every video will have five thumbnails, we need to have a very efficient storage system that can serve a huge read traffic. There will be two consideration before deciding which storage system should be used for thumbnails:

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

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

Bigtable can be a reasonable choice here as it combines multiple files into one block to store on the disk and is very efficient in reading a small amount of data. Both are the two most significant requirements of our service. Keeping hot thumbnails in the cache will also help in improving the latencies and, given that thumbnails files are small, we can easily cache many 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 will be completed the uploader will be notified and the video is made available for view/sharing.

Data Partitioning and Replication

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 us go through different strategies of sharding this data one by one:

Sharding based on UserID: We can try storing all the data for a 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 must 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 a random server where we will store that Videos metadata. To find videos of a user we will query all servers and each server will return 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.

Caching

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 many geographically distributed video cache servers. We need to have a strategy that will maximize user performance and evenly distributes the load on its cache servers.

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

How can we build more intelligent cache? If we go with 80-20 rule, i.e., 20% of daily read volume for videos is generating 80% of traffic, meaning that certain videos are so popular that the majority of people view them; it follows that we can try caching 20% of daily read volume of videos and metadata

Video Deduplication

With a huge number of users uploading a massive amount of video data our service will have to deal with widespread video duplication. Duplicate videos often differ in aspect ratios or encodings, can contain overlays or additional borders, or can be excerpts from a longer original video. The proliferation of duplicate videos can have an impact on many levels:

1. Data Storage: We could be wasting storage space by keeping multiple copies of the same video
2. Caching: Duplicate videos would result in degraded cache efficiency by taking up space that could be used for unique content.
3. Network usage: 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 such as Block, Phase Correlation 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.

Load Balancing and Fault Tolerance

We should use Consistent Hashing among our cache servers, which will also help in balancing the load between cache servers. Since we will be using a static hash-based scheme to map videos to hostnames it can lead to an uneven load on the logical replicas due to the different popularity of each video. 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 cannot 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 datacenter) redirections lead a client to a distant cache location because the higher tier caches are only present at a small number of locations.

Content-Delivery Network

A CDN is a system of distributed servers that deliver web content to a user based in the geographic locations of the user, the origin of the web page and a content delivery server

Our service can move popular videos to CDNs:

CDNs replicate content in multiple places. There is 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.