Design Youtube

What is YouTube?

YouTube is one of the most popular video-sharing platforms that allows users to upload, watch, search, and share video-based content. It also provides features to like, dislike, add comments to videos, etc. Overall, YouTube is a huge system! But have you ever thought about how YouTube works and the underlying principles behind its functionalities? Let’s understand this in detail.

The Key Requirement of the System

In this blog, we will focus on creating the smallest version of Youtube with the following features:

Functional Requirements

- Users should be able to upload videos.
- Users should be able to view videos.
- Users should be able to change video quality.
- The system should keep the count of likes, dislikes, comments, and views.

Non-Functional Requirements

- Video uploading should be fast and users should have a smooth streaming experience.
- The system should be highly available, scalable, and reliable. We can compromise with consistency to offer high availability. It should be fine if a user sees a slight lag in the video data.
- The system should offer low latency and high throughput.
- The system should be cost-efficient.

Capacity estimation and System Constraints

The following estimates are based on many assumptions:

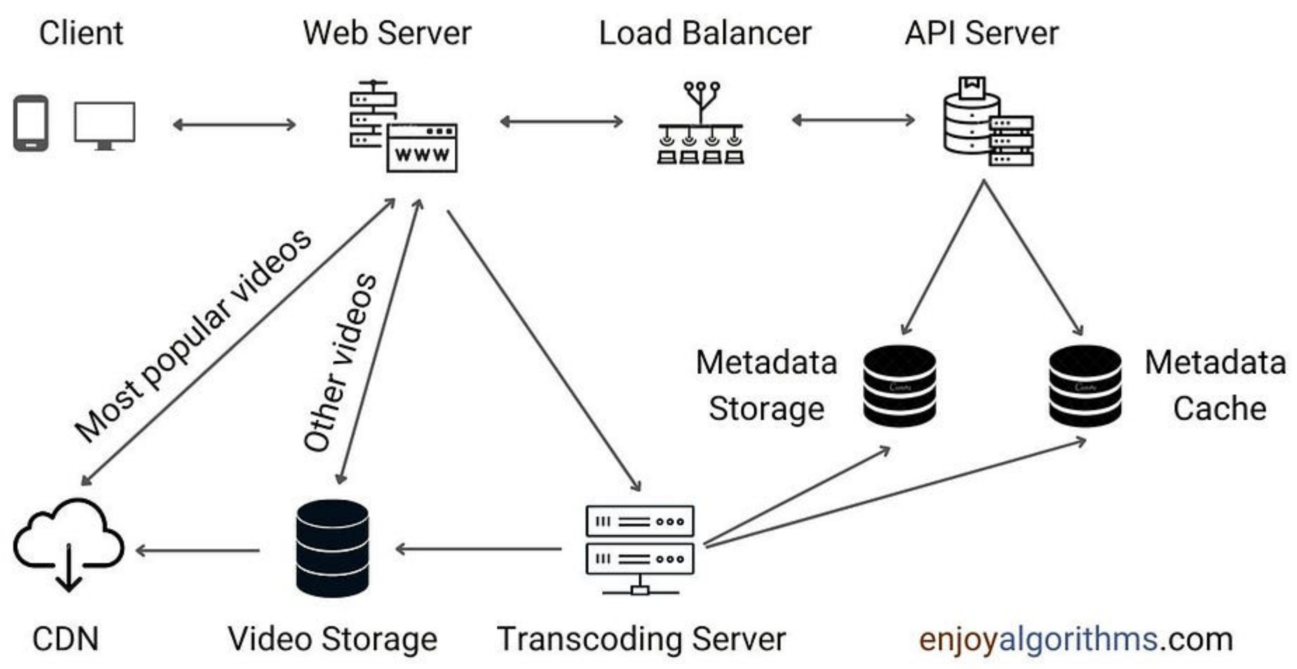
- Let’s assume total users = 2 billion.
- Daily active users = 400 million.
- The number of videos watched/day/user = 5.
- Total video views/day = 400 million * 5 = 2 billion views/day.

Youtube would be view-heavy (read-heavy) system. Suppose upload to view ratio (read-to-write ratio) is 1:100, then total video upload/day = 2 billion/100 = 20 million videos upload/day.

Suppose the average video size is 100 MB. Total storage needed/day = 20 million * 100 MB = 2000 TB/day = 2 PB/day. Here we ignore video compression and replication, which would change our estimates.

If we use existing CDN cloud services to serve videos, then it would cost money for data transfer. Suppose we use Amazon’s CDN CloudFront, which costs $0.01 per GB of data transfer. So the total cost for video streaming/day = Total video views/day * avg video size in GB * $0.01 = 2 billion * 0.1 * 0.01 = 2 million $/day. As we observe, serving videos from the CDN would cost lots of money.

High-Level Design



High-level design requires the following components:

Client: A computer, mobile phone, etc.

Video Storage: This is a BLOB storage system for storing transcoded videos. A Binary Large Object (BLOB) is a collection of binary data stored as a single entity in a database management system.

Transcoding Server: We use this for video transcoding or encoding purposes, which converts a video into multiple formats and resolutions (144p, 240p, 360p, 480p, 720p, 1080p & 4K) to provide the best video streaming for different devices and bandwidth requirements.

API server: We use this to handle all requests except video streaming. This includes feed recommendations, generating video upload URL, updating metadata database and cache, user signup, etc. This server talks to various databases to get all the data.

Web Server: This is used for handling incoming requests from the client. Based on the type of request, it routes the request to the application server or transcoding server.

CDN: Encoded videos are also stored in CDN for fast streaming purposes.

When we play a video, a popular video is streamed from the CDN.

Load Balancer: We use this to evenly distributes requests among API servers.

Metadata Storage: We use this to store all the video metadata like title, video URL, thumbnails, user information, view count, likes, comment, size, resolution, format, etc. It is sharded and replicated to meet performance and high availability requirements.

Metadata Cache: We use this to cache video metadata, user info, and thumbnails for better read performance.

As YouTube is a heavily loaded service, it has various APIs to perform operations smoothly. There could be different APIs to design features like video uploading, video streaming, video search, adding comments, videos, recommendations, and many more. We can use either **SOAP** or **REST** architecture to implement the system.

Video Uploading Process

We use the uploadVideo API for uploading the video content, which returns an HTTP response that demonstrates video is uploaded successfully or not.

string **uploadVideo**(string apiKey, stream videoData, string videoTitle, string videoDescription, string videoCategory, string videoTags[], string videoLanguage, string videoLocation)

- **apiKey:** An identification of the registered account.
- **videoData:** Uploaded video data.
- **videoTitle:** The title of the video.
- **videoDescription:** The description text of the video.
- **videoCategory:** Video category data like sports, education, etc.
- **videoTags[]:** A list of tags for the video.
- **videoLanguage:** The language of the content like English, Hindi, etc.
- **videoLocation:** The location where the video was recorded.

The video upload flow is divided into two processes running in parallel: 1) Uploading the video content and 2) Updating the video metadata.

1) Uploading the video content

First, videos are uploaded by the user, and then transcoding servers start the video encoding process. The encoding process converts video into multiple formats and resolutions set by the platform user. For increasing the throughput, we can parallelize the process by spreading this task across several machines. If a video will be popular, it can do another level of compression to ensure the same visual quality of the video at a much smaller size. Overall, video is processed by a batch job that runs several automated processes such as generating thumbnails, metadata, video transcripts, encoding, etc.

Video encoding is possible in two ways: **lossless** and **lossy**. In lossless encoding, there is no data loss between the original format to a new format. In the lossy encoding, some data is dropped from the original video to reduce the size of the new format. We might have experienced this when uploading a high-resolution image on a social network. After the upload, the image doesn’t look as good as the original image. Think!

After the completion of the encoding process, two things get executed in parallel: 1) Storing the encoded videos in a transcoded database and CDN and 2) Updating the metadata database and cache. Finally, API servers inform the client that the video is successfully uploaded and ready for streaming.

2) Updating the video metadata

While content is being uploaded to the video storage, the client, in parallel, sends a request to update the video metadata. This data includes title, video URL, thumbnails, user information, view count, likes, comment, size, resolution, etc.

Video Streaming Process

We use the uploadVideo API for uploading the video content, which continuously sends the small pieces of the video media stream from the given offset.

stream **viewVideo**(string apiKey, string videoId, int videoOffset, string codec, string videoResolution)

- **apiKey:** An identification of the registered account.
- **videoId:** An identifier for the video.
- **videoOffset:** This is the time from the start of the video, which enables users to watch a video on any device from the same point where they left off.
- **codec:** Codec is the video compression standard to compress large video content into smaller sizes. It uses efficient video compression algorithms to facilitate this process. We send the client's codec info in the API to support play/pause from multiple devices.
- **videoResolution:** We also send the client's resolution details because different devices may have different resolutions.

Whenever a user sends a request to watch the video, the platform checks the viewer's device type, screen size, processing capability, the network bandwidth. Based on that, it delivers the best video version from the nearest edge server in real-time. After this, our device immediately loads a little bit of data at a time and continuously receives video streams from CDN or Video Storage. Here are two critical observations:

- The CDN is a crucial component of our system. It ensures fast video delivery on a global scale. However, we know CDN is expensive, especially when a video size is large.
- There will be two types of video in our system: 1) Popular videos that are accessed frequently 2) Unpopular videos which have few or no viewers.

Based on the above observation, we can make cost-effective decisions and implement a few optimizations. For example, We can only stream the most popular videos from CDN directly. Other videos from the high-capacity video storage database. If a video will be popular later, it will move from video storage to CDN.

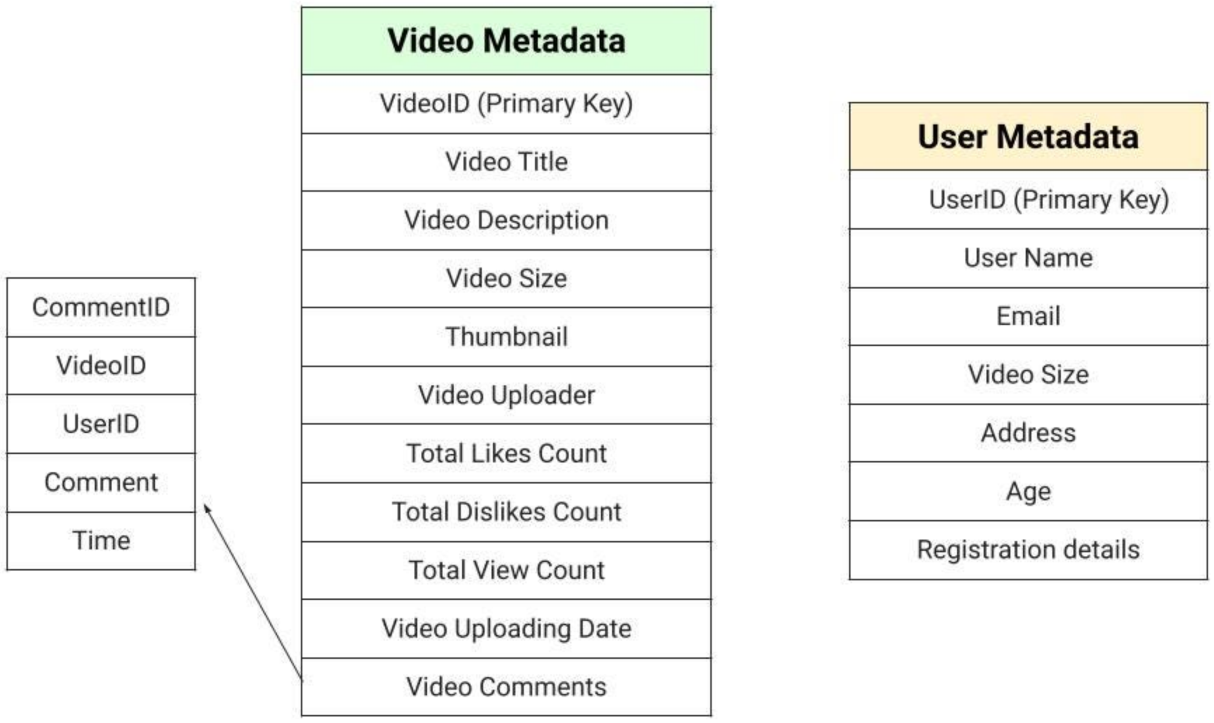
We need to use a standard streaming protocol to control data transfer during video streaming. Some popular streaming protocols are MPEG–DASH, Apple HLS, Microsoft Smooth Streaming, Adobe HTTP Dynamic Streaming (HDS), etc. The different streaming protocols support different video encodings and playback players, so we need to choose the proper streaming protocol.

Suppose we use Dynamic Adaptive Streaming (MPEG–DASH)protocol for video streaming, which can help us in two ways:

- It helps us to make content available to the users at different bit rates and reduce the buffering as much as possible.
- This helps us deliver quality content based on network bandwidth and device type of the end-user.

Metadata Schema

We can use MySQL to store metadata like user and video information. For this, we can maintain two separate tables: one table to store user metadata and another table to store video metadata.



Metadata Replication: Master-Slave Architecture

In this architecture, we need two data sources to scale out the application: one to handle the write query and the other one to handle the read query. Here writes request will go to the master first and then apply to all the slaves. Read requests will be routed to slave replicas parallelly to reduce the load on the master. This could help us to increase the read throughput.

Such a design may cause staleness in data from the read replica. How? Let's think! Suppose we performed a write operation by adding a new video, then its metadata would be first inserted in the master. Now before this new data gets updated to the slave, a new read request came. At this point, slaves would not be able to see it and return stale data to the user.

For example, this inconsistency may create a difference in view counts for a video between the master and the replica. But this can be okay for the user if there is a slight inconsistency (for a short duration) in the view count.

But here is a problem with this approach: Since we have a huge number of new videos every day and our read operation is extremely high, the master-slave architecture will suffer from replication lag. On another side, update operation causes cache misses, which go to disk where slow I/O causes slow replication. Now the critical question is: how can we improve the performance of the read/write operations? Think!

Metadata Sharding

Sharding is one of the ways of scaling a relational database besides the master-slave architecture. In this process, we distribute our data across multiple machines so that we can perform read/write operations efficiently. Now instead of a single master handling the write requests, it could be done on various sharded machines and increase the write performance. We can also create separate replicas to improve redundancy and throughput.

Sharding can increase the system complexity and we need an abstract system to handle the scalability and manageability challenges. This requirement led to the development of Vitess!

Vitess: A database clustering system for horizontal scaling of MySQL

Vitess is a database clustering system that runs on top of MySQL. It has several built-in features that allow us to scale horizontally similar to the NoSQL database.

Vitess Architecture. Source: https://vitess.io/