① Design Youtube .

What is YouTube?

YouTube is a popular video streaming service where users upload, stream, search, comment, share, and like or dislike videos. Large businesses as well as individuals maintain channels where they host their videos. YouTube allows free hosting of video content to be shared with users globally. YouTube is considered a primary source of entertainment, especially among young people, and as of 2022, it is listed as the second-most viewed website after Google by Wikipedia.

YouTube's popularity

Some of the reasons why YouTube has gained popularity over the years include the following reasons:

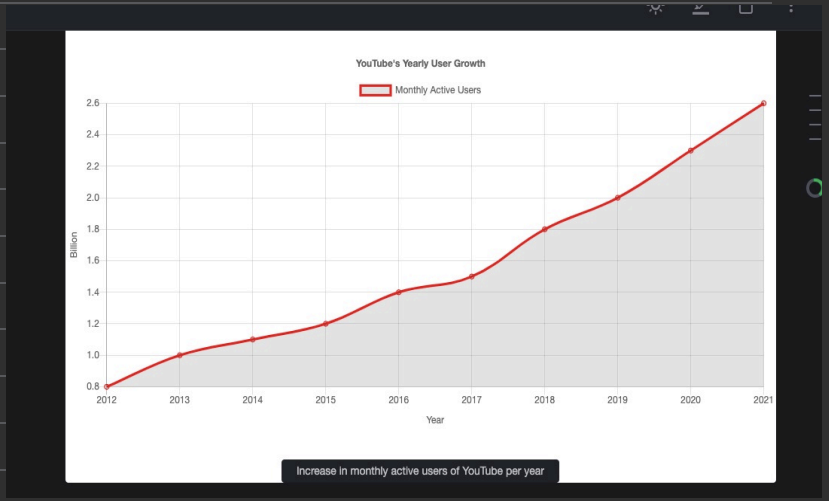
Simplicity: YouTube has a simple yet powerful interface.
Rich content: Its simple interface and free hosting has attracted a large number of content creators.
Continuous improvement: The development team of YouTube has worked relentlessly to meet scalability requirements over time. Additionally, Google acquired YouTube in 2007, which has added to its credibility.
Source of income: YouTube coined a partnership program where it allowed content creators to earn money with their viral content.

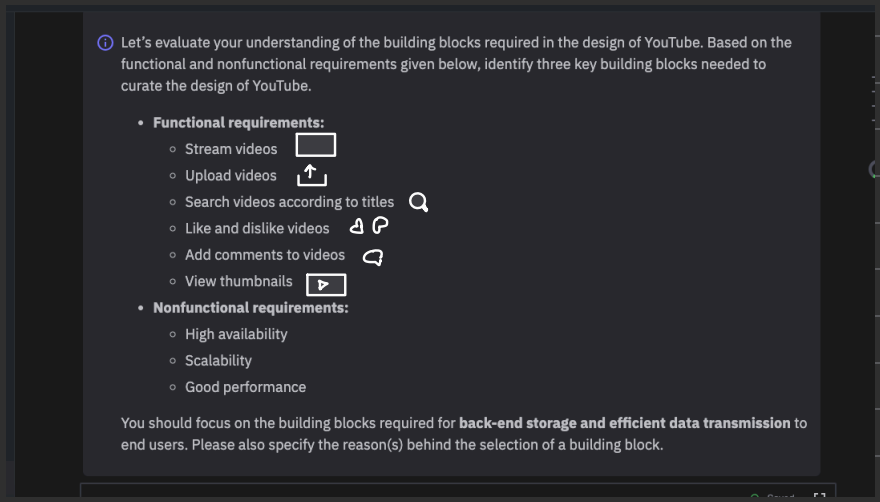
Apart from the above reasons, YouTube also partnered up with well-established market giants like CNN and NBC to set a stronger foothold.

YouTube's growth

Let's look at some interesting statistics about YouTube and its popularity.

Since its inception in February 2005, the number of YouTube users has multiplied. As of now, there are more than 2.5 billion monthly active users of YouTube. Let's take a look at the chart below to see how YouTube users have increased in the last decade.





List 3 building Blocks:

- Database (NoSQL) - vido, thumbnail, users, comments.
- CDN (viral content) - reduce latency
- BLOB: store video files

How will we design YouTube?

Requirements: This is where we identify the functional and non-functional requirements. We also estimate the resources required to serve millions of users each day. This lesson answers questions like how much storage space YouTube will need to store 500 hours of video content uploaded to YouTube per minute.

Design: In this lesson, we explain how we'll design the YouTube service. We also briefly explain the API design and database schemas. Lastly, we will also briefly go over how YouTube's search works.

Evaluation: This lesson explains how YouTube is able to fulfill all the requirements through the proposed design. It also looks at how scaling in the future can affect the system and what solutions are required to deal with scaling problems.

Reality is more complicated: During this lesson, we'll explore different techniques that YouTube employs to deliver content effectively to the client and avoid network congestions.

Quiz: We reinforce major concepts we learned designing YouTube by considering how we could design Netflix's system.Our discussion on the usage of various building blocks in the design will be limited since we've already explored them in detail in the building blocks chapter.

Functional requirements#

We require that our system is able to perform the following functions:

Stream videos
Upload videos
Search videos according to titles
Like and dislike videos
Add comments to videos
View thumbnails

Non-functional requirements (ARMS)

It's important that our system also meets the following requirements:

- **High availability**: The system should be highly available. High availability requires a good percentage of uptime. Generally, an uptime of 99% and above is considered good.
- **Scalability**: As the number of users grows, these issues should not become bottlenecks: storage for uploading content, the bandwidth required for simultaneous viewing, and the number of concurrent user requests should not overwhelm our application/web server.
- **Good performance**: A smooth streaming experience leads to better performance overall.
- **Reliability**: Content uploaded to the system should not be lost or damaged.

Resource Estimation:

- storage: uploaded content
- concurrent processing: IIRd
- upload / dwnld bandwidth

Actual numbers:

- Total users = 1.5 billion
- Active daily = 500 million
- Avg length of video = 5 min
- size of avg vid = 600MB
  after encoding = 30 MB

ESTIMATION:

- Storage: · Total num of video
  · Len of each video ↑ Uploaded / min

eg: 500 hours uploaded per min

30 MB for 5 min ⇒ 6 MB for 1 min vid

Total storage: Total storage req

Total upload/min: content ↑ per min

eg: 500 hours of content in 1 min

Storage min: storage req for each min (6MB)

$$\text{Total}_{\text{storage}} = \text{Total}_{\text{upload/min}} \times \text{Storage}_{\text{min}}$$

$$= 500\,\text{hr} \times 60\,\frac{\text{min}}{\text{hr}} \times \frac{6\,\text{MB}}{\text{min}}$$

$$= 180{,}000\ \text{MB}$$

$$\approx 180\ \text{GB}$$

Storage per min		min/hr		hr/day		Days/yr		Storage/yr
180 GB / min	×	60 min / hr	×	24 hr / day	×	365 day / yr	=	94.6 PB / yr

- **Bandwidth:**

upload : view ratio = 1 : 300

Total $_{bw}$: Total bw reqd

Total $_{content}$: content uploaded per min

Size $_{min}$: Transmission req

$$\text{Total}_{bw} = \text{Total}_{content} \times \text{Size}_{min}$$

$$= 500 \text{ hr} \times 60 \frac{\text{min}}{\text{hr}} \times 120 \frac{\text{MB}}{\text{min}}$$

$$= 480 \frac{\text{GB}}{\text{s}}$$

1. If 480 Gbps of bandwidth is required to satisfy uploading needs, how much bandwidth would be required to stream videos? Assume each minute of video requires 10 MB of bandwidth on average.

   Hint: The `upload:view` ratio is provided.

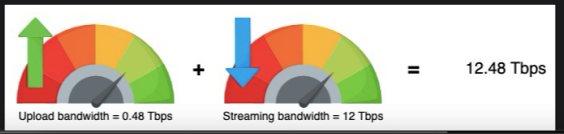
   Hide Answer

   For every video uploaded, 300 videos are watched. Therefore, the equation becomes this:

   $$\textit{total number of viewing hours per minutes} \times \textit{size in MB of each minute} \times \textit{view ratio} = 500 \frac{\textit{hours}}{\textit{minute}} \times 60 \frac{\textit{minutes}}{\textit{hour}} \times 10 \frac{\textit{MB}}{\textit{minute}} \times 300 = 90 \frac{\textit{TB}}{\textit{minute}} \times 8 \textit{ bits} = 720 \frac{\textit{Tb}}{\textit{minute}} = 12 \textit{ Tbps}$$

   Q1 / Q1

   Did you find this helpful?



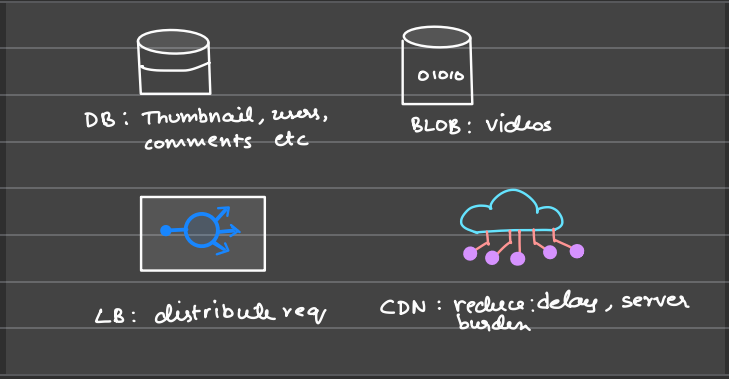
- **Number of servers:** 500 mil daily users.

500 mil req/sec.

$$\text{servers needed at peak} = \frac{\text{Num of Req / sec}}{\text{RPS of server}}$$

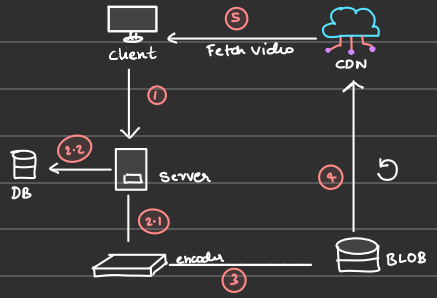
$$= \frac{500 \text{ mil}}{64{,}000} = 781.5 \approx 8\text{k servers}$$

Building Blocks used :



- servers
- encoders : compress video

DESIGN



The workflow for the abstract design is provided below:

- The user uploads a video to the server.
- The server stores the metadata and the accompanying user data to the database and, at the same time, hands over the video to the encoder for encoding (see 2.1 and 2.2 in the illustration above).
- The encoder, along with the transcoder, compresses the video and transforms it into multiple resolutions (like 2160p, 1440p, 1080p, and so on). The videos are stored on blob storage (similar to GFS or S3).
- Some popular videos may be forwarded to the CDN, which acts as a cache.
- The CDN, because of its vicinity to the user, lets the user stream the video with low latency. However, CDN is not the only infrastructure for serving videos to the end user, which we will see in the detailed design.

1. Why don't we upload the video directly to the encoder instead of to the server? Doesn't the current strategy introduce an additional delay?

Hide Answer

There are several reasons why it's a good idea to introduce a server in between the encoder and the client:

- The client could be malicious and could abuse the encoder.
- If the uploaded video is a duplicate, the server could filter it out.
- Encoders will be available on a private IP address within YouTube's network and not available for public access.

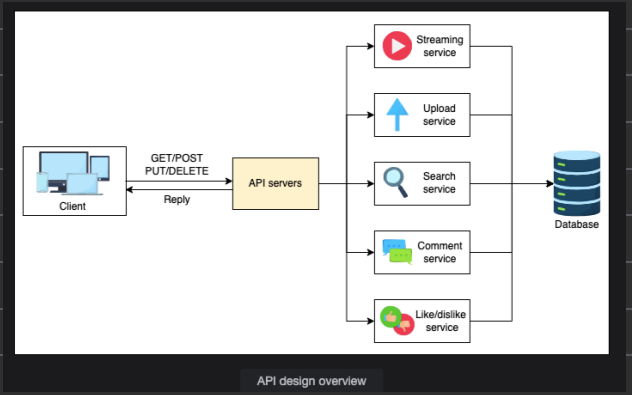
API Design:

- Feature set —API translate→ Tech specifications
- REST API :

A **REST API** (Representational State Transfer Application Programming Interface) is a set of rules and conventions for building and interacting with web services. It allows different software systems to communicate over the internet using standard HTTP methods.

Request: GET https://api.ex.com/prod/42

Response :
```
{
    id : 42,
    name : "Wireless mouse",
    price : 29.99
}
```



API design overview

Core Concepts of REST API:

1. Resources:
   Everything is treated as a resource, usually represented by a URL.
   Example: https://api.example.com/users/123 represents a user with ID 123.
2. HTTP Methods:
   GET – Retrieve data (e.g., get user info)
   POST – Create a new resource (e.g., create a new user)
   PUT – Update a resource completely
   PATCH – Update a resource partially
   DELETE – Delete a resource
3. Stateless:
   Each request contains all the information needed; the server doesn't store session state between requests.
4. Uses standard HTTP response codes:
   200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server Error, etc.

Upload videos
Stream videos
Search videos
View thumbnails
Like or dislike videos
Comment on videos

UPLOD VIDEO :

POST method : /uploadVideo API :

uploadVideo (user_id, ... privacy_settings)

Param	Desc
• user_id	user uploading
• video_file	BLOB
• category_id	* engineering * makeup
• title	title
• discription	desc
• dypault_language	
• privacy_settings	public , private

STREAM :

• Screen_res	user's screen res
• user bitrate	transmission capacity
• device_chipset	handheld

SearchVideo

• search_string	query
• length	len of time
• quality	res
• date	new

ViewThumbnail

video_id	user_id

Like and DisLike

video_id	user_id	like

comment

video_id	user_id	comment_text

Storage schema

Each of the above features in the API design requires support from the database—we'll need to store the details above in our storage schema to provide services to the API gateway.

User	
id:	INT
user_email:	VARCHAR
username:	VARCHAR
password:	VARCHAR
DOB:	DATE

Video	
id:	INT
title:	VARCHAR(256)
desc:	VARCHAR
upload_date:	DATE
channel_id:	INT
likes_count:	INT
dislikes_count:	INT
views_count:	INT
video_URI:	VARCHAR
privacy_level:	SMALLINT
default_lang:	VARCHAR

Comments	
id:	INT
video_id:	INT
user_id:	INT
date_posted:	DATE
comment_text:	VARCHAR(2048)
likes_count:	INT
dislikes_count:	INT

Channel	
id:	INT
channel_name:	VARCHAR
user_id:	INT
subscribers:	INT
description:	VARCHAR
category_id:	INT

Storage schema

Detailed design

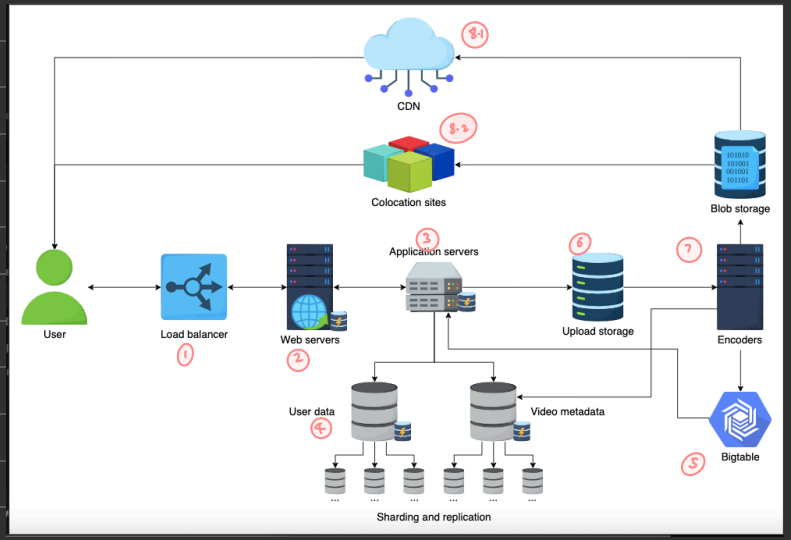
Now, let's get back to our high-level design and see if we can further explore parts of the design. In particular, the following areas require more discussion:

Component integration: We'll cover some interconnections between the servers and storage components to better understand how the system will work.

Thumbnails: It's important for users to see some parts of the video through thumbnails. Therefore, we'll add thumbnail generation and storage to the detailed design.

Database structure: Our estimation showed that we require massive storage space. We also require storing varying types of data, such as videos, video metadata, and thumbnails, each of which demands specialized data storage for performance reasons. Understanding the database details will enable us to design a system with the least possible lag.

Let's take a look at the diagram below. We'll explain our design in two steps, where the first looks at what the newly added components are, and the second considers how they coordinate to build the YouTube system.



Detailed design components

Since we highlighted the requirements of smooth streaming, server-level details, and thumbnail features, the following design will meet our expectations. Let's explain the purpose of each added component here:

- **Load balancers**: To divide a large number of user requests among the web servers, we require load balancers.
- **Web servers**: Web servers take in user requests and respond to them. These can be considered the interface to our API servers that entertain user requests.
- **Application server**: The application and business logic resides in application servers. They prepare the data needed by the web servers to handle the end users' queries.

- **User and metadata storage**: Since we have a large number of users and videos, the storage required to hold the metadata of videos and the content related to users must be stored in different storage clusters. This is because a large amount of not-so-related data should be decoupled for scalability purposes.
- **Bigtable**: For each video, we'll require multiple thumbnails. Bigtable is a good choice for storing thumbnails because of its high throughput and scalability for storing key-value data. Bigtable is optimal for storing a large number of data items each below 10 MB. Therefore, it is the ideal choice for YouTube's thumbnails.
- **Upload storage**: The upload storage is temporary storage that can store user-uploaded videos.
- **Encoders**: Each uploaded video requires compression and transcoding into various formats. Thumbnail generation service is also obtained from the encoders.
- **CDN and colocation sites**: CDNs and colocation sites store popular and moderately popular content that is closer to the user for easy access. Colocation centers are used where it's not possible to invest in a data center facility due to business reasons.

Design flow and technology usage

Now that we understand the purpose of every component, let's discuss the flow and technology used in different components in the following steps:

1. The user can upload a video by connecting to the web servers. The web server can run Apache or Lighttpd. Lighttpd is preferable because it can serve static pages and videos due to its fast speed.
2. Requests from the web servers are passed onto application servers that can contact various data stores to read or write user, videos, or videos' metadata. There are separate web and application servers because we want to decouple clients' services from the application and business logic. Different programming languages can be used on this layer to perform different tasks efficiently. For example, the C programming language can be used for encryption. Moreover, this gives us an additional layer of caching, where the most requested objects are stored on the application server while the most frequently requested pages will be stored on the web servers.
3. Multiple storage units are used. Let's go through each of these:
   I. Upload storage is used to store user-uploaded videos temporarily before they are encoded.
   II. User account data is stored in a separate database, whereas videos metadata is stored separately. The idea is to separate the more frequently and less frequently accessed storage clusters from each other for optimal access time. We can use MySQL if there are a limited number of concurrent reads and writes. However, as the number of users—and therefore the number of concurrent reads and writes—grows, we can move towards NoSQL types of data management systems.
   III. Since Bigtable is based on Google File System (GFS), it is designed to store a large number of small files with low retrieval latency. It is a reasonable choice for storing thumbnails.
4. The encoders generate thumbnails and also store additional metadata related to videos in the metadata database. It will also provide popular and moderately popular content to CDNs and colocation servers, respectively.
5. The user can finally stream videos from any available site.

YouTube search

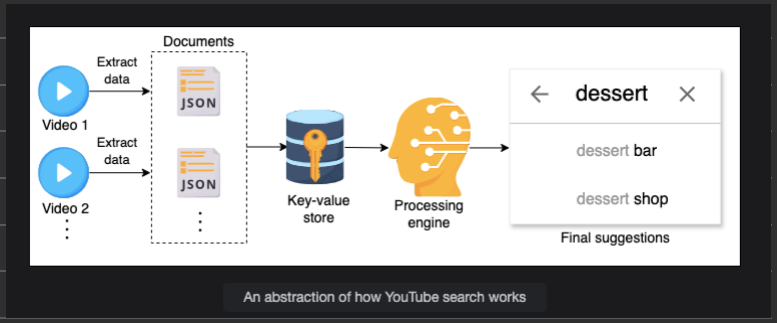
Since YouTube is one of the most visited websites, a large number of users will be using the search feature. Even though we have covered a building block on distributed search, we'll provide a basic overview of how search inside the YouTube system will work.

Each new video uploaded to YouTube will be processed for data extraction. We can use a JSON file to store extracted data, which includes the following:

Title of the video.
Channel name.
Description of the video.
The content of the video, possibly extracted from the transcripts.
Video length.
Categories.

Each of the JSON files can be referred to as a document. Next, keywords will be extracted from the documents and stored in a key-value store. The *key* in the key-value store will hold all the keywords searched by the users, while the *value* in the key-value store will contain the occurrence of each key, its frequency, and the location of the occurrence in the different documents. When a user searches for a keyword, the videos with the most relevant keywords will be returned.

The approach above is simplistic, and the relevance of keywords is not the only factor affecting search in YouTube. In reality, a number of other factors will matter. The processing engine will improve the search results by filtering and ranking videos. It will make use of other factors like view count, the watch time of videos, and the context, along with the history of the user, to improve search results.



An abstraction of how YouTube search works

EVALUATION

Fulfilling requirements

Our proposed design needs to fulfill the requirements we mentioned in the previous lessons. Our main requirements are smooth streaming (low latency), availability, and reliability. Let's discuss them one by one.

1. **Low latency/Smooth streaming** can be achieved through these strategies:
   - Geographically distributed cache servers at the ISP level to keep the most viewed content.
   - Choosing appropriate storage systems for different types of data. For example, we'll can use Bigtable for thumbnails, blob storage for videos, and so on.
   - Using caching at various layers via a distributed cache management system.
   - Utilizing content delivery networks (CDNs) that make heavy use of caching and mostly serve videos out of memory. A CDN deploys its services in close vicinity to the end users for low-latency services.

2. **Scalability**: We've taken various steps to ensure scalability in our design as depicted in the table below. The horizontal scalability of web and application servers will not be a problem as the users grow. However, MySQL storage cannot scale beyond a certain point. As we'll see in the coming sections, that may require some restructuring.
3. **Availability**: The system can be made available through redundancy by replicating data to as many servers as possible to avoid a single point of failure. Replicating data across data centers will ensure high availability, even if an entire data center fails because of power or network issues. Furthermore, local load balancers can exclude any dead servers, and global load balancers can steer traffic to a different region if the need arises.
4. **Reliability**: YouTube's system can be made reliable by using data partitioning and fault-tolerance techniques. Through data partitioning, the non-availability of one type of data will not affect others. We can use redundant hardware and software components for fault tolerance. Furthermore, we can use the heartbeat protocol to monitor the health of servers and omit servers that are faulty and erroneous. We can use a variant of consistent hashing to add or remove servers seamlessly and reduce the burden on specific servers in case of non-uniform load.

1. Isn't the load balancer a single point of failure (SPOF)?
Just like with servers, we can use multiple load balancers. Users can be randomly forwarded to different load balancers from the Domain Name System (DNS).

Requirements	Techniques
Scalability	• Load balancers to multiplex between servers. • Ability to horizontally add (or remove) web servers according to our current needs. • Addition of multiple storage units specific to the required types of data. • Serving from different colocation sites and CDNs. • Separating read/write operations on different servers.
Availability	• Replication of content on different sites. • Important data is persisted on replicated data stores so that we could re-spawn the service in case of major disruption or failures. • Using local and global load balancers.
Performance	• Lighttpd for serving videos/static content. • Caching at each layer (file system, database, cluster, application server, web server). • Addition of multiple storage units specific to the required types of data. • CDNs. • Using an appropriate programming language to perform specific tasks—for example, using C for encryption and Python otherwise.

How YouTube achieves scalability, availability, and good performance

Consistency : TRADE OFFS !

Our solution prefers high availability and low latency. However, strong consistency can take a hit because of high availability (see the CAP theorem). Nonetheless, for a system like YouTube, we can afford to let go of strong consistency. This is because we don't need to show a consistent feed to all the users. For example, different users subscribed to the same channel may not see a newly uploaded video at the same time. It's important to mention that we'll maintain strong consistency of user data. This is another reason why we've decoupled user data from video metadata.

Distributed cache

We prefer a distributed cache over a centralized cache in our YouTube design. This is because the factors of scalability, availability, and fault-tolerance, which are needed to run YouTube, require a cache that is not a single point of failure. This is why we use a distributed cache. Since YouTube mostly serves static content (thumbnails and videos), Memcached is a good choice because it is open source and uses the popular Least Recently Used (LRU) algorithm. Since YouTube video access patterns are long-tailed, LRU-like algorithms are suitable for such data sets.

Bigtable versus MySQL

Another interesting aspect of our design is the use of different storage technologies for different data sets. Why did we choose MySQL and Bigtable?

The primary reason for the choice is performance and flexibility. The number of users in YouTube may not scale as much as the number of videos and thumbnails do. Moreover, we require storing the user and metadata in structured form for convenient searching. Therefore, MySQL is a suitable choice for such cases.

However, the number of videos uploaded and the thumbnails for each video would be very large in number. Scalability needs would force us to use a custom or NoSQL type of design for that storage. One could use alternatives to GFS and Bigtable, such as HDFS and Cassandra.

Public versus private CDN

Our design relies on CDNs for low latency serving of the content. However, CDNs can be private or public. YouTube can choose between any one of the two options.

This choice is more of a cost issue than a design issue. However, for areas where there is little traffic, YouTube can use the public CDN because of the following reasons:

Setting up a private CDN will require a lot of CAPEX.
For rather little viral traffic in certain regions, there will not be enough time to set up a new CDN.
There may not be enough users to sustain the business.
However, YouTube can consider building its own CDN if the number of users is too high, since public CDNs can prove to be expensive if the traffic is high. Private CDNs can also be optimized for internal usage to better serve customers.

Public versus private CDN

Our design relies on CDNs for low latency serving of the content. However, CDNs can be private or public. YouTube can choose between any one of the two options.

This choice is more of a cost issue than a design issue. However, for areas where there is little traffic, YouTube can use the public CDN because of the following reasons:

Setting up a private CDN will require a lot of CAPEX.
For rather little viral traffic in certain regions, there will not be enough time to set up a new CDN.
There may not be enough users to sustain the business.
However, YouTube can consider building its own CDN if the number of users is too high, since public CDNs can prove to be expensive if the traffic is high. Private CDNs can also be optimized for internal usage to better serve customers.

Duplicate videos

The current YouTube design doesn't handle duplicate videos that have been uploaded by a user or spammers. Duplicated videos take extra space, which leads to a trade-off. As a result, we either waste storage space or face an additional complexity to the upload process for handling duplicate videos.

Let's perform some calculations to resolve this problem. Assume that 50 out of 500 hours of videos uploaded to YouTube are duplicates. Considering that one minute of video requires 6 MB of storage space, the duplicated content will take up the following storage space:

$$(50 \times 60)\ \text{min} \times 60\ \frac{\text{MB}}{\text{min}} = 18\ \text{GB}$$

If we avoid video duplication, we can save up to 9.5 petabytes of storage space in a year. The calculations are as follows:

$$18\ GB/min \times (60 \times 24 \times 365) = 9.5\ PB$$

Storage space being wasted, and other computational costs are not the only issues with duplicate videos. An important aspect of duplicate videos is the copyright issue. No content creator would want their content plagiarized. Therefore, it's plausible to add the complexity of handling duplicate videos to the design of YouTube.

Duplication can be solved with simple techniques like locality-sensitive hashing. However, there can be complex techniques like Block Matching Algorithms (BMAs) and phase correlation to find duplications. Implementing this solution can be quite complex in a huge database of videos. We may have to use technologies like artificial intelligence (AI).

Future scaling

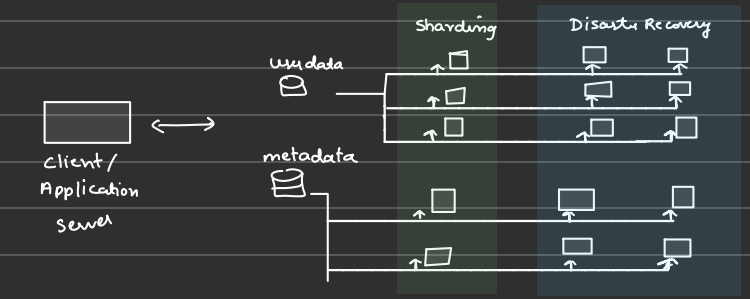
So far, we've focused on the design and analysis of the proposed design for YouTube. In reality, the design of YouTube is quite complex and requires advanced systems. In this section, we'll focus on the pragmatic structure of data stores and the web server.

We'll begin our discussion with some limitations in terms of scaling YouTube. In particular, we'll consider what design changes we'll have to make if the traffic load to our service goes up by, say, a few folds.

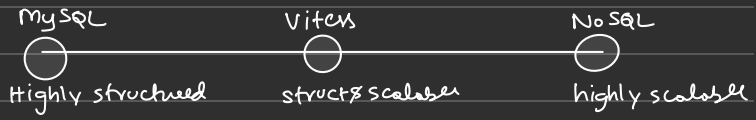
We already know that we'll have to scale our existing infrastructure, which includes the below elements:

Web servers
Application servers
Datastores
Placing load balancers among each of the layers above
Implementing distributed caches

Any infrastructure mentioned above requires some modifications and adaptation to the application-level logic. For example, if we continue to increase our data in MySQL servers, it can become a choke point. To effectively use a sharded database, we might have to make changes to our database client to achieve a good level of performance and maintain the ACID (atomicity, consistency, isolation, durability) properties. However, even if we continue to change to the database client as we scale, its complexity may reach a point where it is no longer manageable. Also note that we haven't incorporated a disaster recovery mechanism into our design yet.
To resolve the problems above, YouTube has developed a solution called Vitess.



The key idea in Vitess is to put an abstraction on top of all the database layers, giving the database client the illusion that it is talking to a single database server. The single database in this case is the Vitess system. Therefore, all the database-client complexity is migrated to and handled by Vitess. This maintains the ACID properties because the internal database in use is MySQL. However, we can enable scaling through partitioning. Consequently, we'll get a MySQL structured database that gives the performance of a NoSQL storage system. At the same time, we won't have to live with a rich database client (application logic). The following illustration highlights how Vitess is able to achieve both scalability and structure.



One could imagine using techniques like data denormalization instead of the Vitess system. However, data denormalization won't work because it comes at the cost of reduced writing performance. Even if our work is read-intensive, as the system scales, writing performance will degrade to an unbearable limit.

Web server

A **web server** is an extremely important component and, with scale, a custom web server can be a viable solution. This is because most commercial or open-source solutions are general purpose and are developed with a wide range of users in mind. Therefore, a custom solution for such a successful service is desirable.

Let's try an interesting exercise to see which server YouTube currently uses. Click on the terminal below and execute the following command:

```
lynx -head -dump http://www.youtube.com | grep ^Server
```

The Reality Is More Complicated

Learn how YouTube can use different techniques to effectively deliver content to the end user.

Introduction

Now that we've understood the design well, let's see how YouTube can optimize the usage of storage and network demands while maintaining good quality of experience (QoE) for the end user.

When talking about providing effective service to end users, the following three steps are important:

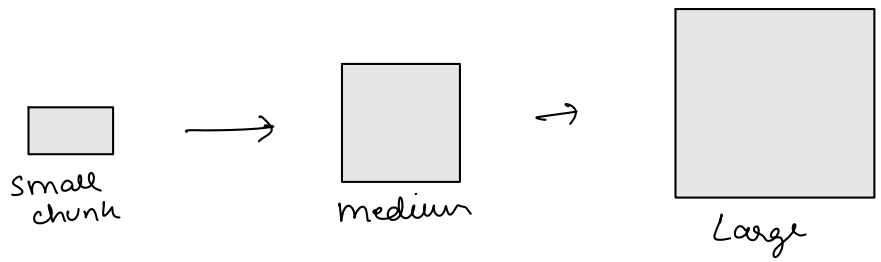
- **Encode**: The raw videos uploaded to YouTube have significant storage requirements. It's possible to use various encoding schemes to reduce the size of these raw video files. Apart from compression, the choice of encoding scheme will also depend on the types of end devices used to stream the video content. Since multiple devices could be used to stream the same video, we may have to encode the same video using different encoding schemes resulting in one raw video file being converted into multiple files each encoded differently. This strategy will result in a good user-perceived experience because of two reasons: users will save bandwidth because the video file will be encoded and compressed to some limit, and the encoded video file will be appropriate for the client for a smooth playback experience.
- **Deploy**: For low latency, content must be intelligently deployed so that it is closer to a large number of end users. Not only will this reduce latency, but it will also put less burden on the networks as well as YouTube's core servers.
- **Deliver**: Delivering to the client requires knowledge about the client or device used for playing the video. This knowledge will help in adapting to the client and network conditions. Therefore, we'll enable ourselves to serve content efficiently.

Let's understand each phase in detail now.

Encode

Until now, we've considered encoding one video with different encoding schemes. However, if we encode videos on a per-shot basis, we'll divide the video into smaller time frames and encode them individually. We can divide videos into shorter time frames and refer to them as segments. Each segment will be encoded using multiple encoding schemes to generate different files called chunks. The choice of encoding scheme for a segment will be based on the detail within the segment to get optimized quality with lesser storage requirements. Eventually, each shot will be encoded into multiple chunk sizes depending on the segment's content and encoding scheme used. As we divide the raw video into segments, we'll see its advantages during the deployment and delivery phase.

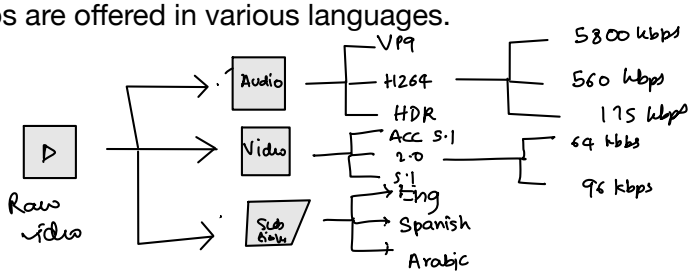
Let's understand how the per-segment encoding will work. For any video with dynamic colors and high depth, we'll encode it differently from a video with fewer colors. This means that a not-so-dynamic segment will be encoded such that it's compressed more to save additional storage space. Eventually, we'll have to transfer smaller file sizes and save bandwidth during the deployment and streaming phases.



Higher quality requiring higher bitrate from left to right

Using the strategy above, we'll have to encode individual shots of a video in various formats. However, the alternative to this would be storing an entire video (using no segmenting) after encoding it in various formats. If we encode on a per-shot basis, we would be able to optimally

reduce the size of the entire video by doing the encoding on a granular level. We can also encode audio in various formats to optimally allow streaming for various clients like TVs, mobile phones, and desktop machines. Specifically, for services like Netflix, audio encoding is more useful because audios are offered in various languages.



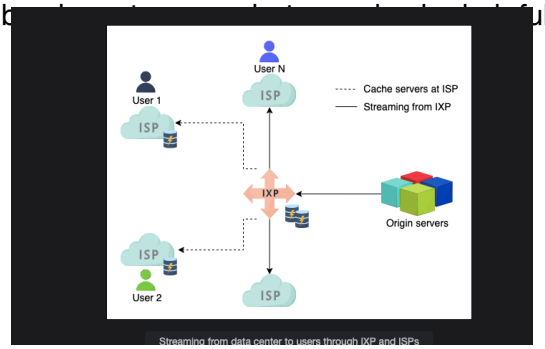
A raw video file being encoded into different formats including its audio and subtitles

Deploy

As discussed in our design and evaluation sections, we have to bring the content closer to the user. This has three main advantages:

1. Users will be able to stream videos quickly.
2. There will be a reduced burden on the origin servers.
3. Internet service providers (ISPs) will have spare bandwidth.

So, instead of streaming from our data centers directly, we can deploy chunks of popular videos in CDNs and point of presence (PoPs) of ISPs. In places where there is no collaboration possible with the ISPs, our content can be placed in internet exchange point (IXPs). We can put content in IXPs that will not only b[illegible]ul in filling the cache of ISP PoPs.



Streaming from data center to users through IXP and ISPs

We should keep in mind that the caching at the ISP or IXP is performed only for the popular content or moderately popular content because of limited storage capacity. Since our per-shot encoding scheme saves storage space, we'll be able to serve out more content using the cache infrastructure closer to end users.

Additionally, we can have two types of storage at the origin servers:

1. **Flash servers**: These servers hold popular and moderately popular content. They are optimized for low-latency delivery.
2. **Storage servers**: This type holds a large portion of videos that are not popular. These servers are optimized to hold large storage.

Note: When we transfer streaming content, it can result in the congestion of networks. That is why we have to transfer the content to ISPs in off-peak hours.

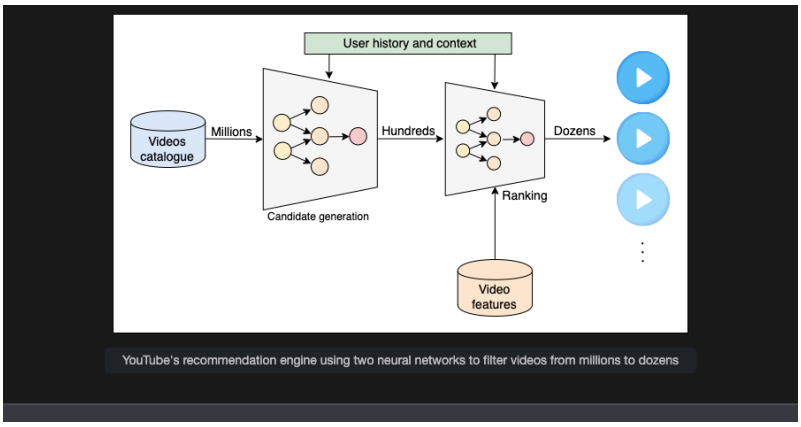
Recommendations

YouTube recommends videos to users based on their profile, taking into account factors such as their interests, view and search history, subscribed channels, related topics to already viewed content, and activities on content such as comments and likes.

An approximation of the recommendation engine of YouTube is provided below. YouTube filters videos in two phases:

1. **Candidate generation**: During this phase, millions of YouTube videos are filtered down to hundreds based on the user's history and current context.
2. **Ranking**: The ranking phase rates videos based on their features and according to the user's interests and history. Hundreds of videos are filtered and ranked down to a few dozen videos during this phase.

YouTube employs machine learning technology in both phases to provide recommendations to users.

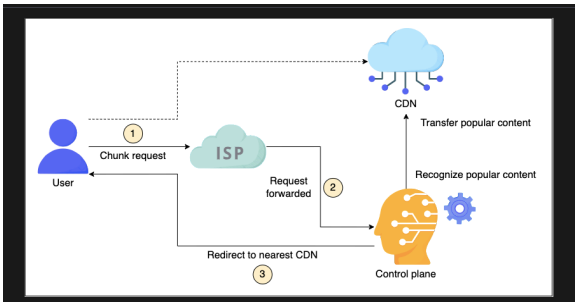


YouTube's recommendation engine using two neural networks to filter videos from millions to dozens
Points to Ponder
1.
Previously, we said that popular content is sent to ISPs, IXPs, and CDNs. We've now discussed YouTube's feature that recommends content. What is the difference between popular and recommended content on YouTube?

Show Answer

Q1 / Q3

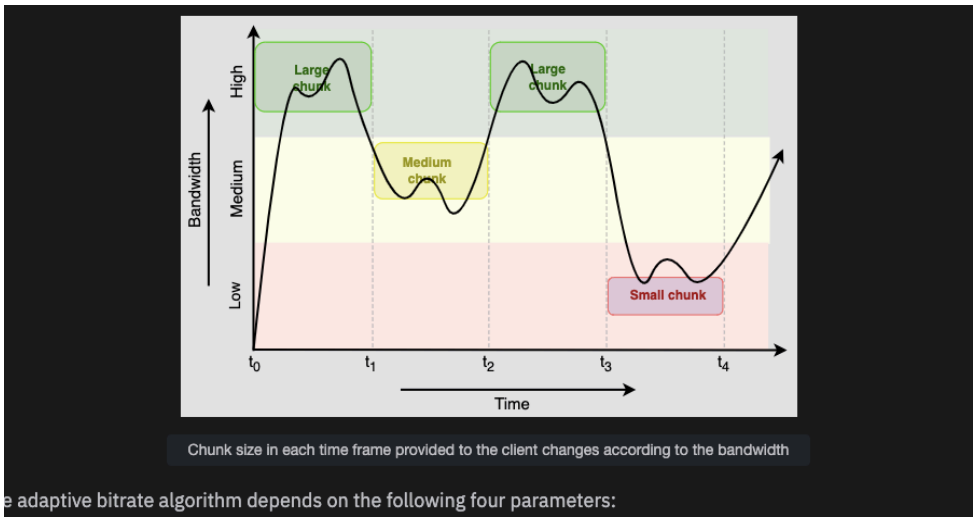
Deliver
Let's see how the end user gets the content on their device. Since we have the chunks of videos already deployed near users, we redirect users to the nearest available chunks. As shown below, whenever a user requests content that YouTube has recognized as popular, YouTube redirects the user to the nearest CDN.
However, in the case of non-popular content, the user is served from colocation sites or YouTube's data center where the content is stored initially. We have already learned how YouTube can reduce latency times by having distributed caches at different design layers.



The user requests for chunk using the ISP.
Adaptive streaming
While the content is being served, the bandwidth of the user is also being monitored at all times. Since the video is divided into chunks of different qualities, each of the same time frame, the chunks are provided to clients based on changing network conditions.
As shown below, when the bandwidth is high, a higher quality chunk is sent to the client and vice versa.

Chunk size in each time frame provided to the client changes according to the bandwidth
The adaptive bitrate algorithm depends on the following four parameters:

1. End-to-end available bandwidth (from a CDN/servers to a specific client).
2. The device capabilities of the user.
3. Encoding techniques used.
4. The buffer space at the client [source].

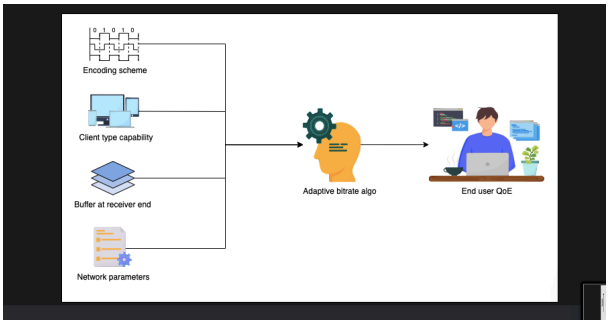


We have now discussed the detailed system design of YouTube. In the requirements of YouTube's design, we mentioned the need for our design solution to have low latency as a primary non-functional requirement.

What were the measures taken in our design to ensure low latency?

Saved

State the ways we achieve low latency in the design of YouTube.
Provide your answer here and use a separate line for each measure.



Give me a Hint
Potential follow-up questions
There can be many different aspects of the system design of YouTube as there is more depth in the subject area. Therefore, many questions can arise. Some questions and directions toward their answers are as follows:

1. **Question**: It is possible to quantify availability by adding numbers like 99.99% availability or 99.999% availability. In that case, what design changes will be required?
   This is a hard question. In reality, such numbers are part of a service's SLA, and they are generated from models or long-term empirical studies. While it is good to know how the above numbers are obtained and how organizations use monitoring to keep availability high, it might be a better strategy to discuss the fault tolerance of the system—what will happen if there are software faults, server failures, full data center failures, and so on. If the system is resilient against faults, that implies the system will have good availability.

2. **Question**: We assumed some reasonable numbers to come up with broad resource requirements. For example, we said the average length of a video is five minutes. In this case, are we designing for average behavior? What will happen to users who don't follow the average profile?
   A possible answer could be that the above number will likely change over time. Our system design should be horizontally scalable so that with increasing users, the system keeps functioning adequately. Practically, systems might not scale when some aspect of the system increases by an order of magnitude. When some aspects of a system increase by an order of magnitude (for example, 10x), we usually need a new, different design. Cost points

of designing 10x and 100x scales are very different.

3. **Question**: Why didn't we discuss and estimate resources for video comments and likes?
   Concurrent users' comments on videos are roughly at the same complexity as designing a messaging system. We'll discuss that problem elsewhere in the course.

4. **Question**: How to manage unexpected spikes in system load?
   A possible answer is that because our design is horizontally scalable, we can shed some load on the public cloud due to its elasticity features. However, public clouds are not infinitely scalable. They often need a priori business contracts that might put a limit on maximum, concurrently allowed resource use at different data centers.

5. **Question**: How will we deploy a global network to connect data centers and CDN sites?
   In practice, YouTube uses Google's network, which is built for that purpose and peers with many ISPs of the world. It is a fascinating field of study that we have left outside this course for further review.

6. **Question**: Why isn't there more detail on audio/video encoding?
   There are many audio/video encoding choices, many publicly known and some proprietary. Due to excessive redundancy in multimedia content, encoding is often able to reduce a huge raw format content to a much smaller size (for example, from 600 MB to 30 MB). We have left the details of such encoding algorithms to you if you're interested in further exploration.

7. **Question**: Cant we use specialized hardware (or accelerators like GPUs) to speed up some aspects of the YouTube computation?
   When we estimated the number of servers, we assumed that any server could fulfill any required functionality. In reality, with the slowing of Moore's law, we have special-purpose hardware available (for example, hardware encoders/decoders, machine-learning accelerators like Tensor Processing Units, and many more). All such platforms need their own courses to do justice to the content. So, we avoided that discussion in this design problem.

8. **Question**: Should compression be performed at the client-side or the server-side during the content uploading stage?
   We might use some lossless but fast compression (for example, Google Snappy) on the client end to reduce data that needs to be uploaded. This might mean that we'll need a rich client, or we would have to fall back to plain data if the compressor was unavailable. Both of those options add complexity to the system.

9. **Question**: Are there any other benefits to making file chunks other than in adaptive bitrates?
   We discussed video file chunks in the context of adaptive bit rates only. Such chunks also help to parallelize any preprocessing, which is important to meet real-time requirements, especially for live streams. Parallel processing is again a full-fledged topic in itself, and we've left it to you for further exploration.