DESIGN NETFLIX

Question 1
Q: From a high-level point of view, Netflix is a fairly straightforward service: users go on the platform, they're served movies and shows, and they watch them. Are we designing this high-level system entirely, or would you like me to focus on a particular subsystem, like the Netflix home page?

A: We're just designing the core Netflix product--so the overarching system / product that you described.

Question 2
Q: Should we worry about auxiliary services like authentication and payments?

A: You can ignore those auxiliary services; focus on the primary user flow. That being said, one thing to note is that, by nature of the product, we're going to have access to a lot of user-activity data that's going to need to be processed in order to enable Netflix's recommendation system. You'll need to come up with a way to aggregate and process user-activity data on the website.

Question 3
Q: For this recommendation system, should I think about the kind of algorithm that'll fuel it?

A: No, you don't need to worry about implementing any algorithm or formula for the recommendation engine. You just need to think about how user-activity data will be gathered and processed**.It sounds like there are 2 main points of focus in this system: the video-serving service and the recommendation engine.** Regarding the video-serving service, I'm assuming that we're looking for high availability and fast latencies globally; is this correct ? Yes, but just to clarify, the video-streaming service is actually the only part of the system for which we care about fast latencies.

Question 4
Q: So is the recommendation engine a system that consumes the user-activity data you mentioned and operates asynchronously in the background?

A: Yes.

Question 5
Q: How many users do we expect to be building this for?

A: Netflix has about 100M to 200M users, so let's go with 200M.

Question 6
Q: Should we worry about designing this for various clients, like desktop clients, mobile clients, etc.?

A: Even though we're indeed designing Netflix to be used by all sorts of clients, let's focus purely on the distributed-system component--so no need to get into details about clients or to optimize for certain clients.

1. GATHERING SYSTEM REQUIREMENTS

As with any systems design interview question, the first thing that we want to do is to gather system requirements; we need to figure out what system we're building exactly.

We're designing the core Netflix service, which allows users to stream movies and shows from the Netflix website.

Specifically, we'll want to focus on:

- **Delivering large amounts of high-definition video content** to hundreds of millions of users around the globe without too much buffering.
- Processing large amounts of **user-activity data to support Netflix's recommendation engine.**

2. COMING UP WITH A PLAN

We'll tackle this system by dividing it into four main sections:

1. Storage (Video Content, Static Content, and User Metadata)
2. General Client-Server Interaction (i.e., the life of a query)
3. Video Content Delivery
4. User-Activity Data Processing

3. VIDEO-CONTENT STORAGE

Since Netflix's service, which caters to millions of customers, is centered around video content, we might need *a lot* of storage space and a complex storage solution. Let's start by estimating how much space we'll need.

We were told that Netflix has about 200 million users; we can make a few assumptions about other Netflix metrics (alternatively, we can ask our interviewer for guidance here):

- Netflix offers roughly 10 thousand movies and shows at any given time
- Since movies can be up to 2+ hours in length and shows tend to be between 20 and 40 minutes per episode, we can assume an average video length of 1 hour
- Each movie / show will have a **Standard Definition** version and a **High Definition** version. Per hour, SD will take up about 10GB of space, while HD will take about 20GB.

~10K videos (stored in SD & HD)
~1 hour average video length
~10 GB/h for SD + ~20 GB/h for HD = 30 GB/h per video
~30 GB/h * 10K videos = 300,000 GB = 300 TB

This number highlights the importance of estimations. Naively, one might think that Netflix stores many petabytes of video, since its core product revolves around video content; but a simple back-of-the-napkin estimation shows us that it actually stores a very modest amount of video.

This is because Netflix, unlike other servies like YouTube, Google Drive, and Facebook, has a bounded amount of video content: the movies and shows that it offers; those other services allow users to upload unlimited amounts of video.

Since we're only dealing with a few hundred terabytes of data, we can use a simple **blob storage solution like S3 or GCS to reliably handle the storage and replication of Netflix's video content**; we don't need a more complex data-storage solution.

4. STATIC-CONTENT STORAGE

Apart from video content, we'll want to store various pieces of static content for Netflix's movies and shows, including video titles, descriptions, and cast lists.

This content will be bounded in size by the size of the video content, since it'll be tied to the number of movies and shows, just like the video content, and since it'll naturally take up less space than the video data.

We can easily store all of this **static content in a relational database or even in a document store**, and we can cache most of it in our API servers.

5. USER METADATA STORAGE

We can expect to store some user metadata for each video on the Netflix platform. For instance, we might want to store the timestamp that a user left a video at, a user's rating on a video, etc..

Just like the static content mentioned above, this user metadata will be tied to the number of videos on Netflix. However, unlike the static content, this user metadata will grow with the Netflix userbase, since each user will have user metadata.

We can quickly estimate how much space we'll need for this user metadata:

~200M users
~1K videos watched per user per lifetime (~10% of total content)
~100 bytes/video/user
~100 bytes * 1K videos * 200M users = 100 KB * 200M = 1 GB * 20K = 20 TB

Perhaps surprisingly, we'll be storing an amount of user metadata in the same ballpark as the amount of video content that we'll be storing. Once again, this is because of the bounded nature of Netflix's video content, which is in stark contrast with the unbounded nature of its userbase.

We'll likely need to query this metadata, so storing it in a **classic relational database like Postgres** makes sense.

Since Netflix users are effectively isolated from one another (they aren't connected like they would be on a social-media platform, for example), we can expect all of our latency-sensitive database operations to only relate to individual users. In other words, potential operations like *GetUserInfo* and *GetUserWatchedVideos*, which would require fast latencies, are specific to a particular users; on the other hand, complicated database operations involving multiple users' metadata will likely be part of background data-engineering jobs that don't care about latency.

Given this, we can split our **user-metadata database into a handful of shards**, each managing anywhere between 1 and 10 TB of indexed data. This will maintain very quick reads and writes for a given user.

6. General Client-Server Interaction
The part of the system that handles serving user metadata and static content to users shouldn't be too complicated.

We can use some simple **round-robin load balancing** to distribute end-user network requests across our API servers, which can then load-balance database requests according to userId (since our database will be **sharded based on userId**).

As mentioned above, we can cache our static content in our API servers, periodically updating it when new movies and shows are released, and we can even cache user metadata there, using a **write-through caching mechanism**. We'll basically have an auxillary service we'll call "Cache Populator" which'll select certain video content from blob storage and populate in the CDN (cache of the server)

7. VIDEO CONTENT DELIVERY

We need to figure out how we'll be delivering Netflix's video content across the globe with little latency. To start, we'll estimate the maximum amount of bandwidth consumption that we could expect at any point in time. We'll assume that, at peak traffic, like when a popular movie comes out, a fairly large number of Netflix users might be streaming video content concurrently.

~200M total users
~5% of total users streaming concurrently during peak hours
~20 GB/h of HD video ~= 5 MB/s of HD video
~5% of 200M * 5 MB/s = 10M * 5 MB/s = 50 TB/s

This level of bandwidth consumption means we can't just naively serve the video content out of a single data center or even dozens of data centers. We need many thousands of locations around the world to be distributing this content for us. Thankfully, **CDN**s solve this precise problem, since they have many thousands of **Points of Presence (PoPs)** around the world. We can thus use a CDN like Cloudflare or Google Cloud CDN and serve our video content out of the CDN's PoPs.

CDNs are basically 3rd party services that act as cache for your servers. Basically what happens is that sometimes your web-app may be slower in certain regions if your servers are in another region only. Because CDNs have thousands of servers around the world the latency via their servers will always be better, streaming will be faster.

Since the PoPs can't keep the entirety of Netflix's video content in cache, we can have an external service that periodically repopulates CDN PoPs with the most important content (the movies and shows most likely to be watched).

In the real world, Netflix itself solves this issue of – Having the popular video content available locally – by partnering with Internet Service Providers (eg : AT&Ts and Verizon) and inject a special caching layer crafted by Netflix at locations that are called Internet Exchange Points (IXPs – around 10k+ around the world)

8. USER-ACTIVITY DATA PROCESSING

We need to figure out how we'll process vast amounts of user-activity data to feed into Netflix's recommendation engine. We can imagine that this user-activity data will be gathered in the form of logs that are generated by all sorts of user actions; we can expect terabytes of these logs to be generated every day.

mapReduce can help us here. We can store the logs in a distributed file system like **HDFS** and run mapReduce jobs to process massive amounts of data in parallel. The results of these jobs can then be fed into some machine learning pipelines or simply stored in a database.

Map Inputs
Our Map inputs can be our raw logs, which might look like:

{"userId": "userId1", "videoId": "videoId1", "event": "CLICK"}
{"userId": "userId2", "videoId": "videoId2", "event": "PAUSE"}

{"userId": "userId3", "videoId": "videoId3", "event": "MOUSE_MOVE"}

Map Outputs / Reduce Inputs
Our Map function will aggregate logs based on userId and return intermediary key-value pairs indexed on each userId, pointing to lists of tuples with videoIds and relevant events.

These intermediary k/v pairs will be shuffled appropriately and fed into our Reduce functions.

{"userId1": [("CLICK", "videoId1"), ("CLICK", "videoId1"), ..., ("PAUSE", "videoId2")]}
{"userId2": [("PLAY", "videoId1"), ("MOUSE_MOVE", "videoId2"), ..., ("MINIMIZE", "videoId3")]}

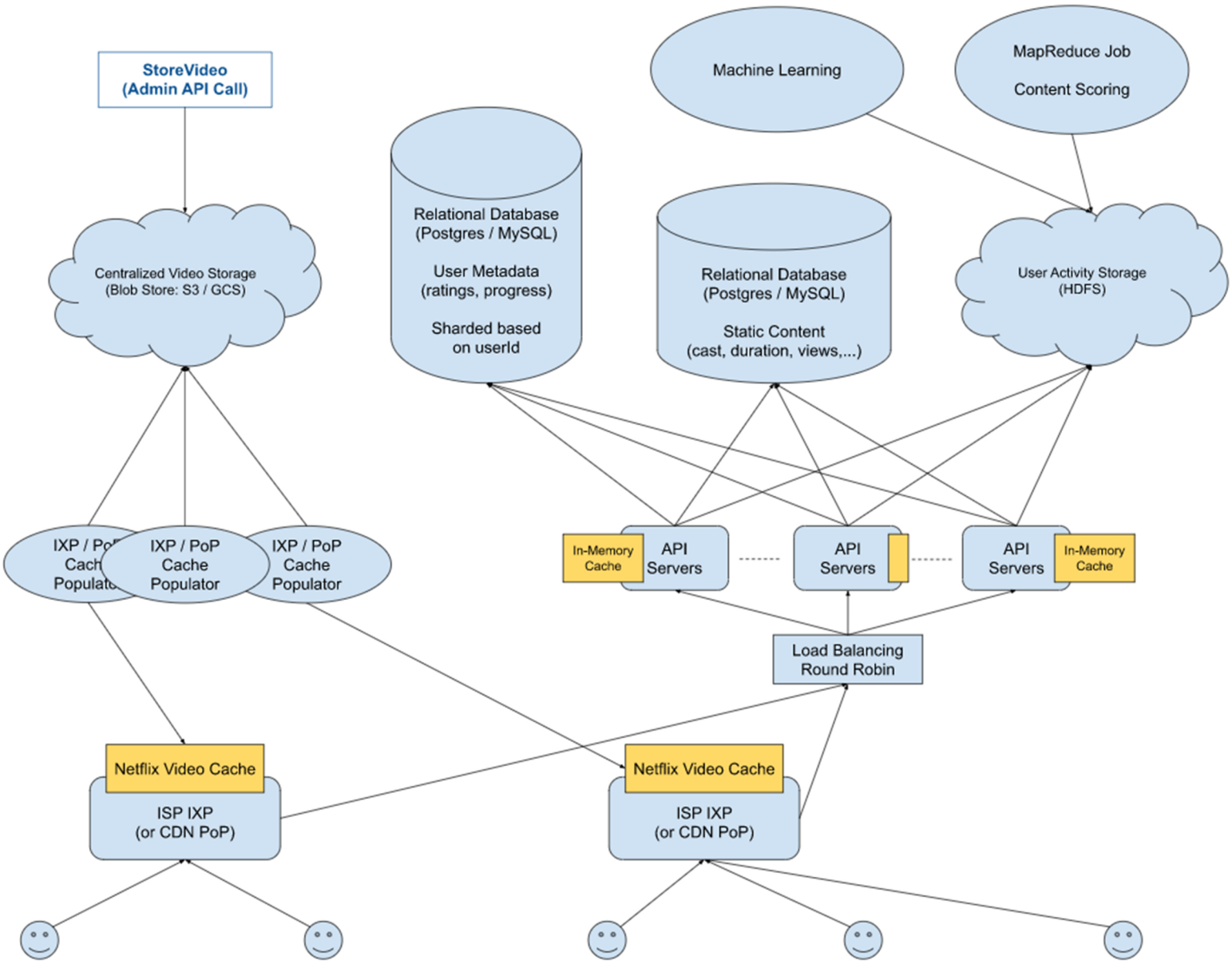
Reduce Outputs
Our Reduce functions could return many different outputs. They could return k/v pairs for each userId|videoId combination, pointing to a computed score for that user/video pair; they could return k/v pairs indexed at each userId, pointing to lists of (videoId, score) tuples; or they could return k/v pairs also indexed at eacher userId but pointing to stack-rankings of videoIds, based on their computed score.
("userId1|videoId1", score)
("userId1|videoId2", score)

OR

{"userId1": [("videoId1", score), ("videoId2", score), ..., ("videoId3", score)]}
{"userId2": [("videoId1", score), ("videoId2", score), ..., ("videoId3", score)]}

OR

("userId1", ["videoId1", "videoId2", ..., "videoId3"])
("userId2", ["videoId1", "videoId2", ..., "videoId3"])


StoreVideo
(Admin API Call)
Centralized Video Storage
(Blob Store: S3 / GCS)
Machine Learning
MapReduce Job
Content Scoring
Relational Database
(Postgres / MySQL)
User Metadata
(ratings, progress)
Sharded based
on userId
Relational Database
(Postgres / MySQL)
Static Content
(cast, duration, views,...)
User Activity Storage
(HDFS)
IXP / PoP
Cache
Populator
IXP / PoP
Cache
Populator
IXP / PoP
Cache
Populator
In-Memory
Cache
API
Servers
API
Servers
API
Servers
In-Memory
Cache
Load Balancing
Round Robin
Netflix Video Cache
ISP IXP
(or CDN PoP)
Netflix Video Cache
ISP IXP
(or CDN PoP)

Q] How does Netflix onboard new content?

Ans- The tasks done when storing a new video are

1) It is stored in diff formats (high quality to low quality) (codecs) → mp4, AVI...

Reason : Diff ppl have diff connec^n speed

Codec :- A way in which you compress video

Lossy compression :- ↓ vid file size by ↓ quality (as you lose some detailing inside data)

2) Use different resolu^{n}s :- If you're playing vid on phone then the res^n req'd to play video is much smaller than res^n req'd for larger screens (TV, laptop..etc.)

→ Created in pairs. You have ~~tuples (for~~
1 tuple = ['format, res^n']

Eg:- low quality, 720p file

No. of videos that you end up processing = (no. of formats) [no. of res^{n}s]

∴ When → new content is uploaded
→ new way of ~~more~~ compressing vids more efficiently is discovered.

Then → all vids are taken and compressed.

Date:
M T W T F S S

However, compressing an entire video (say 6GB) isn't done by just 1 computer

Reason → takes too much time
→ computer is single pt. of failure

∴ Each video is broken up into chunks that are processed separately

1 chunk ⇒ 1 resⁿ, 1 format

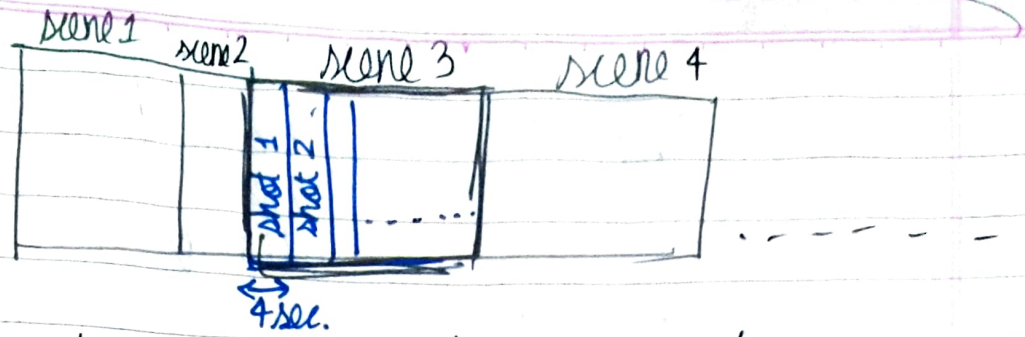
Videos are broken into chunks based on scenes (a collⁿ of 4 sec. shots) such that the ~~scene~~ breaking into chunks/scenes is based on the type of thing happening in the video

eg:- A continuous acⁿ scene for 3 mins, a 4 min. song, a 1 min scene snippet.

eg of a chunk :- Chunk A → 1080p
→ mp4

describes the 1st acⁿ scene of the video.

→ Reason → UX is better. If chunkin were done say at every 3 min. the user's comp. would have to make an API call say in the middle of an engaging scene viz. not good UX



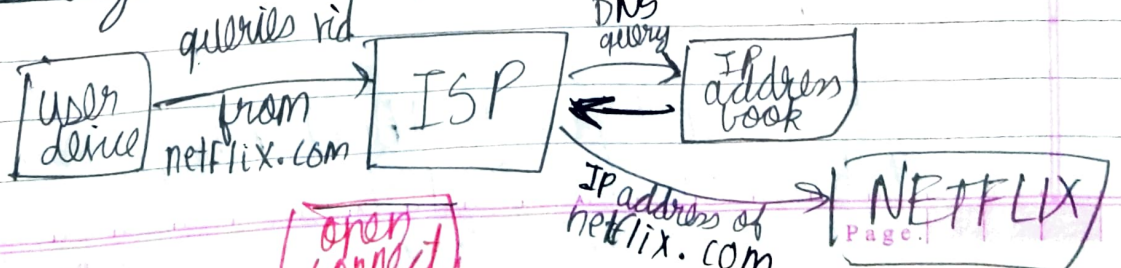
Netflix sees vid → treats it as a set of chunks

sparse vids { If people arbitrarily see diff pts in vid (e.g.:- see a scene, then hit a completely diff timestamp for a completely diff scene), then Netflix's predictn algo just sends the data being asked for.

dense vids { If the vid is too engagin, i.e. the ppl are watchin over cont. periods of time and you can easily say that linearly (in the vid timeline) which ~~chunk~~ scene is gonna be picked up next, then the Netflix predictn algo proactively fetches next part/chunk and stores it on user's device

Storage soln for static content used by Netflix is → Amazon S3
↓
cheap alternative to a dB

Caching most popular regional vids at ISP level



Open Connect → Kinda like a massive hard drive that has a lotta vids. stored

Advantage →
- less bandwidth saved instead of hitting far off stored Netflix servers to fetch video
- → time is saved as API calls are quicker
- → much better UX
- → localised :- vids. popular to a local region are stored here
  eg:- Netflix Sacred Games stored on Open Connect for India's region
- → reduced load on user & ISP

These 'Open Connects' take care of more than 90% of Netflix's traffics

∴ When a new vid is uploaded (after being registered on Netflix)

① It is processed, compressed to create diff chunks

② At a time (say 4AM) when load on these open connects is min. Netflix servers have a lotta write op"s to these open connects to populate them with these new vid chunks