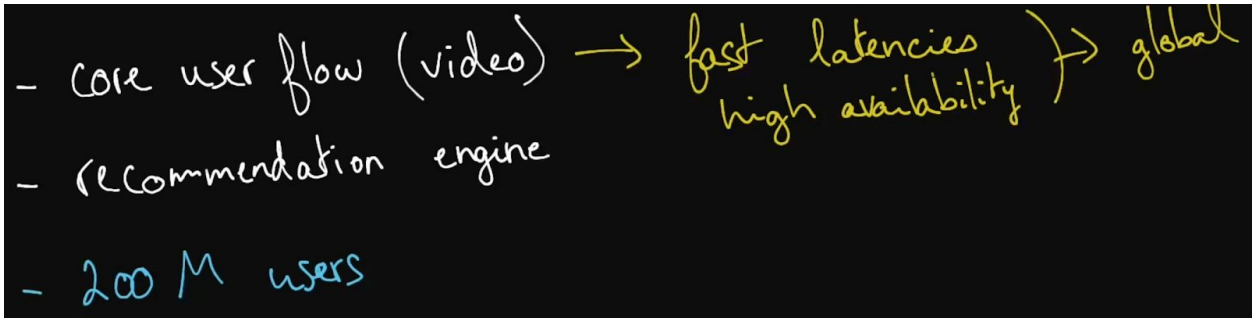
Netflix

Problem: Design Netflix

Solution:

Requirements:

1. Design the core Netflix product, which is serving customers movies and shows.
2. You can ignore auxiliary services such as authentication and payments; focus on the primary user flow.
    - 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 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. For example, you could process it into a set of scores for the content.
3. The video-serving service must have high availability and low latencies
4. The recommendation engine operates asynchronously is the background
5. This system will be designed for roughly 200M users globally



Let's first think about what type of data we're going to be storing.
Types of data:

- Video data
- User metadata (whether a user has watched the video or how much of the video the user has watched)
- Static content (names of movies, descriptions, thumbnails, cast of a particular movie)
- Logs

Let's estimate the storage that we would need

1. Video storage
    - Netflix has a limited number of movies, which are picked by them. Netflix is not like YouTube, where people can upload as many videos as they like.
    - We could estimate Netflix having around 10000 movies/videos

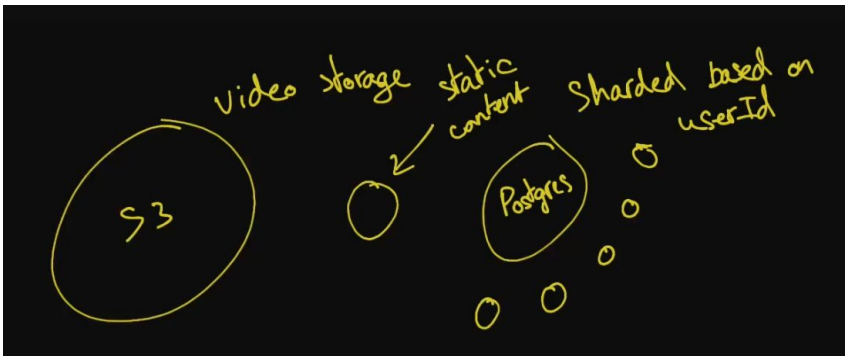
- Netflix also has varying resolutions for their videos. We could assume we would store **Standard Definition (SD)** and **High Definition (HD)** videos.
- Let's also assume that each video would be 1 hour on average since a movie would usually be 2 hours long and an episode of a show would be around 20 mins long.
- 1hr of SD video is approximately 1GB
- 1hr of HD video is approximately 2GB
  Total video storage = total number of videos * (SD + HD)
  = 10000 * (1+2) = 30000 GB = **30TB**

- 30TB is too large of a data so we could use storage solution such as S3 to store our data

2. User metadata - we will have to store user metadata for 200M users.
   - We could store data on a per video basis. For example, has this user watched this video, or how much of this video has the user watched
   - Let's assume an average person watches 2 shows a week. Within a year, let's say there's 50 weeks, which equates to 100 shows per year.
   - Netflix started streaming in 2007 so we can say the total numbers of years spent by a user is 10 years, which equates to 1000 shows per user
   - Overall, we will be storing one user metadata for 1000 videos
   - We could estimate that a user metadata would not be too much data since it will only be if a user has watched a video (1 or 0 / True or False). We could assume that we will be using 100 bytes per video
   - **Metadata per user** = 100bytes * 1000 videos = **100KB**
   - **Total user metadata** = 200M users * 100KB = **20TB**
   - We could store this data into a relational database such as PostgreSQL
   - In addition, since users will be querying their metadata from the database frequently and we want low latencies for our users, we could do sharding on our Postgres database at the **user_id** level
   - Maybe we could split out database into 5 shards, which will give 4TB of data to each shard

3. For the static content, there will be static content for every video. For example, a video will always have a title and a video description. We could just store this data in a relational database

Overall, the storage aspect of the system should look something like:



The next area we'll tackle is the actual streaming or serving of the video data

- First we could estimate our bandwidth consumption at any given point in time, to see how much data we're going to need to serve.
- We have 200M users but we'll assume that all 200M users are not going to be watching the same video all at once. We could assume that there will be peak times
- For example, there is a very popular show. We could say that our peak traffic would be 10M users watching a video all at once.
- Let's say the 10M users watches the video in HD so we have the upper boundary
- Let's convert 2GB/hr, which is roughly 4.5 Mbps

> **4**-6 **mbps**: Will provide a good Web surfing experience. Often fast enough to stream a 720p high-definition video, and it's possible to download some videos within about 20 minutes at this speed. But **4 mbps** can still be sluggish. 6-10 **mbps**: Usually an excellent Web surfing experience. 1 Jul 2015

- **Total bandwidth consumption (peak traffic)** = 10M * 4.5Mbps = **5.5 TBps**

Now we think about the way we would transfer this data to customers.

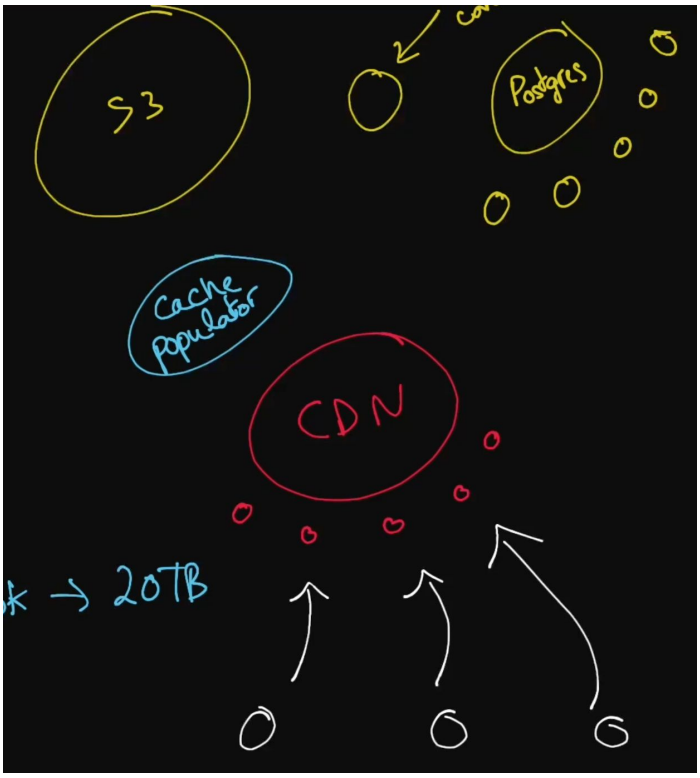
- We don't want all of this data to be stored in one database as this will cause some kind of a bottleneck. We have to avoid overloading our database with too many requests.
- What we could do is utilise **CDN** (Content Delivery Network) such as Cloudflare or AWS Cloudfront
- A CDN will have different Points of Presence around the world which are just a group of machines that can store our data and serve the data from these points. This results in lower latencies and helps not to send too many requests to our main database.

- We have to make sure that each point in our CDN has the right data locally. We don't want to do a request then go back to our database to send a request for the right data.

- The photo below illustrates the architecture for our CDN.

Big red circle = CDN
Small red circles = Points of presence
White circles = users requesting data from CDN PoPs
Blue circle = service for populating the cache (cache populator)

- How this will work is that our CDN PoPs will have the video content cached
- We would use a service for populating the cache in between our main video content database (S3) and asynchronously populate our PoPs with the content that they need to have.
- For example, when Netflix is releasing a new movie, this cache populator will make sure to have these movies be sent to those PoPs. We could maybe prioritise popular movies, deporioritise unpopular movies etc.

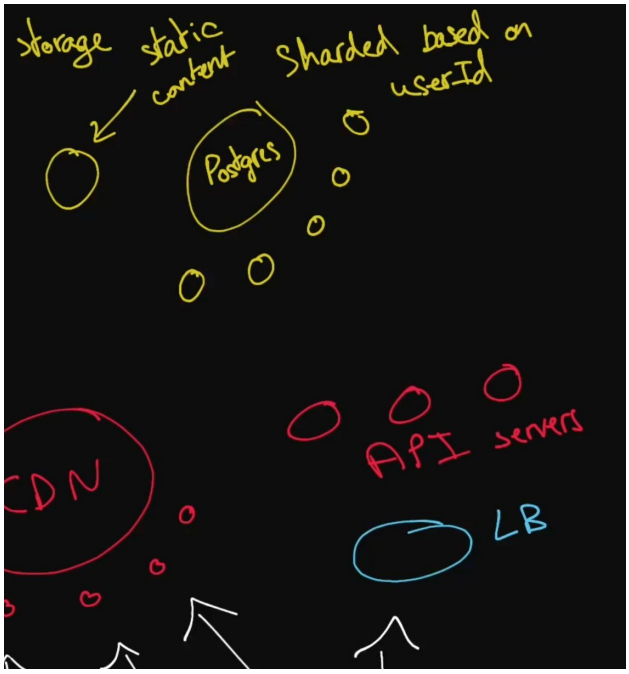


- Turns out in real life, Netflix approaches their content delivery service by partnering with ISPs and injects a special caching layer created by Netflix at locations which we call IXPs (Internet Exchange Points). This essentially behaves like an optimised CDN, which allows for a much cheaper bandwidth for Netflix and lower latencies for users as well.

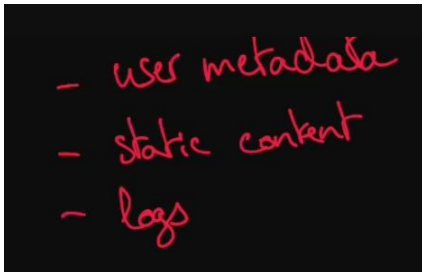
Netflix Content Delivery System
https://medium.com/@christophelimpalair/why-netflix-streaming-works-so-well-hint-they-cheat-be62e2c737b1

- Now for delivering our static content and user metadata
- We could set up multiple API servers that users can interact with to send requests to our databases.
- Maybe have a load balancer that redirects traffic using the round robin approach between the API servers, or maybe load balances by user_id level since we're already sharding our metadata by user_id. The requests goes to the load balancer then depending on the user_id, the API server redirects the request to the right shard

- Then for our static content data, since the data is not that large, an option is to just cache the static content data at the API server level. This cache can be updated regularly, maybe once every day.



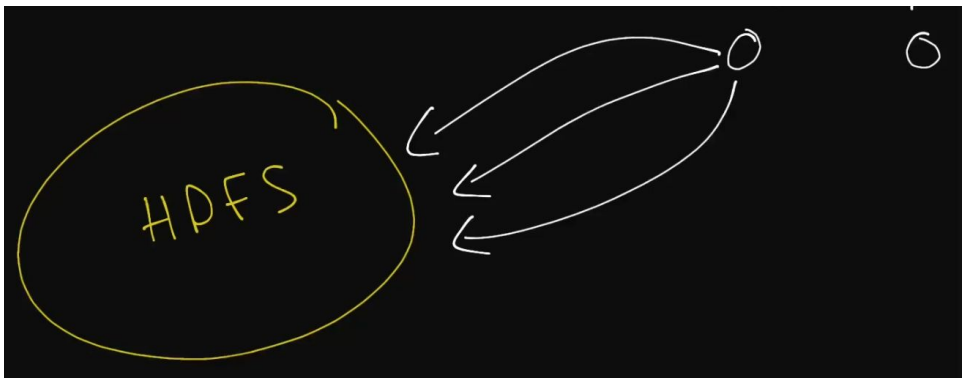
Recommendation Engine



Now let's design the recommendation engine. Now, since we're also gathering logs.

- We could have these logs be fed into the system that backs the recommendation engine.

- We could have a MapReduce job that asynchronously consumes all these logs that are stored somewhere and then spits out a more meaningful data that is used elsewhere in our system.
- SInce we're performing MapRecude jobs on this data, we're going to need a Distributed File System.
- We can store our logs in a DFS like the Hadoop Distributed File System (HDFS)
- Logs are going to be coming from different machines and stored in HDFS.



- Then we've got this asynchronous MapReduce job that takes the logs from the HDFS and actually processes those logs data for the recommendation engine.