**Facebook**

**https://www.youtube.com/watch?v=9-hjBGxuiEs**

**Functional Requirements:**

1. POST (Image + Video)
2. Like, Comment, Share a post.
3. Add Friends
   1. A person should be able to go to someone’s profile and add them as friend.
   2. Non- directional friendship (A is B’s friend => B is A’s friend)
   3. There is a concept of weightage of friendship which is directional
4. See Timeline
   1. See data posted by friends
5. See a User’s Post, Anyone’s timeline/ profile, self -profile/timeline
6. Activity log
   1. Log everything that a user does on this platform (Post, like, share, comment etc.)

**Non-Functional Requirements:**

1. Read Heavy
   1. If one person is posting something, all his friends will be seeing that. For ex: if you create 1 post, 100s of users will see it. That is called read heavy.
2. Fast Rendering, Posting Time
3. Lag is ok.
   1. Latency is diff from Lag.
   2. When page is loaded, it should render quickly. But its ok if you see someone’s post after 20 secs.
4. Access Patterns
   1. Every social network has unique access pattens. Whenever a new post is created, many comments/likes will start to come. Then it will reach peak (highest interaction) and then after couple of weeks the activity on that post will start decaying. After a certain time, there can be no interaction on that post.
   2. This can help in handling videos/photos. Because this is heavy content, and we can optimize some cost on hardware by analyzing access pattern.
5. Global
   1. Huge variety of devices.
   2. Multiple languages content
   3. Platform should handle multiple languages.
   4. Wide variety of internet bandwidths
   5. Various geographies (we should have servers in every geography so that it is not slow anywhere)
6. Scale
   1. 1.7 billion unique Active users
   2. 2.6 billion Monthly Unique Active users
   3. 95% users access through Mobile (take care of mobile specific optimizations)
   4. Events/min (heavy volume, we can’t use caching)
      1. 150K images
      2. 300K statuses
      3. 500K comments

**Users Categories:**

1. Famous
   1. Facebook has a max limit for number of friends, some we need to make some optimizations for them so that their page loads are faster enough
2. Active
   1. people who accessed FB in last 3 days, we need to make some optimizations for them so that their page loads are faster
3. Live
   1. Currently Browsing, notify them about new contents without their request by connection of their device and app server
4. Passive
   1. We will not cache anything for them.
5. Inactive
   1. De-activated account, fake accounts, who doesn’t access their accounts etc.

**Overall Architecture:**

1. **User Onboarding / ADD Friend:**

KAFKA

LB

User Service

User Onboarding

MySQL Cluster

MySQL Cluster

LB

Graph Service

Add Friend

Redis

**User Onboarding: (Create/ Update/ See Profile)**

* **User Interaction point**: Mobile, Browser. It will send a request from device.
* **LB**: Load Balancer + Reverse Proxy + Authentication + Authorization. Which will validate the API calls coming to system
* **User Service** (Web Services. Ex: Kafka consumers) Most of requests are handled by User service. Primary source of information for user related data. Provides APIs: getUserbyId(), ..
  + **MySQL DB:** User Service sits on MySQL clusters. Based on data volume, sharding can be used. User info is a standard relational set of datapoints. That’s why MySQL relational DB is used. The user’s info is not updated very frequently (mobile number, DOB, email etc. is kind of constant). So, MySQL is fairly good DB for Users’ data.
  + **Redis:** User Service also sits on Redis. Lots of users’ info is cached in Redis.
  + If some API is called, we will first check the Redis.
    - If Redis has info: It will return info to user
    - If Redis doesn’t have info: It will query **MySQL DB -> get info -> update Redis -> send response back to use**r
* **KAFKA:** Many events are pushed into KAFKA. For ex:
  + New account creation event is pushed to KAFKA for fraud check
  + Update email event is pushed to KAFKA for sending notifications to others about the change.

**ADD Friend:**

We can access anyone’s profile and add them as friend.

* **User Interaction point**: Mobile, Browser. It will send a request from device.
* **LB**: Load Balancer + Reverse Proxy + Authentication + Authorization. Which will validate the API calls coming to system
* **Graph Service**: Maintains graph of complete network. Keeps relationships and their weightages. Provides APIs to infer how close are two people.
  + **MySQL DB:** Graph Service sits on MySQL clusters. Based on data volume, sharding can be used. It stores basic core info about a user and its friends. Basically, a mapping table. So relational DB is well suited here.
  + **Redis:** Relationships are cached. Key: user\_id, value: list of friends
  + **Others things to store in redis:**
    - Key: user\_id,
    - values: user\_details, friends, user\_type (active, famous etc.), relavance\_tags (use analytics to show information of one’s interest), last\_access\_time(last online time can be used for some functionality like chat etc.)
  + Based on volume of data we can use different Redis clusters for user/graph service.

Diagram

Description automatically generated

**Short URL**: service which will convert url into short URL. Short URL will be posted on the FB platform.

**Asset Service:**

* adjusts the size and aspect ratio based on the accessing platform. For ex: photo, video content for mobile/ browser.
* Takes care of video streaming
* Based on the access pattern, it will remove the content from **CDN**
* It will decide which content to be kept on **CDN**, which on **S3**

**Create Post:**

**Post Ingestion Service:**

It takes care of data to be posted. After data is processed by **Short URL/Asset Service** the content will be sent to **Cassandra. Cassandra is a good choice here because it can handle high read and write rates. Hbase could also be used.**

**Post Service:**

**Owner of posts.** Provides API for **Post Cassandra.** Ex: getPostById(), getPostOfUserWIthId()

Post ingestion service puts the post in Cassandra and create an event in KAFKA. Then its spread out further for other processes.

**How post comes to User:**

**Analytics:**

Many components are used for analytics. There will be a streaming consumer on KAFKA which will keep listening to all events coming. It will take out the post and tag it to a machine learning model. It applies some classification model and create a tag on the post. This tag is used to score the post on various relevance parameters. After adding tag, the post is kept back to KAFKA.

This happens at the time of post creation. Some lag is OK before we send it to users.

**Post Processor:**

Gets post event from KAFKA. This post has tags attached along with the user id. It will figure the friends of the user who potentially should see the post. The post ‘s viewer are added based on the relevance tags also.

It gets the subset of users who should see it and puts into Redis. Redis is the timeline of users.

**Accessing Your Timeline (NewsFeed) /Other User TIMELINE:**

**If the user is a famous person:**

He will be having many friends(millions). If he creates a post. Post Processor will have to update the Redis for millions of users. Which is not scalable. So we give that responsibility to timeline service.

**Seeing another person Timeline:**

It will go to timeline service -> post service -> Cassandra

**If seeing own timeline:**

User and groups service will give the list of friends.

Timeline service will get the posts from the friends which are normal users from REDIS.

Timeline service will get the posts from the friends which are famous users from Post Service since the posts of famous users will not be in the REDIS. (Remember, Celebrity/Famous users post will not be put into redis since they have millions of followers and we will have to put a post into millions of timelines/Redis and hence for famous users, we don’t do anything. When a user accesses his timeline, we check redis for normal friends and access posts service for famous/celebrity friends and merge them)

Merge both posts above and return back to user. But before sending back, timeline service can update Redis as well with the timestamp. For next request if the timestamp is nearby (few mins) we can return the data as it is from Redis.

**Optimizations for Live Users:**

Post processor figures out that a live event is posted, it will update the Redis and get the friends list from User & group service. It will create a new event with another topic and push to KAFKA.

Live user Service takes the event. This service has open connections web sockets with all the clients. It will notify the friend list that this person is live via an update.

**Archival Service:**

Redis can’t have huge data. We can just put today’s data. Archival service will fetch all user’s time at a particular time in day. It will put the info in Aggregated timeline Cassandra and clears Redis cache.

**Aggregate Timeline Cassandra:**

It will store the timeline in this format for all users:

User i, time t: {…….}

It can be scaled.

Good for all type of users (active, famous etc.)

**Timeline service:**

It can query from Redis + Post Service + Archival Service(if want to see older data). The older data need not be computed at that time. It can directly be looked up from Aggregated Timeline Cassandra.

EVERYWHERE WE SAY WE ARE CACHING SOMETHING, WE ARE STORING POST IDS. POST SERVICE CAN BE USED TO GET THE POST CONTENTS.

**Potential problem that can arise because of Cassandra:**

**Hotspot**

If partition key is not chosen wisely, it can cause that a particular machine is heavily accessed. All traffic going on that. For ex: if we keep partition key as date range, all todays read and write will happen on same machine. This can cause hotspot.

User id can be good partition key.

**LIKE COMMENTS Handling:**

Diagram

Description automatically generated

**Like:**

UI -> Like Service -> Cassandra -> **update Redis** -> put event in KAFKA

**Like Service** will handle everything related to likes: getLikeForAPost(), getLikeForAComment(). If the comments also have likes, then the Cassandra will store something like like\_id, content\_id, content\_type which can be either a Post or a comment

Cassandra will store the post\_ids, users\_ids, post\_type etc. for likes

Redis: used to save number of likes on the post. It will just return the numbers when a post is shown only for recent posts.

**Comment:**

Comment Service: Repository of all the comments in the system. APIs like: getCommentsForAPost(),

UI -> comment Service -> Cassandra ->put event in KAFKA

Cassandra will store the post\_ids, user\_ids, comments etc. No redis caching required since for LIKES REDIS was used to get the aggregated data for number of likes but here we are going to get the comments anyways by Id of post.

**Sharing post:**

Same as post with a parent id of original post.

**Activity Tracker:** Takes info from KAFKA. Track the activity of a user (like, comment, post, share etc.). Saves activity of users into Cassandra. Provides get and post APIs to Activity UI.

Format: user\_id, action\_taken, attributes of action, timestamp, post\_id

Activity UI: Shows all the activity for a user.

**Search:** Takes all events coming to KAFKA, strore into elastic search. Elastic search is very effective for text-based searching. Service which sits on elastic search will query elastic search and returns data. We may cache the results before sending to user (refer twitter video for details)

**Analytics that can be done on the data in KAFKA:**

Classification can be done using ML. For ex: a post is tagged at sport.

If a person is posting a lot of sport related posts.

**Spark Streaming:** put all info about that user (post, comment, like etc) into Hadoop

**Spark cluster** will do the user profiling. This will classify user’s interests. User profile is pushed back to KAFKA. This event is used by User Service. User service will create tags on user which can be used for various things in System.

**Graph weight:**

A and B are friends. A likes/comments on most of B’s posts. This info can be captured in weighted graph. That mean A is more interested in B’s posts as compared to anyone else’s posts. Then most of B’s post will be shown to A. It runs on spark cluster. Ut can takes data from Hadoop.

**Trends:**

What people are taking about right now. It will take all comments/post by users. Tokenize them by splitting and placing space character. Keep all words/phrases with their count in Redis. This can tell what is trending now.

**Non- functional requirements:**

All services mentioned are horizontally scalable. You can put new nodes based on the traffic/size.

Lots of technologies being used. Lots of monitoring/ alerting is required.

For ex: for web service, we need to measure latency, throughputs, disk usage, CPU mem etc.