Overview

Design the architecture for Instagram/twitter news feeds which comprises of the people you follow. This will involve 2 flows: publishing and serving.

Functional Requirement

- Users should be able to post text, image video.
- Should be able to follow other users.
- Should have a feed/timeline comprises of post from friends.
- ~~Users can comment on posts (optional to mention – don't as nested comment are very tricky and is separate problem itself to tackle).~~

Non-Functional Requirements

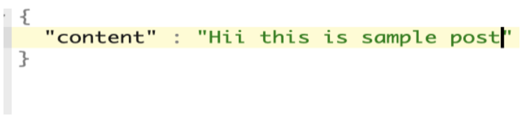
- Low latency.
- High availability
- Near real time.
- Reads >>>> write.

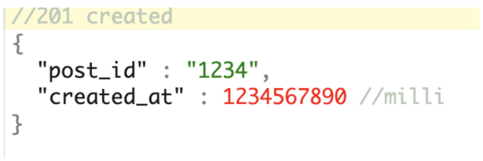
Capacity Estimate

- Daily active users: 1billion.
- Avg user follows : 100 users.
- Avg post/tweet size: 100 characters. Size can be 100 bytes for storing text. 100 bytes for storing other metadata (emoji etc).
- Total number of posts per day: 1000 million = 1000million*365*200 bytes = 73 tb per year of storage.

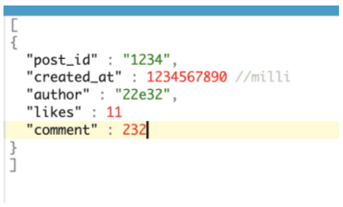
API's

- POST: Publish the post
  /v1/{userid}/post


```
{
  "content" : "Hii this is sample post"
}
```

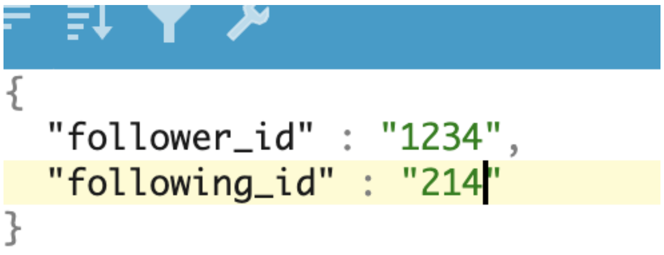

```
//201 created
{
  "post_id" : "1234",
  "created_at" : 1234567890 //milli
}
```

- GET
  /v1/{userid}/feed


```
[
{
  "post_id" : "1234",
  "created_at" : 1234567890 //milli
  "author" : "22e32",
  "likes" : 11
  "comment" : 232
}
]
```

  - May talk about pagination of the api's here
    [Offset based vs cursor based]() (avoid if asked only then mention

- POST
  /v1/users/{action}


```
{
  "follower_id" : "1234",
  "following_id" : "214"
}
```

Database Schema

- POST table
  user_id
  created_at
  content
  metadata : Json (contains location information, tags etc).
  post_id

- User table
  user_id
  created_at

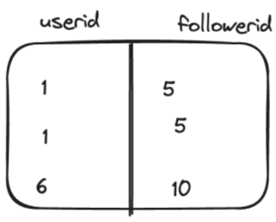
password_hash
user_name
verified_status

Architecture

Fetching follower/following

- getFollowerIds(String userId)
  getFollowingid(String userId)

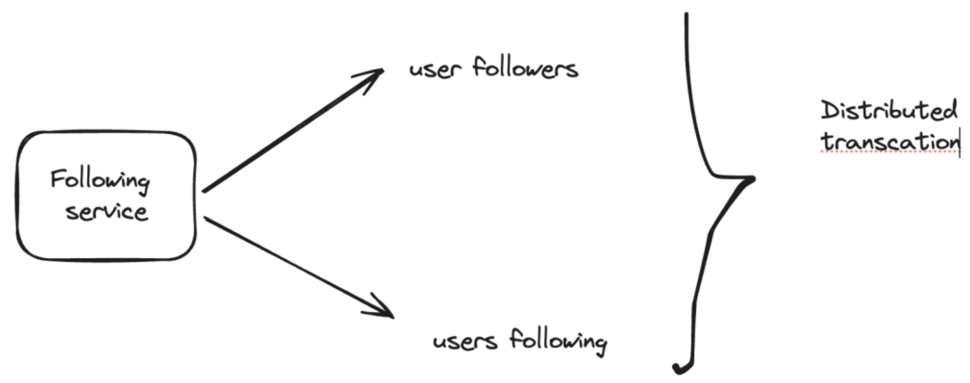
1. For both queries, we want to return the result as quickly as possible.
2. Given the number of users we have, and storage required, we need sharding.



userid	followerid
1	5
1	5
6	10

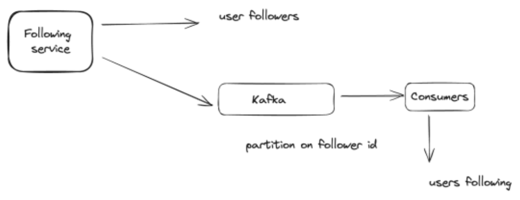
3. Given the above table, if we shard this on userid , getting all the follower ID's will be fast.
4. Problem will be to get all the people I'm following as we need to query all the shard to get the data which can be pretty slow.
5. We need to have both. What will be the approach in this case.
6. Approach 1

Press Escape or Cmd+Enter to finish editing



   Can we update both table in real-time? Yes, but given this will be distributed transaction, it’ll need 2 phase commit which itself is very slow.

7. Approach 2
   Use async processing.

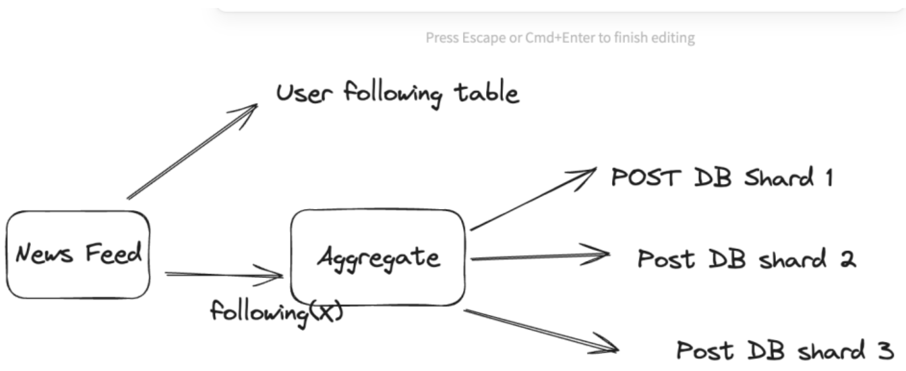


Choice of database in this case?

8. We need good read and write throughput.
9. Can use Cassandra because of following reason.
10. Offer good write throughput because underline storage is LSM trees where writes are first buffered in-memory.
11. Is leaderless so writes can be processed by any node.
12. How to manage the write conflict? Will not matter. They will be merged. If I say node a has info that user x following user y and node b has info that user x following user z, they all be merged where user x is following both.
13. Partition key will be userid.
14. For each partition we can use sort key as following id. Will be beneficial in case of deletion (unfollowed) (search will be faster).

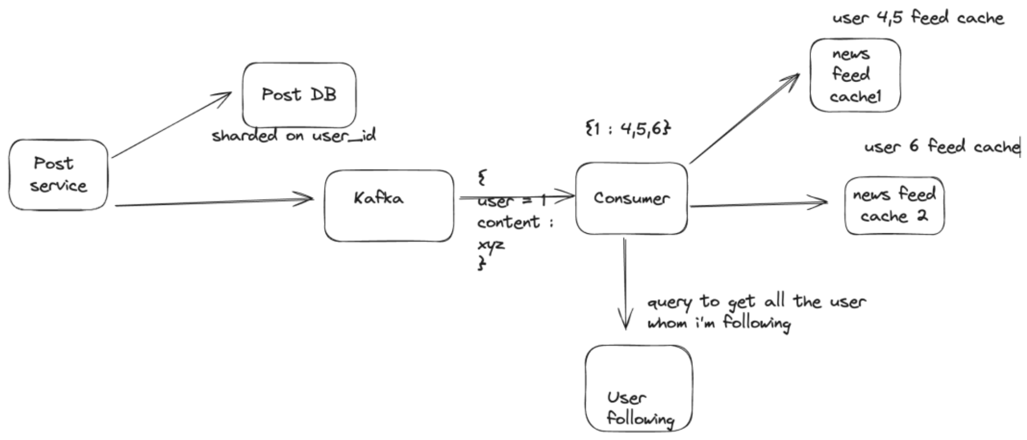
News Feed

- To build a news feed we need following(X) and all the post of (following(X)).
- Approach 1
  Fanout during read



Too slow.

- Approach 2.
  - We need to avoid from reading too many places. What if we know that this post belongs to this news feed. How can we do that.
  - During the write path. What if we can store the news feed of every user.
  - Each post can belong to avg 100 of news feed. Given each post roughly take 200 bytes which will translate to 100*200 bytes = 2 KB.
  - Storage required will be = 1billion post *2KB = 20TB per day. (Create a separate table user news feed sharded on user id and value being list of Json)
  - If we need to speed things up, we can cache the news feed. If we take 256Gb ram, number of machine requires = 2TB/256GB = 80.
  - Drawback of this approach -> Will prepare the cache for inactive users that waste the compute.
  - Architecture



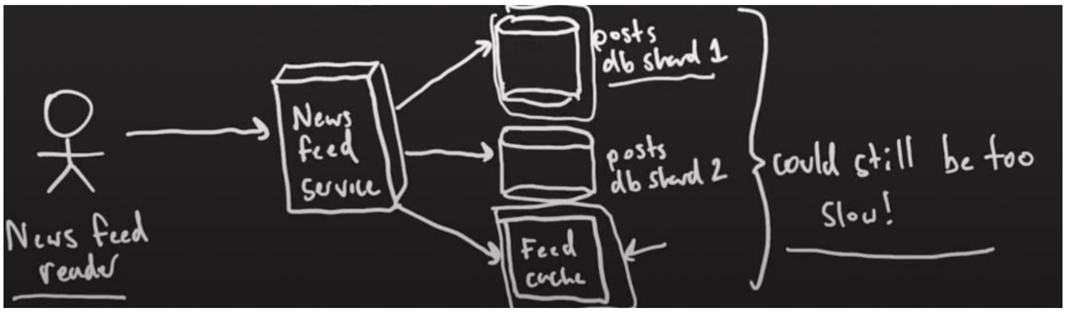
  - **Optional to mention in interview**
    How can we avoid constant call to user following database in the consumer?
    What if the consumer can also consume the user-following Kafka instance. That Kafka can be partitioned on following id (1 -> 2,3,5 i.e. 1 is getting followed by 2,3,5).
    Post service Kafka is partitioned on the user id (user id 1 has posted something).
    This way the consumer can maintain a local state as 1 will be on the same partition always and avoid DB call.

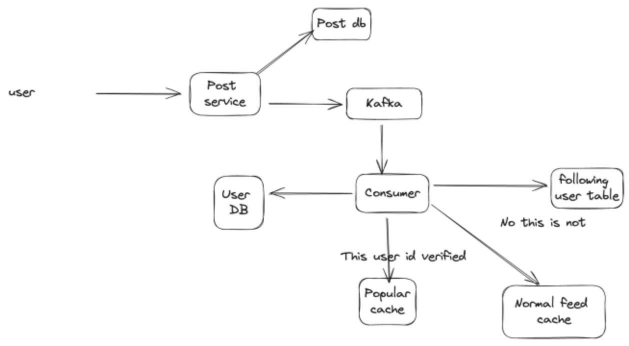
- Database Schema and choice (POST DB)

  - Writes needs to be faster as well as read.
  - Database should be sharded.
  - Can use Cassandra.

- Partition is based on **userid (and not on postid).**
- Sort key can be timestamp
  Userid.   timestamp. content
- Partitioning and sorting make sure that fetching the post of a particular user is very fast.

Celebrity/Popular User

- A popular user can have millions of followers.
- Updating cache using above approach for all the millions of followers is an issue.
- Can we use a hybrid approach.
- For a reader x = post of normal user + post of verified user
- Approach 1
  - For normal user read from normal post.
  - For verified user -> get all the users which we are following (from user following table) and get the verified status from user table.
  - Get the post from the post DB for the verified cases.
  - Still can be very slow.
  - 

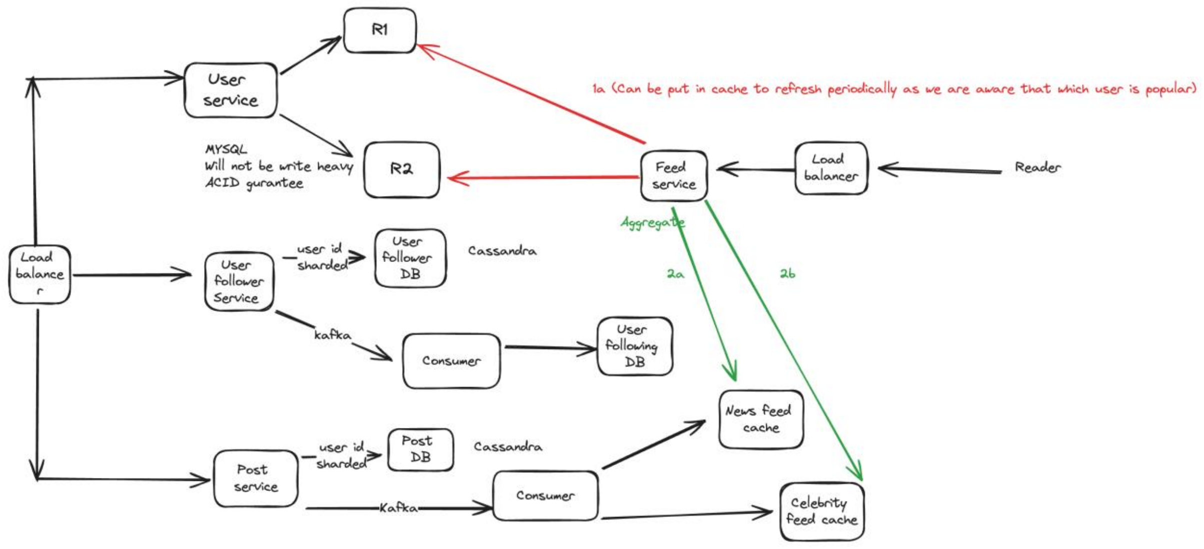
- Approach 2
  - ~~Good thing is we know in advance that the user is popular (verified status e.g.).~~
  - We can create a separate cache for the popular user having the post of popular user only (sharded on user id).
  - 

  - Optional to mention (Don't until ask)
    We already avoided the call to following user table in above case by maintaining the

local state. Same can be done for verified user. User DB changes can be propagated to Kafka partitioned on user id).

- 2nd approach can be we know in advance that the user is popular (verified status e.g.). We can create a cache and refresh this parodically.

Final Architecture



There is one point missing from the feed service. The user following consumer apart from updating the user following db can also update the cache. Cache is going to contain the user id and value can be the list of celebrity user that particular user is following. This way the feed service don't have to query the user following DB (to get the list of following) and then the user service (to get the list of celebrity user) and directly look up the cache.

Also why don't we put the verified update as part of user-following table. Reason being if a particular user is going to be verified in future, we have to update a lot of rows (it's a trade-off).

Also because there will not be any ordering of messages in the post service Kafka consumer, we might need to do that on the fly in the feed service or use something like a sorted set of redis.

Also why are we not using a graph DB ? -> Because there is no traversal involve + whatever as per the tutorial , they are also slow.

Optional to mention (write conflict that might occur when user follow and then unfollows : one write node can say if follows, another can say it didn't) : LWW in this case.

Inspired from