***Capacity Estimation and Constraints***

**Assumptions**

1 Billion total users

DAU day active users -> 20% ->**200M**

**100M tweets per day**

**Avg 200 followers**

**Tweet-Views:** Assume on average a user visits 5 other profiles. On each page if a user reads 20 tweets, total tweet-reads our system will generate:

100M DAU \* ((5\* 20 tweet) => **10B/day**

**Read Heavy**

**Eventual Consistency**

***Requirements and Goals of the System***

**Functional Requirements**

1. Post tweets.
2. Follow users
3. Timeline = TL User Timeline (own tweets) & Home Timeline(The one user follows, default).
4. Tweets can contain photos and videos.

**Non-functional Requirements**

1. Highly available.
2. Low latency
3. Consistency can take a hit (in the interest of availability), if a user doesn’t see a tweet for a while, it should be fine

**Tweet-write:**

Text per day: 140 characters (2Byte/character) + 20 bytes to store metadata for each tweet (like ID, timestamp, user ID, etc.).

100M tweets \* (2\*140 + 20) bytes => 30GB/day =>11 TB per year

Media every day. Not all tweets will have media, let’s assume that on average every 5th tweet has a photo(200KB) and every 10th has a video(2MB).

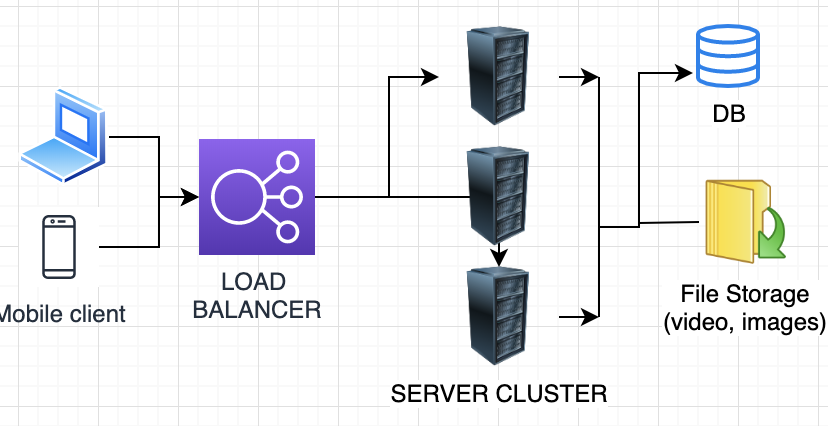
(100M/5 photos \* 200KB) + (100M/10 videos \* 2MB) ~= 24TB/day

***System APIs***

* **POST:** tweet(key, tweetData, userLocation, isMediaAttached) 🡪 url of new tweet
* **GET:** follow(key, tweetId) -> Info about tweet in JSON

**Parameters:**  
key: The API developer key of a registered account. This will be used to throttle users based on their allocated quota. tweetData : The text of the tweet (up to 140 chars) userLocation : Optional location (longitude, latitude) of the user adding the tweet. isMediaAttached : (All the media photo, video, etc.) need to be uploaded separately.

***High Level Design***

******

***DATA STORAGE***

We can store photos in a distributed file storage like [HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop) or [S3](https://en.wikipedia.org/wiki/Amazon_S3).

We need to store relationships between users and photos, to know who owns which photo. We also need to store the list of people a user follows.

For both of these tables, we can use a wide-column datastore like [Cassandra](https://en.wikipedia.org/wiki/Apache_Cassandra). For the ‘UserPhoto’ table, the ‘key’ would be ‘UserID’ and the ‘value’ would be the list of ‘PhotoIDs’ the user owns, stored in different columns. We will have a similar scheme for the ‘UserFollow’ table.

Cassandra or key-value stores in general, always maintain a certain number of replicas to offer reliability.

|  |
| --- |
| **Followers** |
| User1 |
| User2 |

|  |
| --- |
| **User** |
| ID |
| name |
| email |
| dob |
| lastLogin |

***Database schema***

|  |
| --- |
| **Tweet** |
| ID |
| userId |
| Lat |
| Long |
| createDate |

|  |
| --- |
| **Favorites** |
| TWEET\_ID |
| userId |

***Database Sharding***

Since our read load is extremely high too, we need to distribute our data onto multiple machines such that we can read/write it efficiently.

**UserID:** We can try storing all the data of a user on one server. (A-M) on one DB, (N-Z) on another. Famous user like Justin Beiber, Lady Gaga ? There could be a lot of queries on the server holding that user. Unbalanced servers. Solution: use consistent hashing.

**TweetID:** Our hash function will take each TweetID and forward to a random server where we will store that Tweet. solves the problem of hot users, but we have to query all database partitions to find tweets of a user, which can result in higher latencies.  Solution: introduce cache to store hot tweets in front of the database servers.

**Tweet creation time:** advantage of fetching all the top tweets quickly, and we only have to query a very small set of servers. Traffic load will not be distributed, e.g., while writing, all new tweets will be going to one server, and the remaining servers will be sitting idle.

Similarly, while reading, the server holding latest data will have a very high load as compared to servers holding old data.

**Combination of Tweet creation time and TweetId:**

We can use epoch time for this. Let’s say our TweetID will have two parts; the first part will be representing epoch seconds and the second part will be an auto-incrementing sequence. This sequence is difficult to maintain (ZooKeeper)

***Cache***

We can introduce a cache for hot tweets and users. We can use Memcache that can store the whole tweet objects. Application servers before hitting database can quickly check if the cache has desired tweets. Based on clients’ usage pattern we can determine how many cache servers we need.

**Which cache replacement policy would best fit our needs?** When the cache is full, and we want to replace a tweet with a newer/hotter tweet, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system.

**How can we have more intelligent cache?** If we go with 80-20 rule, that is 20% of tweets are generating 80% of read traffic which means that certain tweets are so popular that majority of people read them. This dictates that we can try to cache 20% of daily read volume from each shard.

***Redis AND DB (BOTH)***

User Table | Tweet Table | Follower Table.

DB cannot be scaled, so use in-memory Redis

**How to get timeline:**

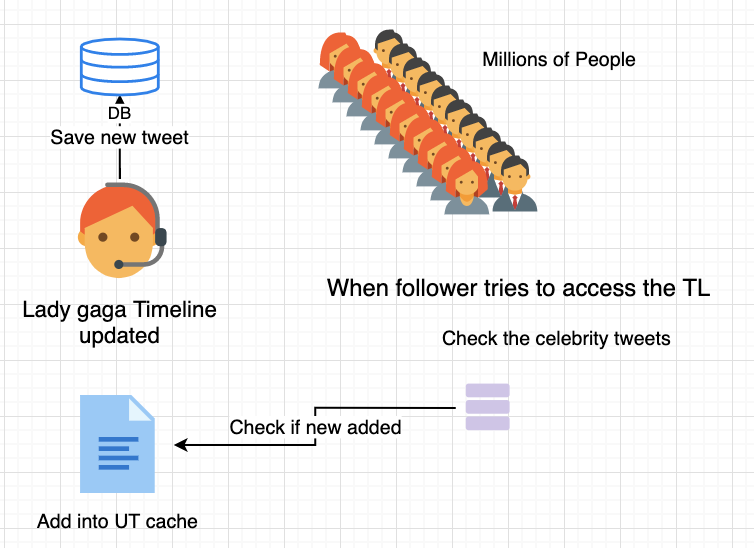
* Get Followers
* Get Latest tweets
* Merge and Display

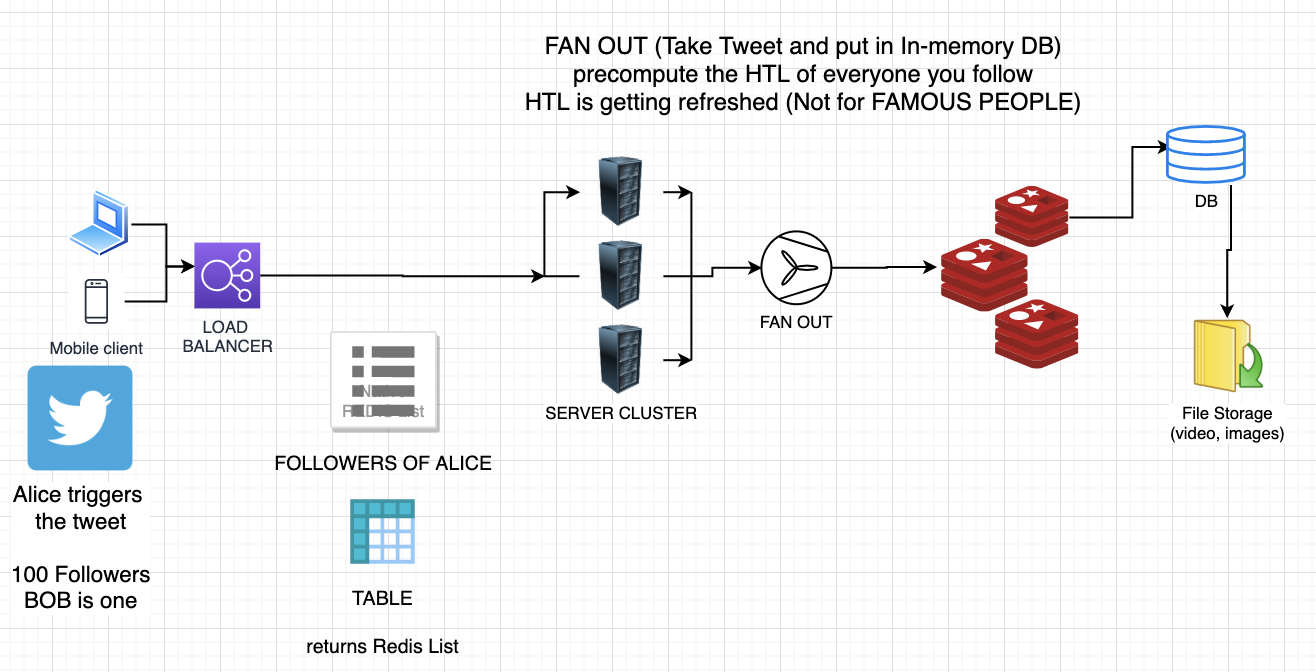
**FanOut:** Pre-processing Home timeline

**Optimization:** who does not login regularly, Avoid computing HTL for them , save lot of computations for Redis.

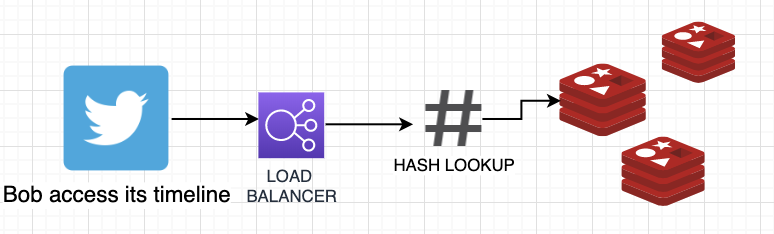
**For hot celebrities (No fan out process is carried)**

Does not work for celebrity with 30 Million followers.

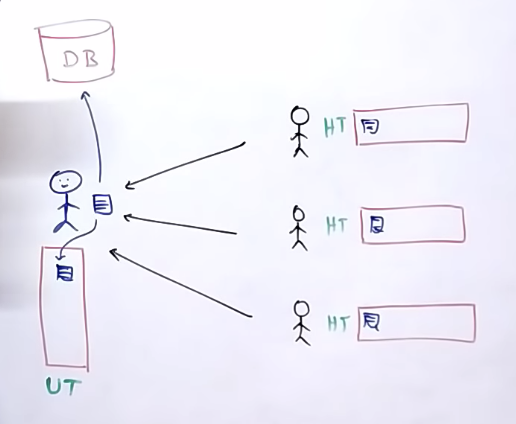




When Bob Access a timeline



**FanOut:** Pre-processing Home timeline



**Trending: Movie release, Elections (Happens in real time)**

1000 tweets in 5 minutes with same hashtag

10000 tweets in 1 month

Apache Storm / Apache Kafka for this

Need to get the data from the Redis cache, For every user cache entry is maintained by Redis (by userId)

Does not work for celebrity with 30 Million followers.

Every user has his HT fetched from REDIS cache