Step 1: Requirements

Functional requirements:
- Users can post new tweets, follow other users, mark tweets as favorites
- Service can create and display a user's timeline of top tweets
- Tweets can contain photos and videos

Non-Functional Requirements
- Highly available, but consistency can take a hit

Step 2: Back of Envelope Calculations

Storage
- 200M users * 5 favorites = 1B favorites
- 200M DAU * 14 tweet-views = 28B/day
- If each tweet has 140 characters each requiring 2 bytes w/o compression:
  - 100M*(280+30)=30GB/day
- If every 5th tweet has a photo and every 10th has a video:
  - 100M/5*200KB+100M/10*2MB=24TB/day

Bandwidth
- 28B*280Bytes/86400+28B/5*200KB/86400+28B/10*2MB/86400=35GB/s

Step 3: System Interface Definition

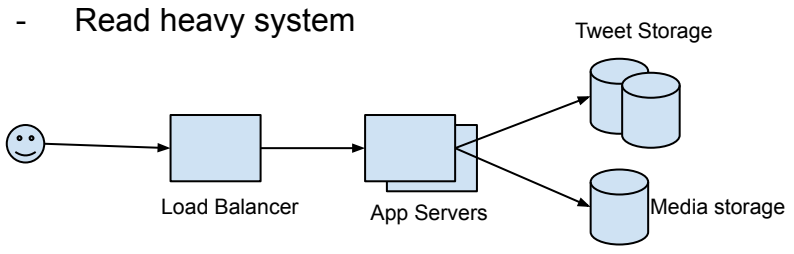
- tweet(api_key, tweet_data, tweet_location, media_ids)

Step 4: Define Data Model

- Tweet (PK: TweetID(int))
- User (PK: UserID(int))
-

Step 5: High-Level Design

- Read heavy system



Step 6: Detailed Design

Shard based on UserID
- Pass UserID to hash function to map user to a DB server
- Store all of user's tweets, favorites, follows etc
- Need to consider hot users which has more load than others

Shard based on TweetID
- Pass TweetID to hash function to map tweet to DB server
- Solves hot users but have to query all DB partitions to find user tweets

Cache
- Cache hot tweets and users
- App servers before hitting DB would check cache first
- Use LRU with 80-20 rule

Load balancing
- Between clients and app servers
- Between app servers and database replicas

Fault tolerance and replication
- Read-heavy so have multiple database servers for each DB partition