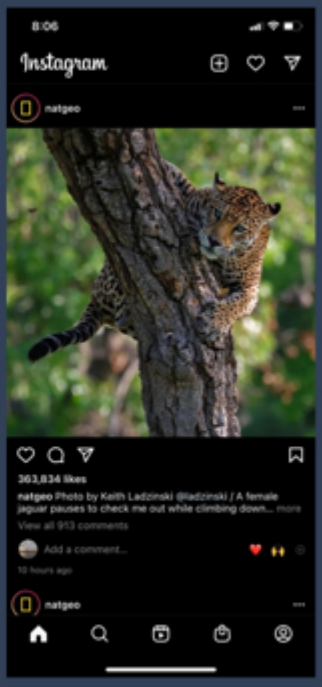
Design Instagram News Feed

Step 1: Gathering Requirements

Functional Requirements:

1. Upload photo(s).
2. View photo(s).
3. Follow/unfollow other users.
4. Generate a news feed, based on who the user follows.



Step 1: Gathering Requirements

Non Functional Requirements

- Service needs to be highly available (prefer availability over consistency).
- Acceptable latency of X milliseconds to generate the news feed, Y milliseconds to view a photo.
- The systems needs to be highly reliable (no lost photos)

1. Total users ~= 1 billion
2. Daily active users ~= 500 million
3. New photos per second ~= 1000
4. Average photo file size ~= 200kb
5. Avg num of timeline fetches a day by each active user ~= 10
6. Photo views per second ~= 500 mil * 10 times a day * 10 per call / 100,000 = 500,000

Identify APIs according to Functional Requirements

1. Users should be able to upload and view photos.
   a. viewPhoto(photo_id, user_id)
   b. viewPhotos(user_id)
   c. postPhoto(user_id, photo, location, time_stamp, title, tags[])

2. Users should be able to follow/unfollow other users.
   a. follow(follower_id, followee_id, time_stamp)
   b. unfollow(follower_id, followee_id, time_stamp)

3. System should generate a news feed for the user, based on who the user follows.
   a. generateFeed(user_id, time_stamp, batch_size, batch_range[])

Data Model

Photo Table

PhotoID (pk)	int64
UserID	int64
Description	varchar64
ImgStorePath	varchar64
ThumbnailPath	varchar64
Latitude	int
Longitude	int
TimeStamp	datetime

UserTable

UserID (pk)	int64
Name	varchar32
Email	varchar32
DOB	datetime
Zipcode	int
CreateTime	datetime
LastLogin	datetime

FollowTable

FollowerID (pk)	int64
FolloweeID (+pk)	int64
TimeStamp	datetime

NewsFeedTable

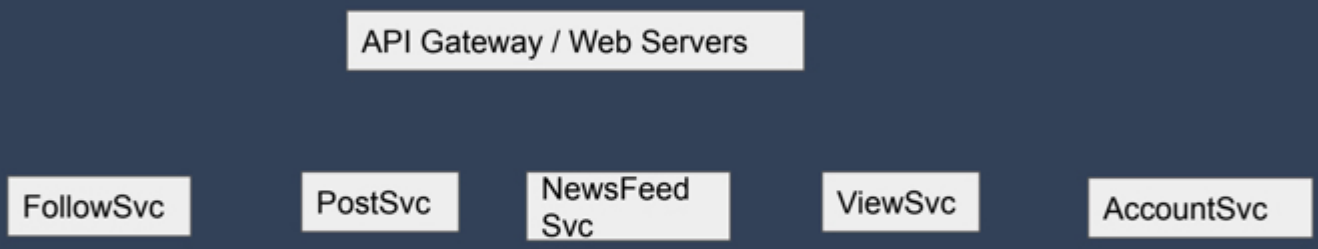
UserID	int64
FeedItemId	int64
PhotoID	int64
TimeStamp	datetime

Step 2: Defining Microservices

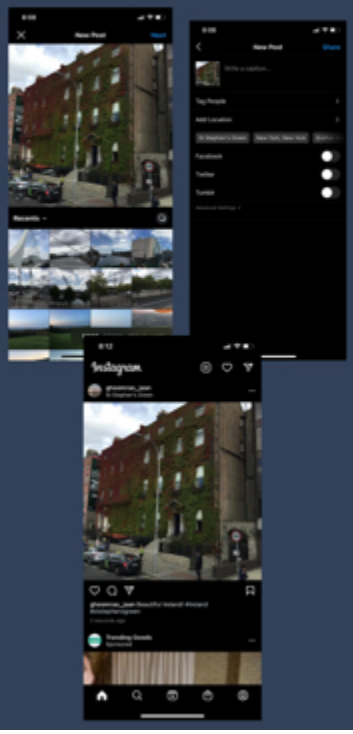
- ViewService
- PostService
- FollowService
- NewsFeedService
- AccountService

Breadth Oriented Problem

Step 3: Logical block diagram



Step 4: PostService



- Tiers: App server tier, Cache or In-memory tier, Storage tier
- Flow of APIs
  - Phone app converts the photo in proper resolution and format
  - App asks the server for a signed URL
  - Server creates meta-data rec in db, returns signed URL
  - App directly stores the img in img store using the signed URL
  - App communicates success/failure to the server
  - Server updates meta-data record
- Data Model: K-V pair
  - K: PhotoID: V: desc, img store path, other properties
- Cache tier - Not needed
- How to store in Storage tier - as a row oriented K-V store
- APIs
  - postPhoto(userID, location, time_stamp, ...)
- Algorithm in APIs
  - Generate photo id (Snowflake or KGS?)
  - Generate signed URL
  - Create(K.V) - CRUD

{ik} INTERVIEW KICKSTART

Step 5: Recall scaling up the solution

- For each microservice
  - Check whether each tier needs to scale
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for storage (storage and cache tiers)
    - Need to scale for throughput (CPU/IO)
    - Need to scale for API parallelization
    - Need to remove hotspots
    - Availability and Geo-distribution
  - The constraints or numbers change from problem to problem
  - Solve algebraically first and then put numbers. Algebraic formula is same for all problems
  - If phase 1 takes more time, do not spend time on calculations unless asked for.

Step 5: PostService:

- For each microservice
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for storage (storage and cache tiers) - Yes
    - Need to scale for throughput (CPU/IO) - Yes
    - Need to scale for API parallelization - No
    - Need to remove hotspots - No
    - Availability and Geo-distribution - Yes
  - Solve algebraically first and then put numbers
    - Total users ~= 1 billion
    - Daily active users ~= 500 million
    - New photos per second ~= 1000
    - Average photo file size ~= 200kb
  - Let's look at each tier separately.

Step 5: PostService:

- App Tier
  - Let t = CPU work time in ms from a single thread
  - So number of operations per second handled by single thread = 1000/t
  - Number of concurrent threads in a commodity server = 100-200 (determined by experiments)
  - So the number of API calls a server can handle per second ~ 100000/t, operating at full capacity.
  - But typical servers operate at 30-40% capacity, so roughly **30000/t API calls per server**.
  -
  - Assuming creating signed URL and later on updating the DB each take ~30 ms.
  - Plug in t = 30 ms in above formula = 30000/30 = 1000 rps per server
  - For 1000 photos per sec, we'll have 2000 operations (1 for signed URL and 1 for updating the DB) ~= 2 servers.

Step 5: PostService:

- DB Tier
  - Image store estimation
  - Total space required for 10 years of photos (w/o replication or growth)
    = 10 years * ~400 days per year * ~100k sec in a day * 1000 pic per s * 200 kb
    = 80,000,000,000,000,000 b = ~80 Pb
  - (more space will be needed for replication)

  - Meta-data estimation
  - PhotoTable: PhotoId(int64) + UserId(int64) + title(varchar 64) + ImgStorePath(varchar 64) + ThumbnailPath (varchar 64) + PhotoLatitude(int) + PhotoLongitude(int) + TimeStamp(datetime) + numLikes(int) + numComments(int)
    = 228 bytes per row * 400 billion photos ~= 100 TB w/o replication.

Step 6: Recall proposing the distributed architecture

- For each microservice
  - If a tier needs scale, scale the tier and propose the distributed system
- This is the most deterministic portion of the solution
- Steps
  - Draw a generic distributed architecture per tier
  - If app server tier and *stateless*, just round robin requests
  - If cache or storage tier
    - Partition the data into *shards* or buckets to suit requirements of scale (changes from problem to problem)
    - Propose algorithm to place shards in servers (*consistent hashing*) (same across all problems)
    - Explain how APIs work in this sharded setting (problem specific)
    - Propose replication (same across all problems)
    - Propose CP or AP (algorithms for CAP theorem do not change from problem to problem)

- This is how each microservice looks within a single data center, either on-premise or VPC in cloud

Step 6: PostService: Reasons for partitioning / sharding

1. If the size of the data is very large
2. If the query throughput is very high (and includes writes)

   PhotoTable ~= 100 TB w/o replication.

   Assuming 1 shard could hold 1TB, we'll need ~100 shards (w/o replication).
   But what should we shard on?

   a. UserID
   b. PhotoID

Step 6: PostService:

Given a key, which node should I go to (ignore replication for now)? And once I am at the right node, how do I access the relevant record?

Option 1: Assign records to nodes randomly. Would this work?

Option 2: Based on key range

Option 3: Based on the “hash of the key” range

Rebalancing partitions

Suppose the load increases, and you add more CPUs to handle the load.

Or suppose the data size increases, and you add more RAM/Disk to store it.

Or suppose a node fails, and other machines need to take over its responsibilities.

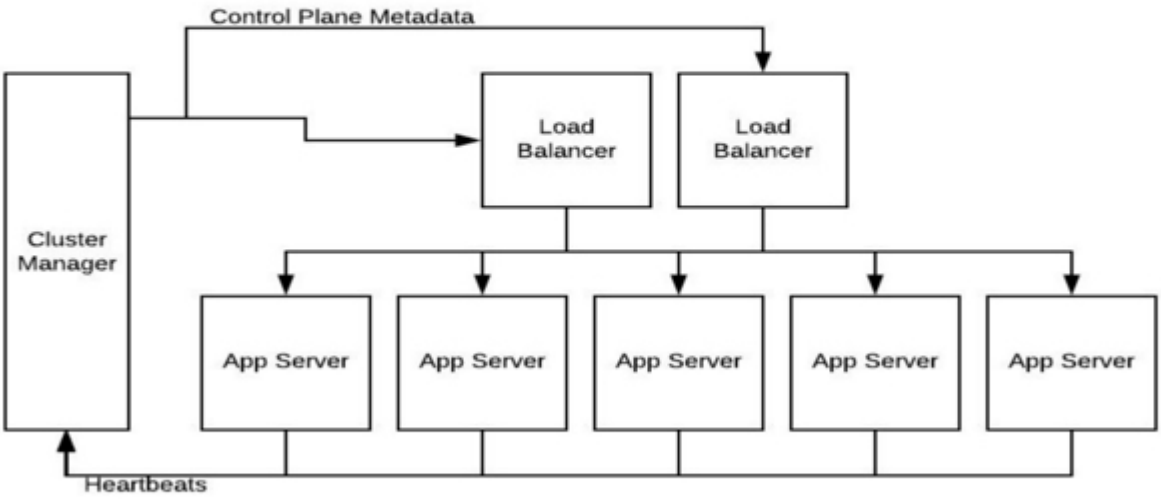
Key ---> Partition ---> Node

If Partition ⇔ Node, then when a node fails, most of the keys need to be moved from node node to another.

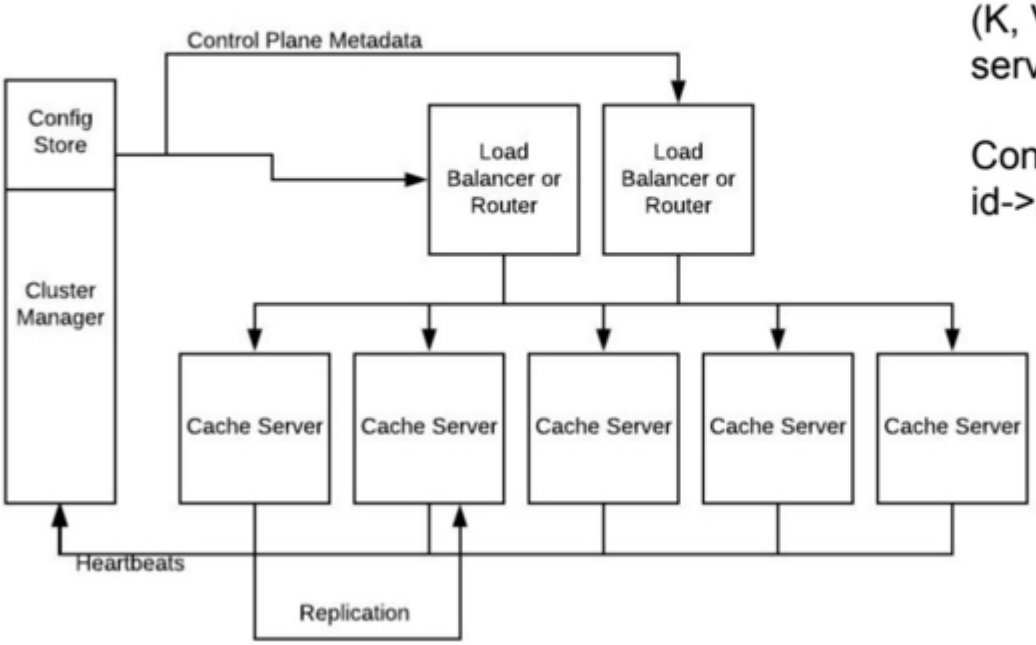
Step 6: PostService

- Sharding
  - Horizontal sharding
  - Map subsets of photos to a single shards
- Placement of shards in servers
  - Consistent hashing
- Replication
  - Yes for availability as well as for throughput
  - To reduce latency for geographically distributed users
- CP or AP?
  - AP: does not need to be strictly nanosecond level consistent

Distributed Architecture - PostService



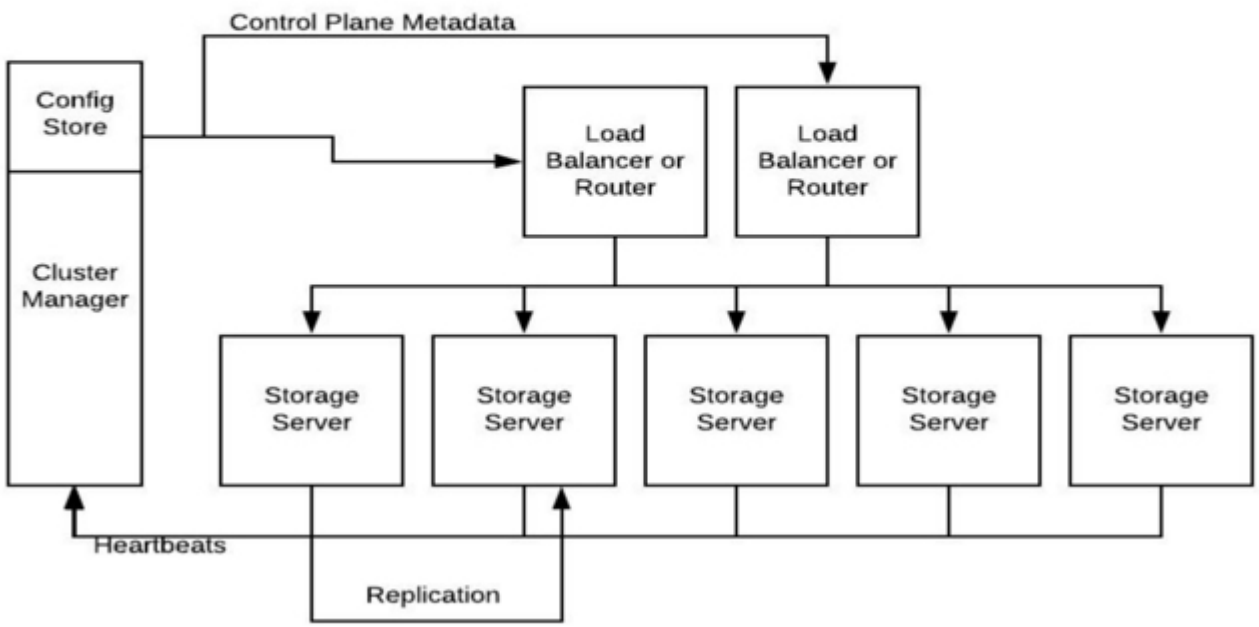
Distributed Architecture - PostService



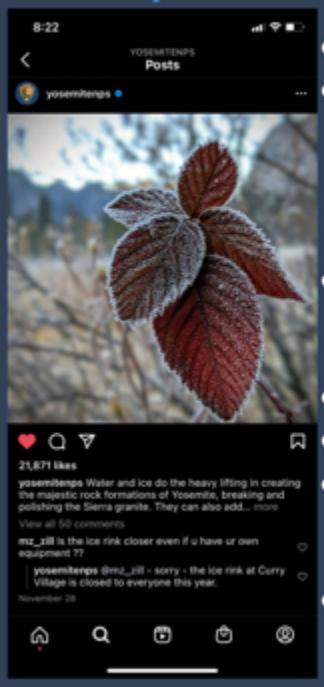
(K, V) -> partition/shard id -> servers

Config store maps shard id->servers

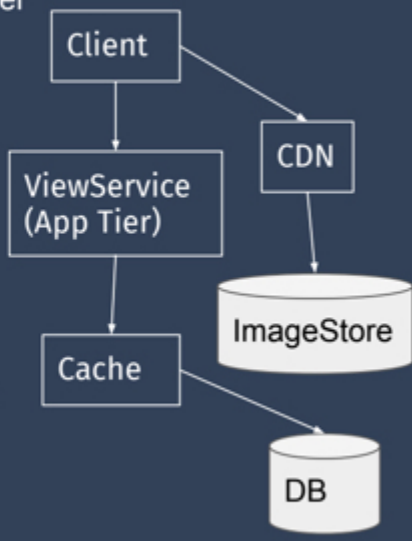
Distributed Architecture - PostService



Step 4: ViewService



- Tiers: App server tier, Cache or In-memory tier, Storage tier
- Flow of APIs
  - Check if meta-data is available in cache
  - If yes, fetch and send it with cdn url
  - If not, read from db, fetch likes, comments
  - Update cache, return to user
- Data Model: K-V pair
  - K: PhotoID: V: desc, img store path, other properties
  - K: UserID: V: a list of photos
- How to store in Cache tier - Hashmap
- How to store in Storage tier - as a row oriented K-V store
- APIs
  - viewPhoto(PhotoID)
  - viewPhotos(UserID)
- Algorithm in APIs
  - Simple K-V workload to fetch basic data (CRUD)
  - Fetch and aggregate like status, comments



{ik} INTERVIEW KICKSTART

Step 5: ViewService: Scale up the solution

- For each microservice
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for storage (storage and cache tiers) - Yes
    - Need to scale for throughput (CPU/IO) - Yes
    - Need to scale for API parallelization - No
    - Need to remove hotspots - No
    - Availability and Geo-distribution - Yes
  - Solve algebraically first and then put numbers
    - Total users ~= 1 billion
    - Daily active users ~= 500 million
    - Photo views per second ~= 20000
    - Average photo file size ~= 200kb
  - Let's look at each tier separately.

Step 5: Need for scale in ViewService:

- App Tier
  - Let t = CPU work time in ms from a single thread
  - So number of operations per second handled by single thread = 1000/t
  - Number of concurrent threads in a commodity server = 100-200 (determined by experiments)
  - So the number of API calls a server can handle per second ~ 100000/t, operating at full capacity.
  - But typical servers operate at 30-40% capacity, so roughly **30000/t API calls per server**.
  -
  - Since we are not sending the photo, but leveraging CDN, let's assume that each photo fetch takes around 60 ms.
  - Plug in t = 60 ms in above formula = 30000/60 = 500 rps per server
  - For 500,000 photos per sec ~= 1000 servers.

Step 5: Need for scale in ViewService:

- Cache Tier
  - Even though we did DB estimation for 10 years, we don't need to store a fraction of that amount in cache.
  - We can store last 5 days worth of photos' metadata in cache
  - So 5 days * 100,000 sec * 1000 pic per s ~= 500 million
  - 500 million * ~200 bytes per pic (meta data) ~= 100 GB
  - Theoretically it could fit in 2 machines with 64 GB RAM

Step 6: Distributed Architecture in ViewService

- Same as before
  - Stateless app servers
  - Shard the data
  - Consistent Hashing
  - CP or AP

Step 4: NewsFeedService

DB Tier

UserNewsFeed table:

UserId	FeedItemId	PhotoId	TimeStamp
1000001	1	20002	8/28/2021-09:10
1000001	2	40003	8/27/2021-16:10
1000002	1	3000001	8/28/2021-08:00

Let's assume we store 3 days worth of news feed items per user, 1000 feeds per day.

How do you cap it at 3 days?

Step 5: Need for Scale in NewsFeedService:

- For each microservice
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for storage (storage and cache tiers) - Yes
    - Need to scale for throughput (CPU/IO) - Yes
    - Need to scale for API parallelization - No
    - Need to remove hotspots - No
    - Availability and Geo-distribution - Yes
  - Solve algebraically first and then put numbers
    - Total users ~= 1 billion
    - Daily active users ~= 500 million
    - Avg num of timeline fetches a day by each active user ~= 10
    - Photo views per second ~= 500,000
    - Average photo file size ~= 200kb
  - Let's look at each tier separately.

Step 5: Need for Scale in NewsFeedService:

- App Tier
  - Timeline fetch req = 500 million daily active users * 10 fetches a day = 5 billion a day
  - = 5,000,000,000 / 100,000 (approx sec in a day) = 50,000
  - Using previous formula, roughly **30000/t API calls per server**.
  - Since we are not sending the photo, but sending a batch of meta-data, let's assume that each fetch takes around 100 ms.
  - Plug in t = 100 ms in above formula = 30000/100 = 300 rps per server
  - For 50000 requests per sec ~= 167 servers.

Step 5: Need for Scale in NewsFeedService:

- Cache Tier
  - Let's assume we store 500 news feed items per active user.
  - So 500 million users * 500 items * 30 bytes per item = 7,500,000,000,000 ~= 7.5 TB
  - For easier calculations, assume 75 GB RAM per server
  - So 7.5 TB / 75 GB ~= 100 servers
  - How do you store it in these server?

Step 5: Need for Scale in NewsFeedService:

- DB Tier
  - Let's assume we store 3 days worth of news feed items per user, 1000 feeds per day.
  - So 1 billion users * 3000 items * 30 bytes per item = 90,000,000,000,000 ~= 90 TB
  - Assume 1 TB per DB shard
  - So 90 TB / 1 TB ~= 90 DB servers

Step 6: Distributed Architecture for NewsFeedService

- For App tier, round robin would suffice unless you are holding the cache on the same servers
- For cache tier, shard based on User ID
- For DB tier as well, shard based on User ID
- Explain how APIs will work in this sharded setting