mecha-mind.medium.com

System Design — Backend for Google Photos - Abhijit Mondal - Medium

Abhijit Mondal

11-14 minutes

source: quanta magazine

> Design the backend system for storing and retrieving Google Photos

Similar Problems

- Design system to store and fetch Instagram Photos.
- Design Youtube

Functional Requirements

- Users should be able to upload, download and delete photos
- Users should be able to view his/her own photos
- Users should be able to filter photos on a range of dates/timestamps.

Non Functional Requirements

- **Durable** — Uploaded photos should not get lost.
- **Available** — Users should be to upload or view photos any time.
- **Consistent** — Users should be able to view fully uploaded photos immediately.
- **Latency** — Photos should be fetched with minimal latency.

Traffic and Throughput Requirements

- Total number of users = 1 billion
- Average size of photo = 500KB
- Number of read queries per second = 1 million QPS
- Number of uploads per day = 10 million photos
- P99 latency for loading photos page = 300 ms
- Read Throughput = 500KB*1 million/s = 477 GB/s
- Write Throughput = 10million*500KB/day = 4.7 TB/day

The “On My Computer” Approach

HashMap + Binary Search

> Store the photos in filesystem. (Lets assume that the filesystem allows unlimited files in a directory).

Maintain one HashMap for mapping user_id to photo_ids.

Instead of maintaining separate timestamp field for photos, we can use the timestamp in photo_id, i.e.

photo_id = timestamp in seconds (32 bits) + 10 bit integer M

The integer M is the number of photos uploaded per second. It is assumed that there will at maximum **1024 photos uploaded per second**.

Thus the id can support maximum of 2^{42} photos which is about 4.4 trillion.

Since the number of users is 1 billion and assuming that the user base will increase to 5 billion, thus number of bits required to store user_id is $\log_2(5*10^9) = 33$.

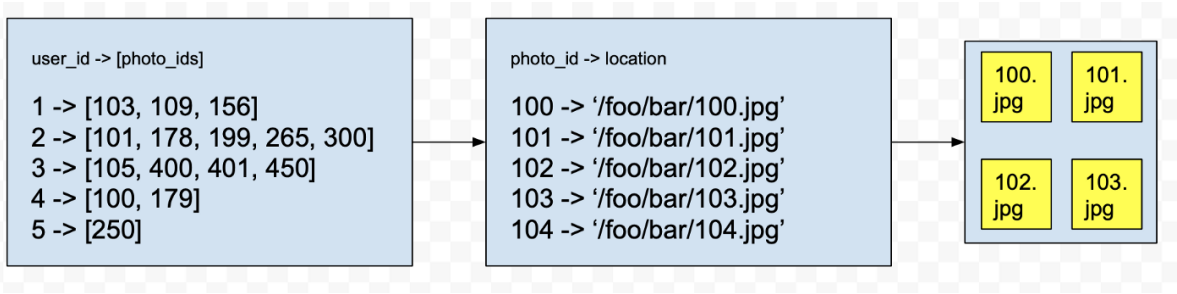
Maintain a HashMap where the key is the user_id and value is a sorted list of photo_ids for the user. Note that since photo_ids are already time sorted and new photo_ids have value > old photo_ids, thus we do not need to maintain any balanced BST but just appending to the end of the list is sufficient.

Maintain another HashMap for mapping photo_id to location of photo on disk.

For a query to fetch photos, using the user_id, fetch all the photo_ids for user_id from the first HashMap.

> Then do a Binary Search to find the photo_id at index ‘i’ with timestamp part ≥ startTime and the photo_id at index ‘j’ with timestamp part ≤ endTime. Then fetch all the photos from the

> filesystem with photo_ids in the index range i to j i.e. photo_ids[i:j+1].



Assuming each photo_id is 42 bits and each location is 20 bytes i.e. 160 bits and number of photos uploaded in 5 years = 5*365*10 million = 20 billion.

Thus size of the second HashMap (photo_id->location)= 20 billion * (160+42) bits = 470 GB.

Size of the first HashMap (user_id->[photo_ids]) = 20 billion*42 bits + 5 billion * 33 bits = 117GB.

Total memory required = 470+117 = 587GB.

Total size of photos on disk = 20billion * 500KB = 10PB.

This approach has several drawbacks:

- The memory required is 587GB, much higher than what can be supported on a 16GB system. Also the disk space required is 10PB whereas most systems comes with 1TB HDD.
- On system crash, the data in HashMap will be erased unless they are persisted on a local database. This violates durability.
- Maximum QPS can be served from this server = 8 cores/300ms = 26.67. But expected QPS is 1 million.

Using Database, Filesystem and In-memory HashMap

One of the important trade-off is how and where to store the photos. Two obvious choices are **Filesystem and Database**.

Database —

- If the photos are shared and updated often then in order to handle concurrency we need the ACID properties of relational DBs. But in our requirement photos are nor shared nor updated once they are uploaded.
- Photos have to be converted to either blobs or base64 encoding before storing them in DB. This has performance bottleneck during reads as these photos also needs to be decoded before sending to client.
- Expensive as compared to Filesystems.
- We do not require any of the database indexing features on photos nor can we query photos efficiently.

> Storing images in filesystem seems to be the better choice as most modern filesystems such as **XFS** allows very fast reads.

Since most images are of small size and storing lots of small images on filesystem has a drawback that the inode table will have lots of entries. One optimization that can be done is to **combine multiple photos in a single file** and maintain an in-memory HashMap of photo_id to (location, offset).

Assuming each filesystem server can store upto 10TB of photos and total size of all photos is 10PB, we would need 1024 filesystem servers to store all the photos.

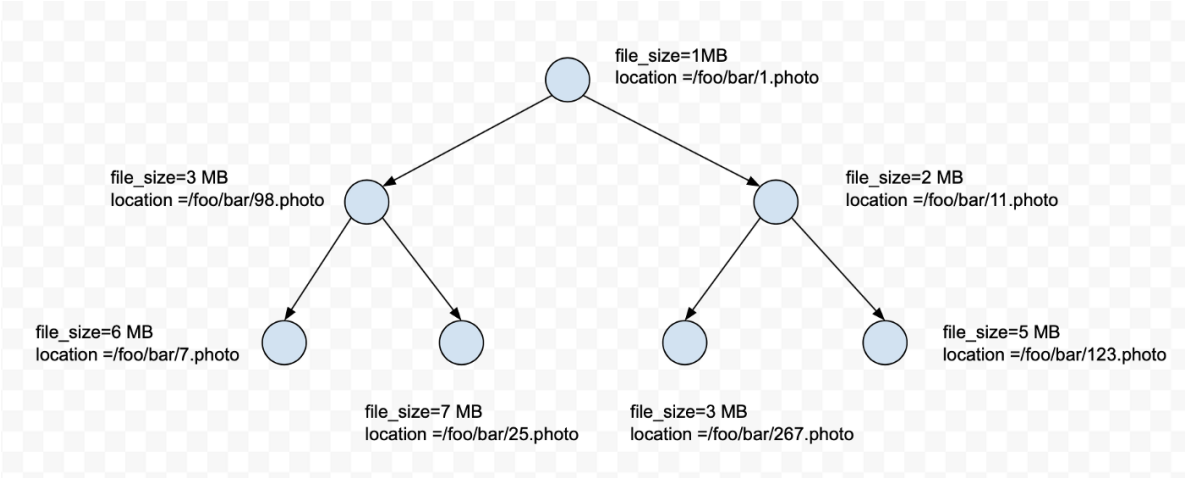
With 10TB photos in single instance, the number of photos is approximately = 10TB/500KB = 21 million.

If we maintain a HashMap from photo_id to (location, offset) where we assume that location is 160bits and offset is 4 bytes = 32 bits, thus total size of HashMap = 21million *(42+160+32) = 586MB.

Thus we can store an in-memory HashMap quite easily with the Filesystem servers.

How does system know which file to write the photo (bytes) in ?

> Maintain a min-heap of the current file sizes and file locations. For a new photo, calculate its size and then fetch the file with the lowest size (i.e. root of heap). If the lowest size + photo_size > max_size, then create a new file and add the photo in that file and add it in the heap, else add the photo bytes in the file location at the root of heap.



What happens when the Filesystem server restarts ?

If it restarts although the photos will be saved in the filesystem but the HashMap and the MinHeap will be lost. Those need to be reconstructed with the data on filesystem by scanning all files.

This could take a long time depending on how many files are there.

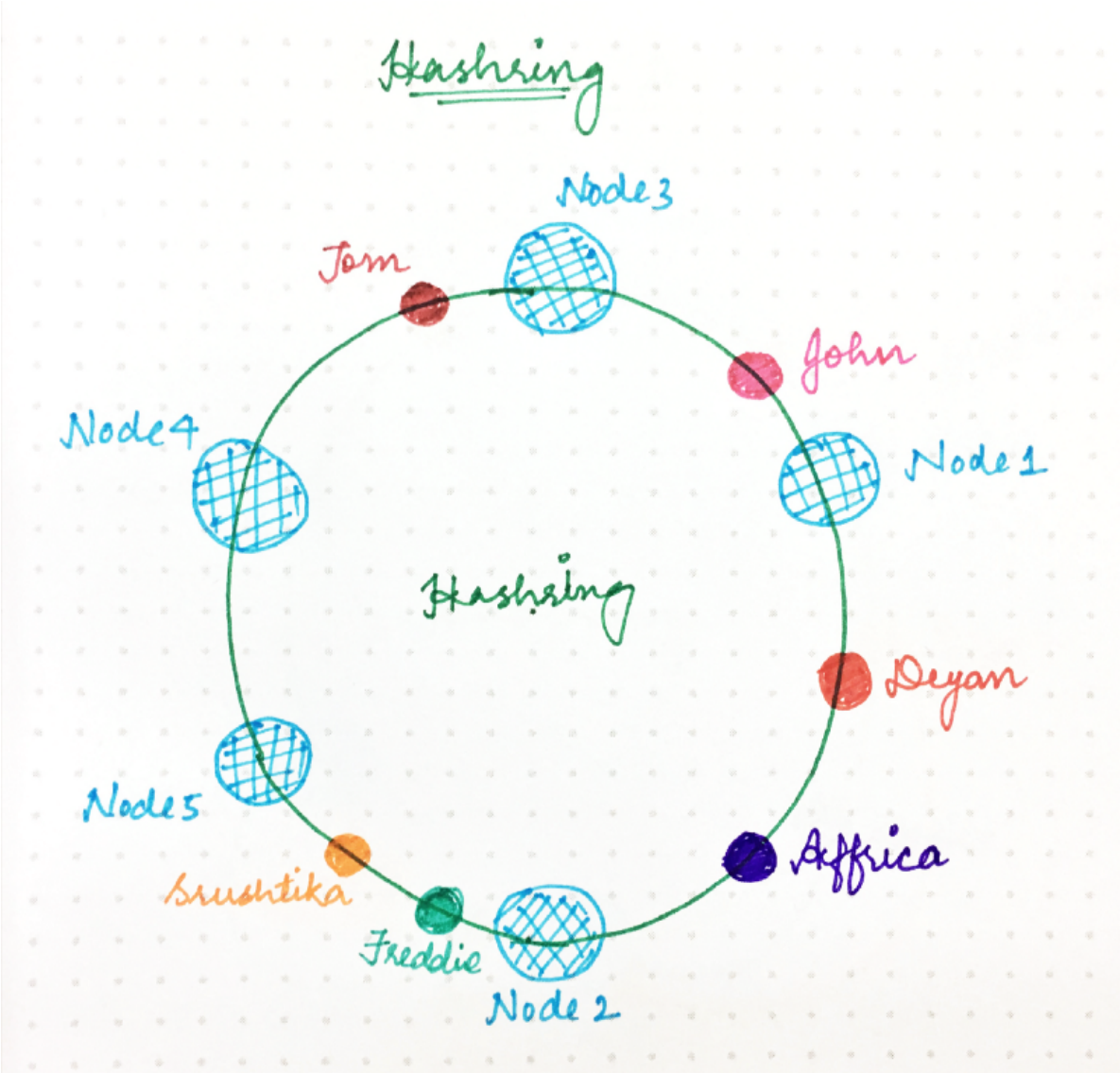
How does the system knows in which server a photo is stored

?

> One solution is to assign a server for a photo_id using random hash functions i.e. hash(photo_id) % num_servers.

What if we have to add a new server or one of the existing servers crashes ?

> Use Consistent Hashing.



source: highscalability.com

How to handle fault tolerance ?

> For each photo_id, instead of storing it in a single filesystem server we store it in 3 different servers. Thus instead of 1024

> servers we need 3072 servers. In the consistent hashing ring, for each photo_id, choose the next 3 servers in the clockwise direction.

Care must be taken such that if any of the server dies, the photos will be assigned to the next server in clockwise direction but it will already have those photos due to replication. So **we should skip the next 2 servers each time a crash happens**.

Before calling the Filesystem service, we also need to store the user_id to photo_ids mapping. **We can either store it in Database table or in-memory cache such as Redis.**

> To account for availability i.e. any uploaded photos should not get lost, if we use complete in-memory db, then with system restart, data would be lost and it will not be possible to know which photo_ids belong to user_id even if the photos are stored in Filesystem servers.

Thus we store this data in Database.

(user_id, photo_id)

> The partition key would be the user_id and range key is photo_id.

Since the size of this table is only 117GB (see above), it can be easily stored under one DB instance and we could maintain 2 more replicas for fault tolerance.

How to handle hot partition issue ?

Since the partition key is the user_id, each user_id would be assigned a logical partition, thus if only a few users are very active then there would be many requests for only a few partitions and thus it could effect read performance.

> One solution is to use photo_id as the partition key.

Then to fetch all photos for an user_id, we have to query multiple partitions and then aggregate the results from all partitions using map-reduce. Since the final result will have photos sorted by timestamp, the results from each partition needs to be aggregated using merge sort.

> Another solution is to use write-through cache.

We can have a Redis LRU Cache which is written before writing to the DB, thus the cache always contains latest data. But we also need to scale the Redis cache here since we need to have the data partitioned across multiple Redis instances, as a single instance will not likely accommodate 117GB data.

Number of Redis instances assuming 16GB RAM = 117/16 = 8

How about scaling the high volume of reads and writes ?

Assuming that each Filesystem server can have a throughput of 100MB/s and we have 3072 servers thus we can serve 300GB/s, but our read throughput requirement is 477GB/s. Thus we need approximately 2000 more Filesystem servers i.e. around 5000 instances with 10TB disk capacity.

> Thus the full data can be partitioned across 1667 instances whereas each partition is stored in 3 replicas.

Assuming each Redis instance can handle throughput of 4GB/s, thus with 8 instance we can handle 32GB/s read throughput. But to handle 477GB/s, we need to have more Redis instances behind load balancers.

> Approximately 120 instances i.e. full data partitioned across 40 instances and each partition is replicated across 3 instances.

Similar to the Filesystem servers, the Redis instance for a particular user_id can be obtained by maintaining a consistent hashing table.

How to handle consistent reads/writes ?

> Whenever a photo is uploaded, it must successfully update all the 3 Redis cache replicas for the user_id as well as all 3 Filesystem server replicas for that photo_id and only then a success message is sent to the user.

But this could lead to lots of performance issues during writes.

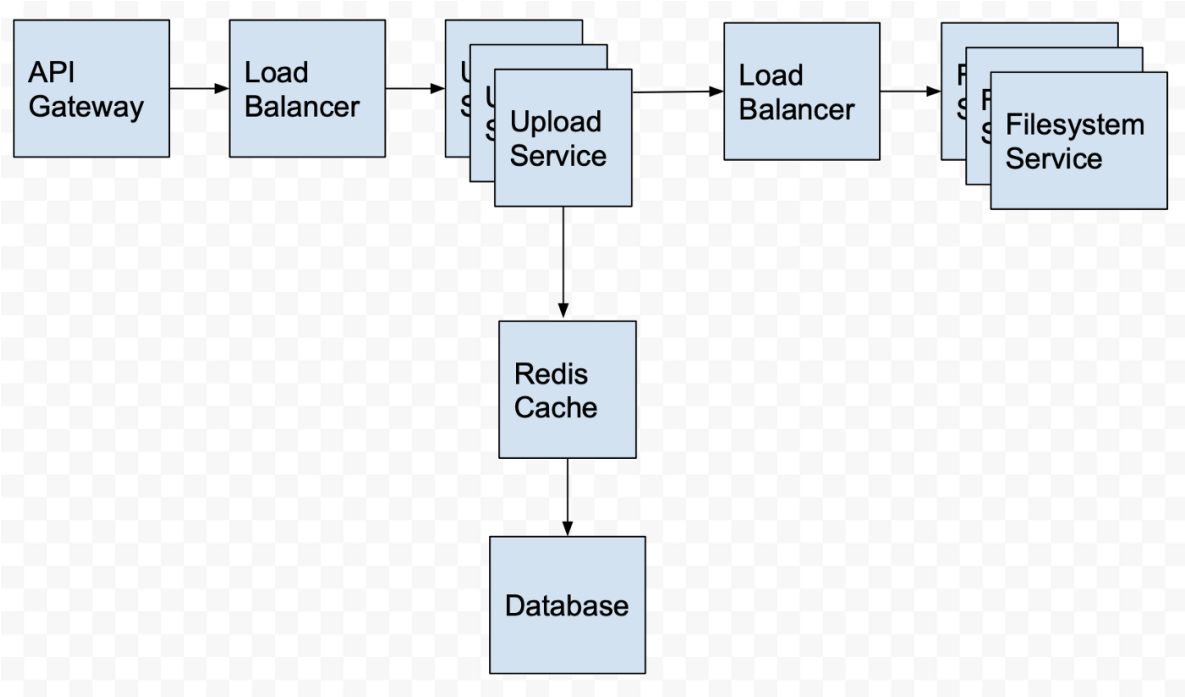
What if some replica instances crashes or becomes unresponsive due to load ?

Other solution is to probably use a **quorum**. For e.g. out of 3 instances, a write is successful only when at-least 2 out of 3 instances sends their ack back and for reads the response with the highest photo_id value (i.e. most recent timestamp) among the first 2 instances to send response back is considered.

> We chose 2 for both read and write because, there will be at-least 1 instance which was used both during write and read flow since 2+2=4 > 3. (something similar to Pigeonhole Principle).

The above solution is good for optimizing writes but suffers for read because earlier when we are successfully updating all 3 replicas, read can happen from any replica but now with write successful on only 2 replicas, read has to happen from 2 replicas instead of one. Thus request needs to be broadcasted as well as aggregated to and from 2 instances.

The Pipeline



Useful Resources

- https://www.quora.com/Which-would-be-the-best-database-for-storing-images-and-videos-in-large-numbers
- https://www.quora.com/Why-is-it-considered-bad-to-store-images-in-a-database
- https://stackoverflow.com/questions/446358/storing-a-large-number-of-images
- https://stackoverflow.com/questions/11239790/storing-large-number-of-images-database-or-filesystem
- https://stackoverflow.com/questions/11239790/storing-large-number-of-images-database-or-filesystem
- https://stackoverflow.com/questions/3748/storing-images-in-db-yea-or-nay
- https://perspectives.mvdirona.com/2008/06/facebook-needle-in-a-haystack-efficient-storage-of-billions-of-photos/

- https://serverfault.com/questions/95444/storing-a-million-images-in-the-filesystem/161014#161014
- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/ch-xfs
- https://stackoverflow.com/questions/466521/how-many-files-can-i-put-in-a-directory
- https://www.toptal.com/big-data/consistent-hashing
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning
- https://docs.oracle.com/database/121/VLDBG/GUID-6CE884AF-84A4-4E6A-A3EF-DCCEBCAB2DB2.htm#VLDBG00202
- https://cseweb.ucsd.edu/classes/sp16/cse291-e/applications/ln/lecture7.html