Designing Instagram: System Design Interview Question

Instagram is a social media platform that enables users to share photos and videos with others. On the platform, content creators can choose to set the visibility of their posts (photos/videos) to either private or public, and other users can interact with the posts by liking or commenting on them.

Instagram also offers a range of other functionality, including the ability for users to follow each other, see each other's news feeds, and search for content across the entire platform. Other features available on Instagram include image editing, location tagging, private messaging, push alerts, group messaging, hashtags, filters, and more.

In this blog, we will look at how to design a simplified version of the Instagram system with features such as photo sharing, following, and news feeds.

Requirements of the system

Functional requirements

- Users should be able to upload and view photos.
- The system should allow users to search for photos based on their titles.
- Users should be able to follow each other.
- Each user should have a custom news feed that displays the top photos from the users they follow.

Non functional requirements

- The system should prioritize high availability and low latency while viewing photos. To improve availability, we can trade off consistency, meaning it's acceptable if a user doesn't see an image immediately.

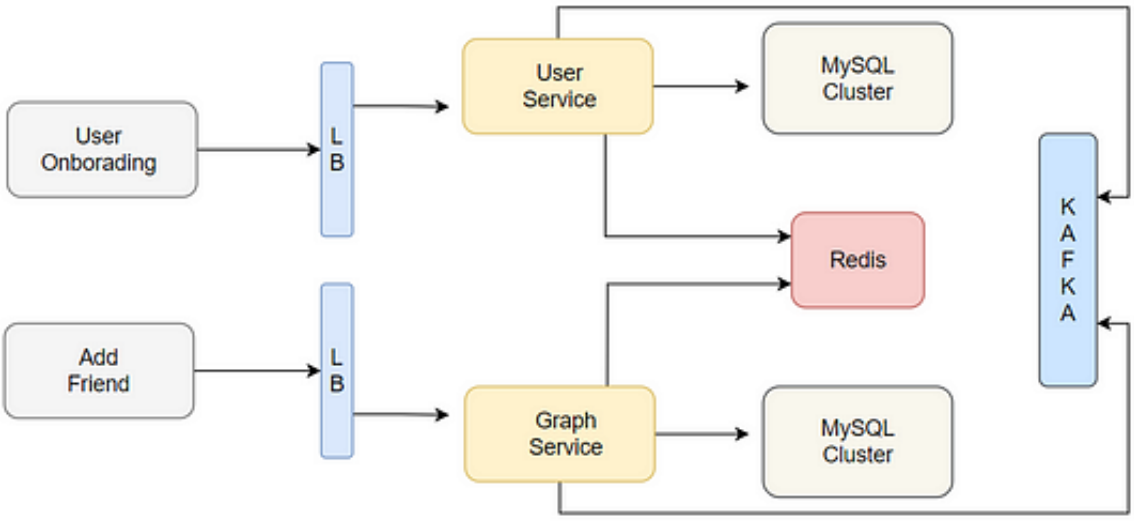
- The system should be highly scalable and optimized for read-heavy workloads, with a high read-to-write ratio.
- The system should be reliable, ensuring that no uploaded photos or videos are lost.
- The system should be optimized for accessing popular posts.
- The system should be compatible with a wide range of devices, support multiple languages, and work well with varying internet bandwidths.

Capacity estimation

It's important to consider that read requests will be significantly more frequent than write requests, with a ratio of approximately 100 to 1.

- Let's assume there are 500 million users registered on the platform, with 1 million active users per day.
- If 5 million images are posted daily, this translates to an average of 57 photos being uploaded per second (5M / (246060)).
- If the average photo size is 150 KB, then the daily storage usage is 716 GB (5M * 150KB).
- If we assume the service will be active for ten years, the total space required will be approximately 2.6 PB (716GB * 365 * 10).

High Level Design



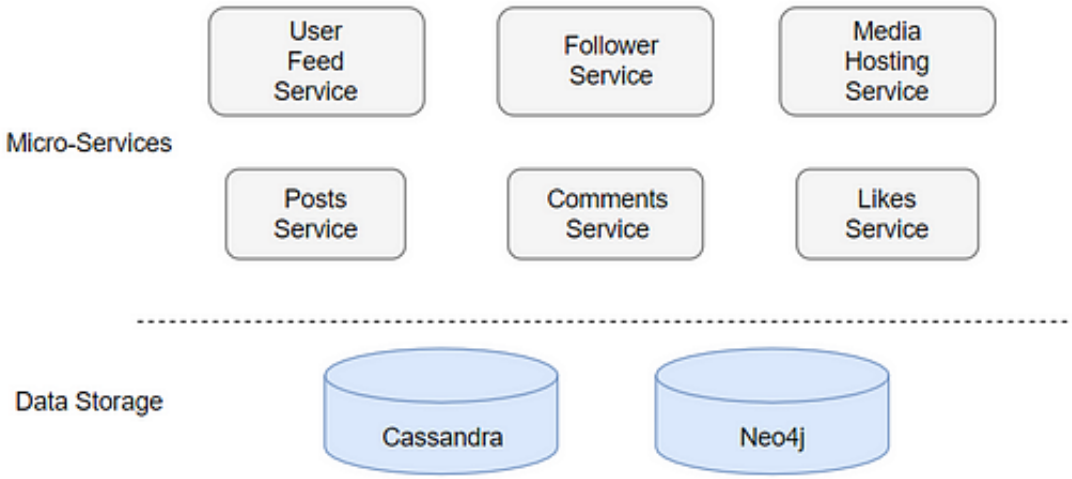
The user service is responsible for managing user onboarding, login, and profile-related actions. It runs on a MySQL database, which was selected because the data is structured in a relatively relational way and the system is optimized for read-heavy workloads, which MySQL is well-suited to handle.

The user service is connected to a Redis database, which stores all user data. When the user service receives a request, it first checks Redis for the requested information and returns it to the user if it is found. If the data is not present in Redis, the user service will check the MySQL database, retrieve the data, and insert it into Redis before returning it to the user. Additionally, a similar process can be followed whenever new users or information are added to the database.

This approach allows the user service to quickly access and returns the most up-to-date data to the user while maintaining a cache of frequently accessed information in Redis to improve performance.

System Components

Our system will be composed of multiple microservices, each responsible for a specific task. To store the data, we will use a graph database like Neo4j. We've chosen this data model because our data contains complex relationships between elements like users, posts, and comments, which can be represented as nodes in the graph. The edges of the graph can be used to record relationships like follows, likes, and comments. We may also use columnar databases like Cassandra to store information such as user feeds, activities, etc.



Data flow and API design

Data flow

1. The user sends a request to the API.
2. The load balancer receives the request and forwards it to an app server.
3. The app server receives the request and performs input validation.
4. If the input is valid, the app server attempts to fulfill the request.
5. If successful, the app server returns an OK response with or without the requested data. If there is an issue, it returns a specified error response.

API design

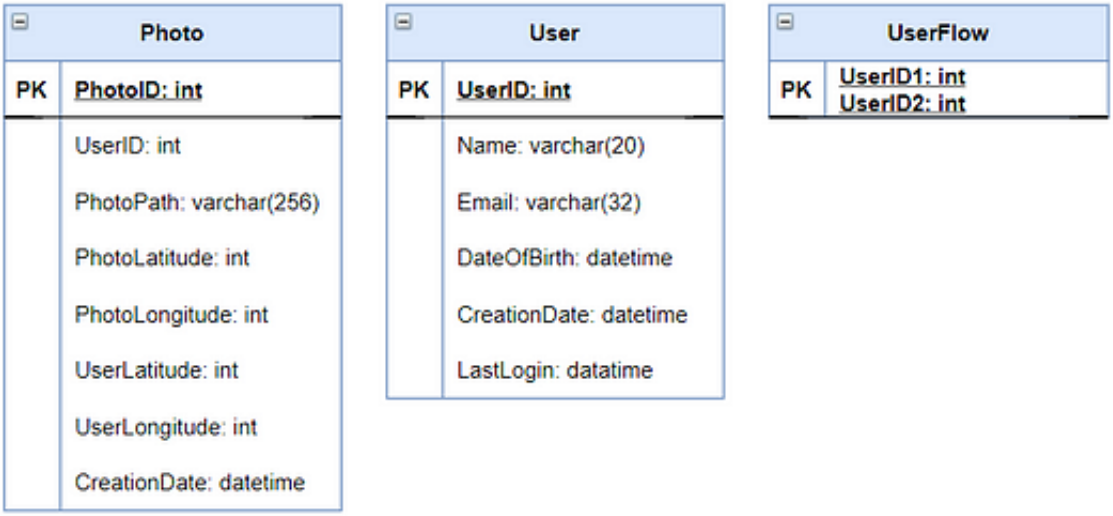
- **signup** (username, first*name, last*name, salted*password*hash, phone_number, email, bio, photo): adds the user to the user table
- **login** (username, salted*password*hash): logs in the user and updates the last login time
- **search_user** (search*string, auth*token): returns public user data for the given search string (can be searched in user first name, last name, and username)
- **get*user*by_id**(user*id, auth*token): returns public user data for the given user ID
- **follow_user**(user*id, target*user*id, auth*token): adds follow data to the

database
- **add_post**(file, caption, user*id, auth*token): uploads the file to the file storage server
- **delete_post**(user*id, post*id, auth_token): deletes the given user's given post along with its metadata (using soft delete)
- **get_feed**(user*id, count, offset, timestamp, auth*token): returns the top posts after the given timestamp of users followed by the given user according to count and offset
- **get*user*posts**(user*id, count, offset, auth*token): returns the posts of the given user according to count and offset
- **post_like**(user*id, post*id, auth_token): adds the given post ID to the given user's likes
- **post_unlike**(user*id, post*id, auth_token): removes the given post ID from the given user's likes
- **add_comment**(user*id, post*id, comment): adds a comment to the given user's comment on the given post
- **delete_comment**(user*id, comment*id): deletes the given user's comment with the given comment ID

Database Design

It's important to clearly define the database structure early in the interview process to help understand the flow of data between different components and determine how to segment the data.

We need to store data about users, their posted images, and the people they follow. To efficiently retrieve recent photos from the photo table, which stores all data related to a photo, we will create an index on (PhotoID, CreationDate).



One option for storing the data described above, which requires joins, is to use a relational database management system (RDBMS) like MySQL. However, RDBMSs can have challenges with scaling. Instead, we could store photos in a distributed file system like HDFS or S3.

To take advantage of NoSQL's features, we can store the schema described above in a distributed key-value store. We can create a table with a "key" of "PhotoID" and a "value" of an object containing all the metadata for the photo, such as PhotoLocation, UserLocation, CreationTimestamp, etc. This allows us to store and retrieve data using a simple key-value interface.

To know who owns which photo, we need to store relationships between users and pictures. We also need to keep track of who a user follows. We can use a wide-column datastore like Cassandra 28 for both of these tables. The ‘key’ for the ‘UserPhoto’ table would be ‘UserID,’ and the ‘value’ would be the user’s list of ‘PhotoIDs,’ kept in distinct columns. The ‘UserFollow’ table will follow a similar pattern.

Like other key-value stores, Cassandra maintains a set number of replicas to ensure reliability. Deletes are also not implemented immediately in key-value stores, as data is typically retained for a certain number of days to allow for undeletion before being permanently erased from the system. This helps to ensure data consistency and recoverability.

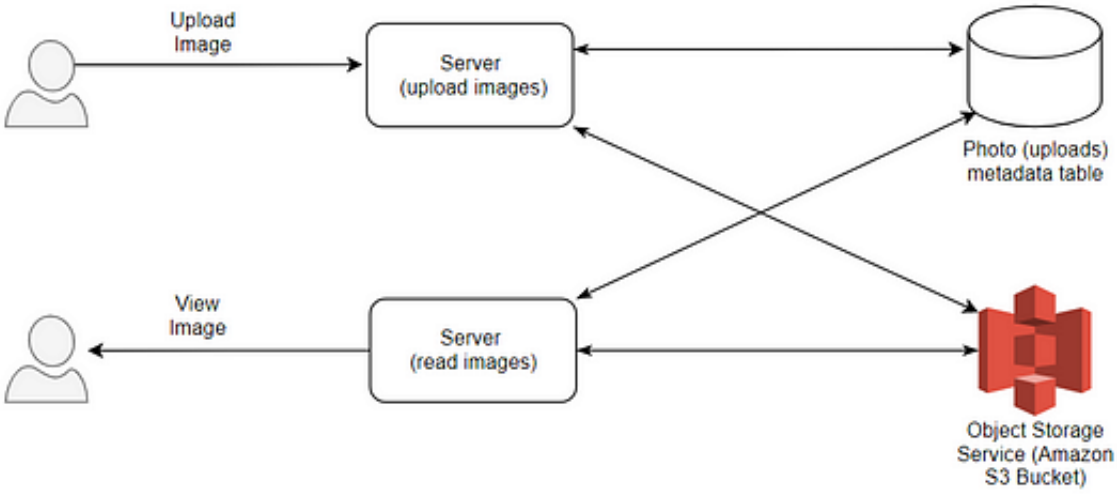
News Feed Generation

Generating news feed

Designing a customized newsfeed for each user that showcases the most recent post from each user they are following is a critical aspect of an Instagram-like service. For the sake of simplicity, let's assume that each user and their followers upload 200 unique photos per day. This means that a user's newsfeed will consist of a combination of these 200 unique photographs, followed by the reputation of previous submissions. This allows the user to see the most recent and relevant content from the users they follow.

To generate a news feed for a user, we will first retrieve the metadata (such as likes, comments, time, location, etc.) of the most recent 200 photographs and pass it to a ranking algorithm. This algorithm will use the metadata to determine the order in which the photos should be displayed in the news feed. This allows the user to see the most relevant and engaging content at the top of their feed.

One disadvantage of the news feed generation approach described above is that it requires simultaneously querying a large number of tables and ranking them based on predefined criteria. This can result in higher latency, meaning it takes a longer time to generate a news feed. To improve performance, we may need to optimize the queries and ranking algorithms, or consider alternative approaches such as pre-computing and caching the results.



To address the latency issues with the news feed generation algorithm described above, we can set up a server that pre-generates a unique news feed for each user and stores it in a separate news feed table. When a user wants to access their news feed, we can simply query this table to retrieve the most recent content. This approach reduces the need to query and rank a large number of tables in real-time, improving the performance and responsiveness of the system.

Serving the news feed

We have now discussed how to create a news feed. The next challenge in designing the architecture of an Instagram-like service is determining how to deliver the generated news feed to users.

One approach is to use a **push** mechanism, where the server alerts all of a user's followers whenever they upload a new photo. This can be done using a technique called long-polling. However, this approach may be inefficient if a user follows a large number of people, as the server would need to push updates and deliver notifications frequently.

An alternative approach is to use a **pull** mechanism, where users refresh their news feeds (send a request to the server) to see new content. However, this can be problematic because new posts may not be visible until the user refreshes, and many refreshes may return empty results.

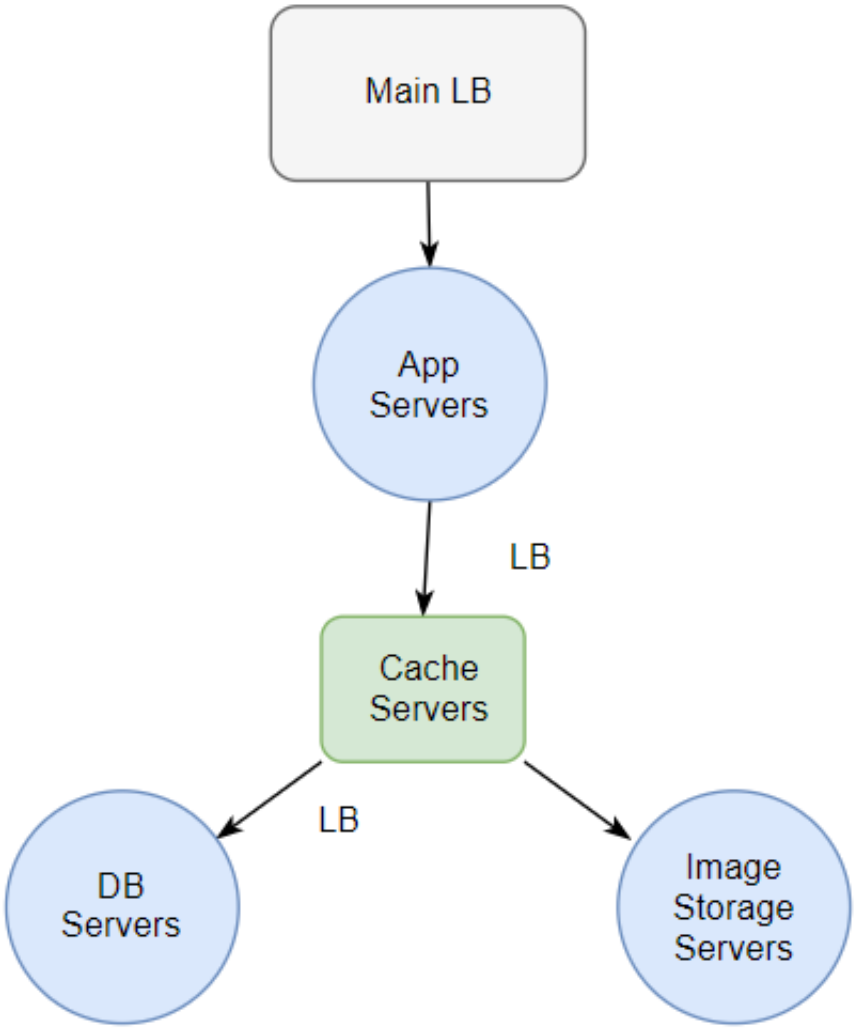
A **hybrid** approach combines the benefits of both push and pull mechanisms. For users with a large number of followers (such as celebrities), the server can use a pull-based approach. For all other users, the server can use a push-based approach. This allows for efficient delivery of updates while minimizing the burden on the server.

Load balancing

To handle user requests, we need to use a load balancer to distribute requests among app servers. One way to do this is to use a round-robin technique, where requests are distributed in a rotating fashion. However,

this approach can be problematic if a server is unavailable, as requests may still be sent to it. To prevent this, we can implement a "heartbeat" system in which each server pings the load balancer at a set interval to indicate that it is not down.

Load balancers are also necessary for database and cache servers, which may also be distributed. To route requests to the appropriate server, we can use a technique called consistent hashing. This involves mapping each request to a specific server based on user-specific information. This helps to ensure that requests are routed to the correct server and that the load is evenly distributed.



To evenly distribute the load among servers, we can use a load balancing algorithm called the "Least Bandwidth Method." This algorithm selects the server with the least amount of traffic (measured in megabits per second)

to handle a request.

Load balancers can be placed at two points in the system: between the client and the server, and between the database and the server. This allows for efficient routing of requests and helps to ensure that the system can handle a large volume of traffic.

Thanks to Navtosh Kumar for his contribution in creating the first version of this content. In case of any queries and feedback, feel free to write us at contact@enjoyalgorithms.com. Enjoy learning, enjoy system design!