highscalability.com

Designing Instagram - High Scalability -

10-13 minutes

This is a guest post by Ankit Sirmorya. Ankit is working as a Machine Learning Lead/Sr. Machine Learning Engineer at Amazon and has led several machine-learning initiatives across the Amazon ecosystem. Ankit has been working on applying machine learning to solve ambiguous business problems and improve customer experience. For instance, he created a platform for experimenting with different hypotheses on Amazon product pages using reinforcement learning techniques. Currently, he is in the Alexa Shopping organization where he is developing machine-learning-based solutions to send personalized reorder hints to customers for improving their experience.

Problem Statement

Design a photo-sharing platform similar to Instagram where users can upload their photos and share it with their followers. Subsequently, the users will be able to view personalized feeds containing posts from all the other users that they follow.

Gathering Requirements

In Scope

The application should be able to support the following requirements.

- Users should be able to upload photos and view the photos they have uploaded.
- Users should be able to follow other users.
- Users can view feeds containing posts from the users they follow.
- Users should be able to like and comment the posts.

Out of Scope

- Sending and receiving messages from other users.
- Generating machine learning based personalized recommendations to discover new people, photos, videos, and stories relevant one's interest.

High Level Design

Architecture

Fig 0: Architecture of Photo Sharing application

When the server receives a request for an action (post, like etc.) from a client it performs two parallel operations: i) persisting the action in the data store ii) publish the action in a streaming data store for a pub-sub model. After that, the various services (e.g. User Feed Service, Media Counter Service) read the actions from the streaming data store and performs their specific tasks. The streaming data store makes the system extensible to support other use-cases (e.g. media search index, locations search index, and so forth) in future.

FUN FACT: In this talk, Rodrigo Schmidt, director of engineering at Instagram talks about the different challenges they have faced in scaling the data infrastructure at Instagram.

System Components

The system will comprise of several micro-services each performing a separate task. We will use a graph database such as Neo4j to store the information. The reason we have chosen a graph data-model is that our data will contain complex relationships between data entities such as users, posts, and comments as nodes of the graph. After that, we will use edges of the graph to store relationships such as follows, likes, comments, and so forth. Additionally, we can use columnar databases like Cassandra to store information like user feeds, activities, and

counters.

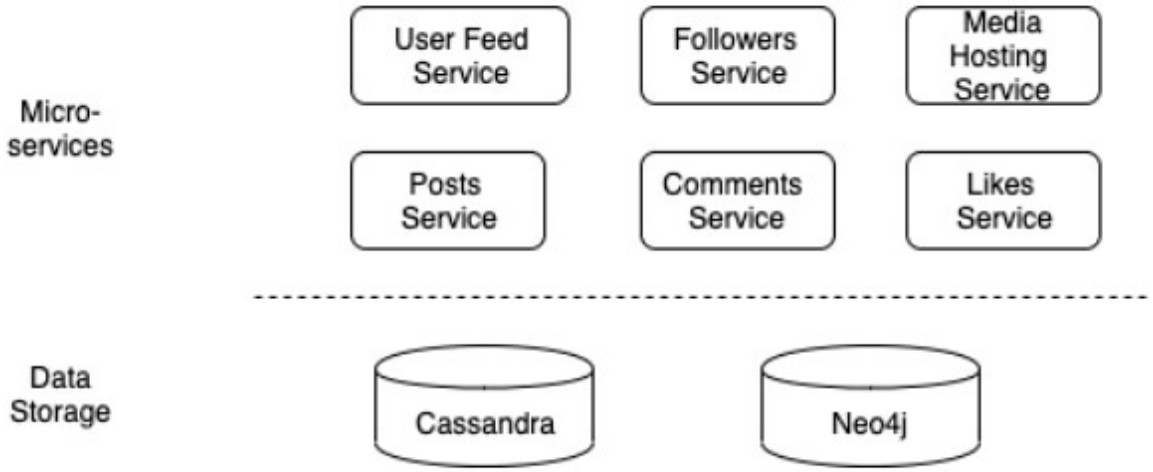

Fig 1: High Level System Components

Component Design

Posting on Instagram

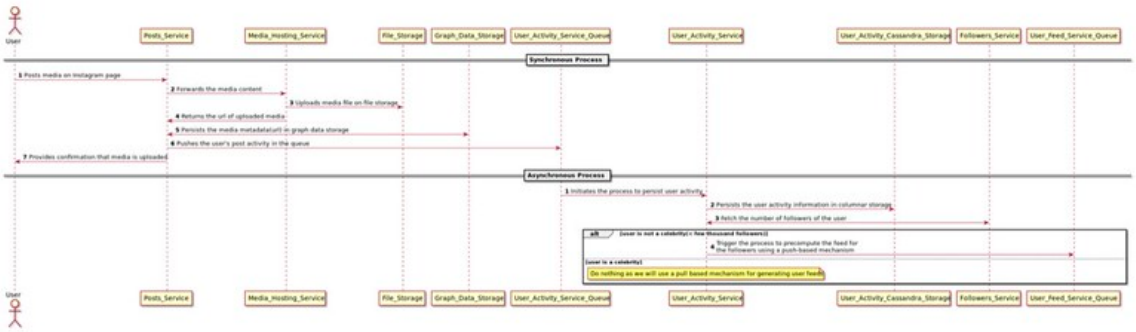

Fig 2: Synchronous and Asynchronous process for posting on Instagram

There are two major processes which gets executed when a user posts a photo on Instagram. Firstly, the synchronous process which is responsible for uploading image content on file storage, persisting the media metadata in graph data-storage, returning the confirmation message to the user and triggering the process to update the user activity. The second process occurs

asynchronously by persisting user activity in a columnar data-storage(Cassandra) and triggering the process to pre-compute the feed of followers of non-celebrity users (having few thousand followers). We don't pre-compute feeds for celebrity users (have 1M+ followers) as the process to fan-out the feeds to all the followers will be extremely compute and I/O intensive.

API Design

We have provided the API design of posting an image on Instagram below. We will send the file and data over in one request using the multipart/form-data content type. The MultiPart/Form-Data contains a series of parts. Each part is expected to contain a content-disposition header [RFC 2183] where the disposition type is "form-data".

URL:
POST /users/<user_id>/posts

Sample Request Body:
Content-Type: ***multipart/form-data***; boundary=ExampleFormBoundary

--ExampleFormBoundary
Content-Disposition: form-data; name="file"; filename="test.png"
Content-Type: image/png

{image file data bytes}
--ExampleFormBoundary--

Sample Response:


```
{
  "type": "ImageFile",
  "id": "667",
  "createdAt": "1466623229",
  "createdBy": "9",
  "name": "test.png",
  "updatedAt": "1466623230",
  "updatedBy": "9",
  "fullImageUrl": "https://mybucket.s3.amazonaws.com/myfolder/test.jpg"
",
  "size": {
```

```
        type : "size",
        "width": "316",
        "height": "316"
    },
}
```

Precompute Feeds

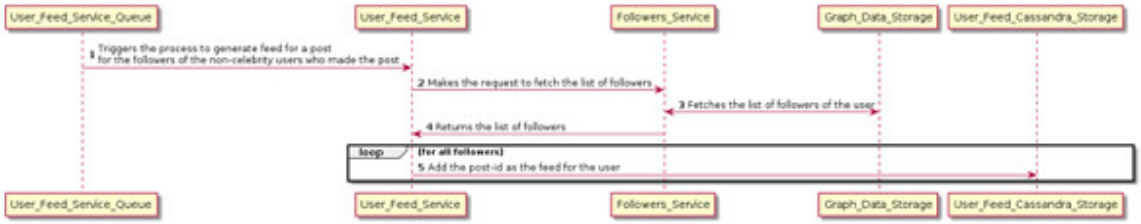

Fig 3: Pre-compute feeds from non-celebrity users

This process gets executed when non-celebrity users makes a post on Instagram. It's triggered when a message is added in the User Feed Service Queue. Once the message is added in the queue, the User Feed Service makes a call to the Followers Service to fetch the list of followers of the user. After that, the post gets added to the feed of all the followers in the columnar data storage.

Fetching User Feed

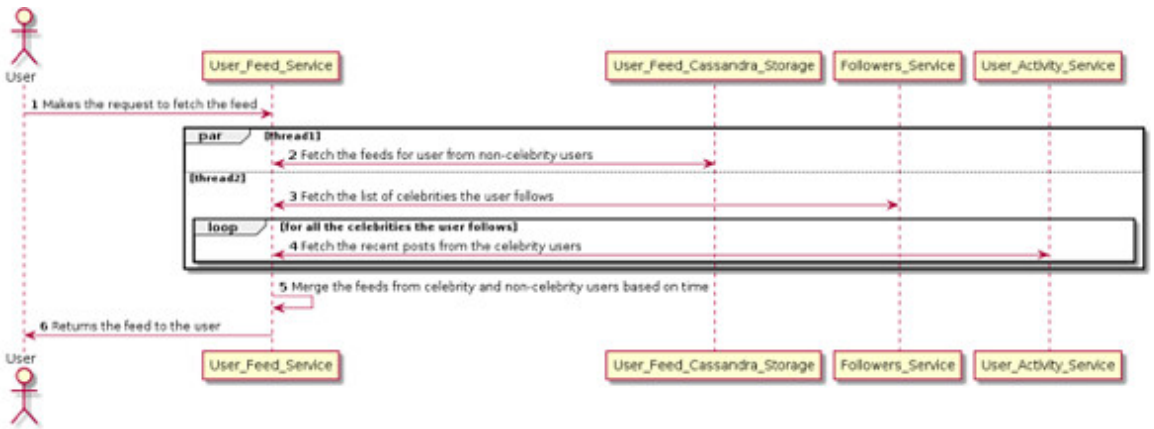

Fig 4: Sequence of operations involved in fetching user feed

When a user requests for feed then there will be two parallel

threads involved in fetching the user feeds to optimize for latency. The first thread will fetch the feeds from non-celebrity users which the user follow. These feeds are populated by the fan-out mechanism described in the PreCompute Feeds section above. The second thread is responsible for fetching the feeds of celebrity users whom the user follow. After that, the User Feed Service will merge the feeds from celebrity and non-celebrity users and return the merged feeds to the user who requested the feed.

API Design

URL:
GET /users/<user_id>/feeds

Sample Response:
The response below can be mapped directly to the graph data model mentioned in the next section.

```
{
   "feeds": [
      {
         "postId": "post001",
         "postOwnerId" : "user001",
         "postOwnerName" : "Bill Gates",
         "mediaURL" : "https://mybucket.s3.amazonaws.com/myfolder/test.jpg",
         "numberOfLikes" : 400000
         "numberOfComments" : 19789
         "comments": [
            {
               "commenterUserId" : "user004",
               "commenterName" : "JeffBezos"
               "comment" : "Amazing!"
               "likes" : [
                  {
                     "likerId" : "user003",
                     "likerUserName" : "Bob"
                  }
               ]
            }
         ]
```

```
        }
    ]
}
```

Data Models

Graph Data Models

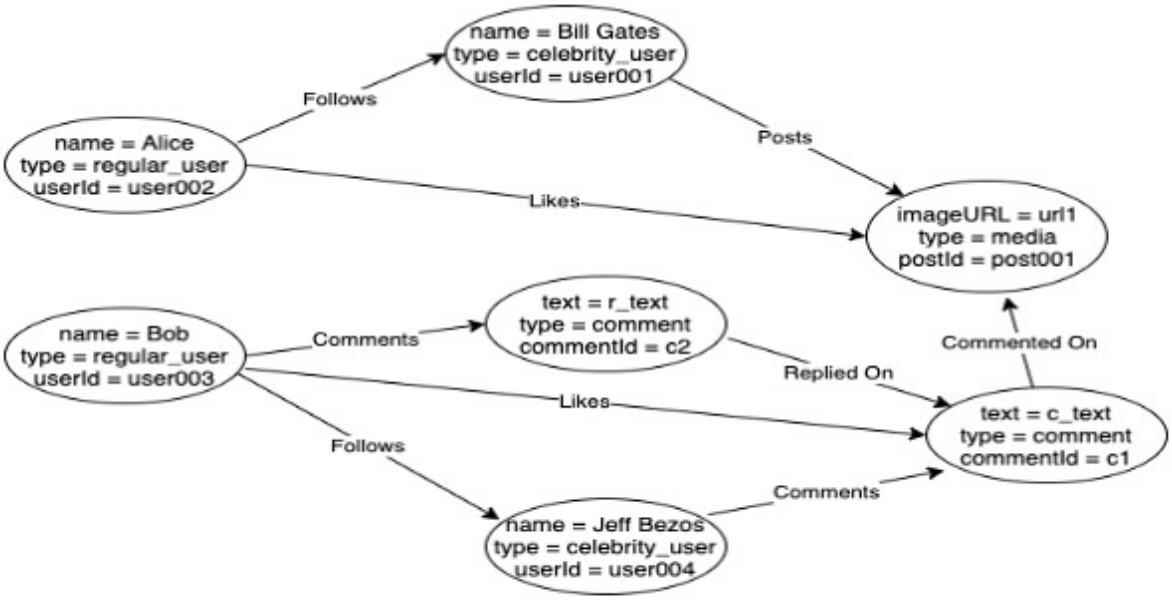

Fig 5: Graph representation of users, posts, and comments

We can use a graph database such as Neo4j which stores data-entities such as user information, posts, comments, and so forth as nodes in the graph. The edges between the nodes are used to store the relationship between data entities such as followers, posts, comments, likes, and replies. All the nodes are added to an index called nodeIndex for faster lookups. We have chosen this NoSQL based solution over relational databases as it provides the scalability to have hierarchies which go beyond two levels and extensibility due to the schema-less behavior of NoSQL data storage.

Sample Queries supported by Graph Database

Fetch all the followers of Jeff Bezos

```
Node jeffBezos = nodeIndex.get(“userId”, “user004”);
List jeffBezosFollowers = new ArrayList();

for (Relationship relationship:
jeffBezos.getRelationships(INGOING, FOLLOWS)) {
   jeffBezosFollowers.add(relationship.getStartNode());
}
```

Fetch all the posts of Bill Gates

```
Node billGates = nodeIndex.get(“userId”, “user001”);
List billGatesPosts = new ArrayList();

for (Relationship relationship:
billGates.getRelationships(OUTGOING, POSTS)) {
   billGatesPosts.add(relationship.getEndNode());
}
```

Fetch all the posts of Bill Gates on which Jeff Bezos has commented

```
List commentsOnBillGatesPosts = new ArrayList<>();

for(Node billGatesPost : billGatesPosts) {
    for (Relationship relationship:
billGates.getRelationships(INGOING, COMMENTED_ON)) {
       commentsOnBillGatesPosts.add(relationship.getStartNode());
    }
```

```
}

List jeffBezosComments = new ArrayList();

for (Relationship relationship:
jeffBezos.getRelationships(OUTGOING, COMMENTS)) {
   jeffBezosComments.add(relationship.getEndNode());
}

List jeffBezosCommentsOnBillGatesPosts =
commentsOnBillGatesPosts.intersect(jeffBezosComments);
```

Columnar Data Models

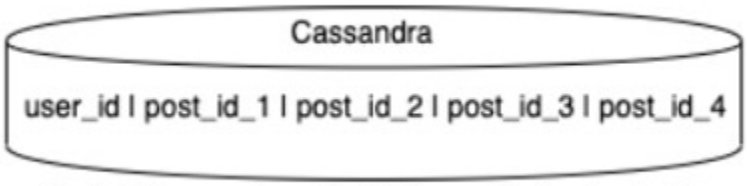

Fig 6: Columnar Data Model for user feed and activities

We will use columnar data storage such as Cassandra to store data entities like user feed and activities. Each row will contain feed/activity information of the user. We can also have a TTL based functionality to evict older posts. The data model will look something similar to:

User_id -> List

FUN FACT: In this talk, Dikang Gu, a software engineer at Instagram core infra team has mentioned about how they use Cassandra to serve critical usecases, high scalability requirements, and some pain points.

Streaming Data Model

We can use cloud technologies such as Amazon Kinesis or Azure Stream Analytics for collecting, processing, and analyzing real-time, streaming data to get timely insights and react quickly to new information(e.g. a new like, comment, etc.). We have listed below the de-normalized form of some major streaming data entities and action.

A. class User {
String id;
String userName;
String fullName;
boolean isPrivate;
…
}
	B. class Post {
	String id;
	String ownerId;
	MediaContentType contentType;
	…
	}

C. class LikeEvent {
String likerId;
String postId;
String postOwnerId;
Post post;
User liker;
User postOwner;
boolean isFollowingMediaOwner
…
}
	D. class Follow {
	User follower;
	User followedUser;
	boolean isFollowedUserCelebrity;
	…
	}

Table: De-normalized major data-entities and actions

The data entities **A** and **B** above show the containers which contain denormalized information about the Users and their Posts. Subsequently, the data entities **C** and **D** denote the different actions which users may take. The entity **C** denotes the event where a user likes a post and entity **D** denotes the action when a user follows another user. These actions are read by the related micro-services from the stream and processed accordingly. For instance, the *LikeEvent* can be read by the *Media Counter Service* and is used to update the media count in the data storage.

Optimization

We will use a cache having an LRU based eviction policy for

caching user feeds of active users. This will not only reduce the overall latency in displaying the user-feeds to users but will also prevent re-computation of user-feeds.

Fig 7: LRU based cache for storing user feeds of active users

Another scope of optimization lies in providing the best content in the user feeds. We can do this by ranking the new feeds (the ones generated after users last login) from those who the user follows. We can apply machine learning techniques to rank the user feeds by assigning scores to the individual feeds which would indicate the probability of click, like, comment and so forth. We can do this by representing each feed by a feature vector which contains information about the user, the feed and the interactions which the user has had with the people in the feed (e.g. whether the user had clicked/liked/commented on the previous feeds by the people in the story). It's apparent that the most important features for feed ranking will be related to social network. Some of the keys of understanding the user network are listed below.

- Who is the user a close follower of? For example, one user is a close follower of Elon Musk while another user can be a close follower of Gordon Ramsay.
- Whose photos the user always like?
- Whose links are most interesting to the user?

We can use deep neural networks which would take the several

features (> 100K dense features) which we require for training the model. Those features will be passed through the n-fold layers, and will be used for predicting the probability of the different events (likes, comments, shares, etc.).

FUN FACT: In this talk, Lars Backstrom, VP of Engineering @ Facebook talks about the machine learning done to create personalized news feeds for users. He talks about the classical machine learning approach they used in the initial phases for personalizing News Feeds by using decision trees and logistic regression. He then goes to talk about the improvements they have observed in using neural networks.

References

- https://www.youtube.com/watch?v=_BfMH4GQWnk
- https://www.youtube.com/watch?v=hnpzNAPiC0E
- https://instagram-engineering.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad
- https://instagram-engineering.com/types-for-python-http-apis-an-instagram-story-d3c3a207fdb7
- http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html
- https://docs.oracle.com/cloud/latest/marketingcs_gs/OMCAC/op-api-rest-1.0-assets-image-content-post.html
- https://instagram-engineering.com/under-the-hood-instagram-in-2015-8e8aff5ab7c2