39. Twitter

Let's design a Twitter like social media service, similar to services like Facebook, Instagram, etc.

What is Twitter?

Twitter is a social media service where users can read or post short messages (up to 280 characters) called tweets. It is available on the web and mobile platforms such as Android and iOS.

Requirements

Our system should meet the following requirements:

Functional requirements

- Should be able to post new tweets (can be text, image, video, etc.).
- Should be able to follow other users.
- Should have a newsfeed feature consisting of tweets from the people the user is following.
- Should be able to search tweets.

Non-Functional requirements

- High availability with minimal latency.
- The system should be scalable and efficient.

Extended requirements

- Metrics and analytics.
- Retweet functionality.
- Favorite tweets.

Estimation and Constraints

Let's start with the estimation and constraints.

Note: Make sure to check any scale or traffic-related assumptions with your interviewer.

Traffic

This will be a read-heavy system, let us assume we have 1 billion total users with 200 million daily active users (DAU), and on average each user tweets 5 times a day. This gives us 1 billion tweets per day.

$$200 \text{ million} \times 5 \text{ tweets} = 1 \text{ billion/day}$$

Tweets can also contain media such as images, or videos. We can assume that 10 percent of tweets are media files shared by the users, which gives us additional 100 million files we would need to store.

$$10 \text{ percent} \times 1 \text{ billion} = 100 \text{ million/day}$$

What would be Requests Per Second (RPS) for our system?

1 billion requests per day translate into 12K requests per second.

$$\frac{1\ billion}{(24\ hrs \times 3600\ seconds)} =\sim 12K\ requests\ /\ second$$

Storage

If we assume each message on average is 100 bytes, we will require about 100 GB of database storage every day.

$$1 \text{ billion} \times 100 \text{ bytes} = 100\text{GB/day}$$

We also know that around 10 percent of our daily messages (100 million) are media files per our requirements. If we assume each file is 50 KB on average, we will require 5 TB of storage every day.

$$100 \text{ million} \times 50 \text{ KB} = 5\text{TB/day}$$

And for 10 years, we will require about 19 PB of storage.

$$(5 \text{ TB} + 0.1 \text{ TB}) \times 365 \text{ days} \times 10 \text{ years} = 19\text{PB}$$

Bandwidth

As our system is handling 5.1 TB of ingress every day, we will require a minimum bandwidth of around 60 MB per second.

$$\frac{5.1\,TB}{(24\,hrs \times 3600\,seconds)} =\sim 60\,MB\ /\ second$$

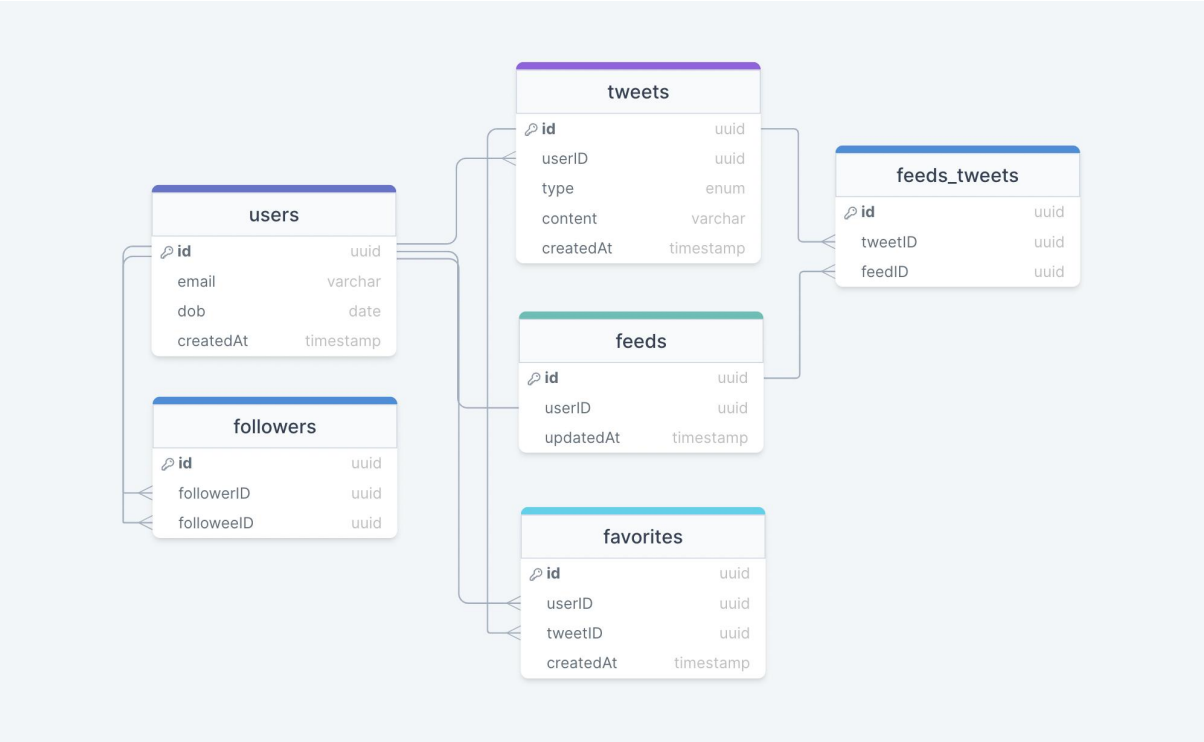
High-level estimate

Here is our high-level estimate:

Type	Estimate
Daily active users (DAU)	100 million
Requests per second (RPS)	12K/s
Storage (per day)	~5.1 TB
Storage (10 years)	~19 PB
Bandwidth	~60 MB/s

Data model design

This is the general data model which reflects our requirements.



We have the following tables:

users

- This table will contain a user's information such as name, email, dob, and other details.

tweets

- As the name suggests, this table will store tweets and their properties such as type (text, image, video, etc.), content, etc. We will also store the corresponding userID.

favorites

- This table maps tweets with users for the favorite tweets functionality in our application.

followers

- This table maps the followers and followees as users can follow each other (N:M relationship).

feeds

- This table stores feed properties with the corresponding userID.

feeds_tweets

- This table maps tweets and feed (N:M relationship).

What kind of database should we use?

While our data model seems quite relational, we don't necessarily need to store everything in a single database, as this can limit our scalability and quickly become a bottleneck.

We will split the data between different services each having ownership over a particular table. Then we can use a relational database such as PostgreSQL or a distributed NoSQL database such as Apache Cassandra for our use case.

API design

Let us do a basic API design for our services:

Post a tweet

This API will allow the user to post a tweet on the platform.

```
postTweet(userID: UUID, content: string, mediaURL?: string): boolean
```

Parameters

- User ID (UUID): ID of the user.
- Content (string): Contents of the tweet.
- Media URL (string): URL of the attached media (optional).
- Returns
- Result (boolean): Represents whether the operation was successful or not.

Follow or unfollow a user

This API will allow the user to follow or unfollow another user.

```
follow(followerID: UUID, followeeID: UUID): boolean

unfollow(followerID: UUID, followeeID: UUID): boolean
```

Parameters

- Follower ID (UUID): ID of the current user.
- Followee ID (UUID): ID of the user we want to follow or unfollow.
- Media URL (string): URL of the attached media (optional).
- Returns
- Result (boolean): Represents whether the operation was successful or not.

Get newsfeed

This API will return all the tweets to be shown within a given newsfeed.

```
getNewsfeed(userID: UUID): Tweet[]
```

Parameters

- User ID (UUID): ID of the user.
- Returns
- Tweets (Tweet[]): All the tweets to be shown within a given newsfeed.

High-level design

Now let us do a high-level design of our system.

Architecture

We will be using microservices architecture since it will make it easier to horizontally scale and decouple our services. Each service will have ownership of its own data model. Let's try to divide our system into some core services.

User Service

This service handles user-related concerns such as authentication and user information.

Newsfeed Service

This service will handle the generation and publishing of user newsfeeds. It will be discussed in detail separately.

Tweet Service

The tweet service will handle tweet-related use cases such as posting a tweet, favorites, etc.

Search Service

The service is responsible for handling search-related functionality. It will be discussed in detail separately.

Media service

This service will handle the media (images, videos, files, etc.) uploads. It will be discussed in detail separately.

Notification Service

This service will simply send push notifications to the users.

Analytics Service

This service will be used for metrics and analytics use cases.

What about inter-service communication and service discovery?

Since our architecture is microservices-based, services will be communicating with each other as well. Generally, REST or HTTP performs well but we can further improve the performance using gRPC which is more lightweight and efficient.

Service discovery is another thing we will have to take into account. We can also use a service mesh that enables managed, observable, and secure communication between individual services.

Note: Learn more about REST, GraphQL, gRPC and how they compare with each other.

Newsfeed

When it comes to the newsfeed, it seems easy enough to implement, but there are a lot of things that can make or break this feature. So, let's divide our problem into two parts:

Generation

Let's assume we want to generate the feed for user A, we will perform the following steps:

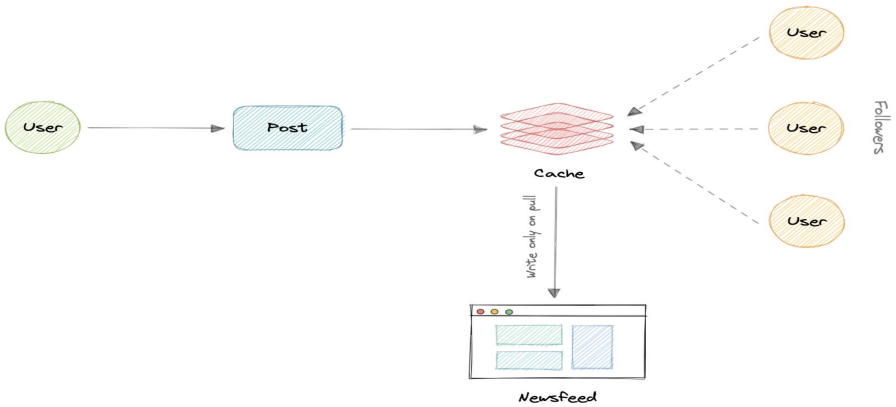
1. Retrieve the IDs of all the users and entities (hashtags, topics, etc.) user A follows.
2. Fetch the relevant tweets for each of the retrieved IDs.
3. Use a ranking algorithm to rank the tweets based on parameters such as relevance, time, engagement, etc.
4. Return the ranked tweets data to the client in a paginated manner.

Feed generation is an intensive process and can take quite a lot of time, especially for users following a lot of people. To improve the performance, the feed can be pre-generated and stored in the cache, then we can have a mechanism to periodically update the feed and apply our ranking algorithm to the new tweets.

Publishing

Publishing is the step where the feed data is pushed according to each specific user. This can be a quite heavy operation, as a user may have millions of friends or followers. To deal with this, we have three different approaches:

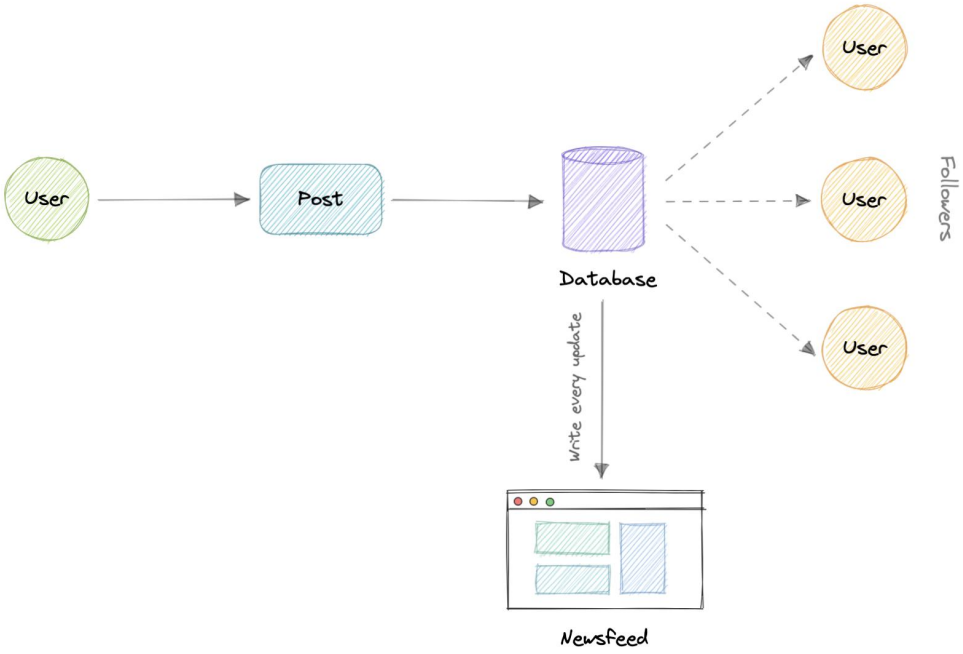
Pull Model (or Fan-out on load)



When a user creates a tweet, and a follower reloads their newsfeed, the feed is created and stored in memory. The most recent feed is only loaded when the user requests it. This approach reduces the number of write operations on our database.

The downside of this approach is that the users will not be able to view recent feeds unless they "pull" the data from the server, which will increase the number of read operations on the server.

Push Model (or Fan-out on write)



In this model, once a user creates a tweet, it is "pushed" to all the follower's feeds immediately. This prevents the system from having to go through a user's entire followers list to check for updates.

However, the downside of this approach is that it would increase the number of write operations on the database.

Hybrid Model

A third approach is a hybrid model between the pull and push model. It combines the beneficial features of the above two models and tries to provide a balanced approach between the two.

The hybrid model allows only users with a lesser number of followers to use the push model. For users with a higher number of followers such as celebrities, the pull model is used.

Ranking Algorithm

As we discussed, we will need a ranking algorithm to rank each tweet according to its relevance to each specific user.

For example, Facebook used to utilize an EdgeRank algorithm. Here, the rank of each feed item is described by:

$$\text{Rank} = \text{Affinity} \times \text{Weight} \times \text{Decay}$$

Where,

- Affinity: is the "closeness" of the user to the creator of the edge. If a user frequently likes, comments, or messages the edge creator, then the value of affinity will be higher, resulting in a higher rank for the post.
- Weight: is the value assigned according to each edge. A comment can have a higher weightage than likes, and thus a post with more comments is more likely to get a higher rank.
- Decay: is the measure of the creation of the edge. The older the edge, the lesser will be the value of decay and eventually the rank.

Nowadays, algorithms are much more complex and ranking is done using machine learning models which can take thousands of factors into consideration.

Retweets

Retweets are one of our extended requirements. To implement this feature, we can simply create a new tweet with the user id of the user retweeting the original tweet and then modify the type enum and content property of the new tweet to link it with the original tweet.

For example, the type enum property can be of type tweet, similar to text, video, etc and content can be the id of the original tweet. Here the first row indicates the original tweet while the second row is how we can represent a retweet.

Id	userId	Type	Content	CreatedAt
ad34-291a-45f6-b36c	ad34-291a-45f6-b36c	Text	Hey, this is my first tweet…	1658905644054
f064-49ad-9aa2-84a6	f064-49ad-9aa2-84a6	Tweet	ad34-291a-45f6-b36c	1658906165427

This is a very basic implementation. To improve this we can create a separate table itself to store retweets.

Search

Sometimes traditional DBMS are not performant enough, we need something which allows us to store, search, and analyze huge volumes of data quickly and in near real-time and give results within milliseconds. Elasticsearch can help us with this use case.

Elasticsearch is a distributed, free and open search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured. It is built on top of Apache Lucene.

How do we identify trending topics?

Trending functionality will be based on top of the search functionality. We can cache the most frequently searched queries, hashtags, and topics in the last N seconds and update them every M seconds using some sort of batch job mechanism. Our ranking algorithm can also be applied to the trending topics to give them more weight and personalize them for the user.

Notifications

Push notifications are an integral part of any social media platform. We can use a message queue or a message broker such as Apache Kafka with the notification service to dispatch requests to Firebase Cloud Messaging (FCM) or Apple Push Notification Service (APNS) which will handle the delivery of the push notifications to user devices.

For more details, refer to the WhatsApp system design where we discuss push notifications in detail.

Detailed design

It's time to discuss our design decisions in detail.

Data Partitioning

To scale out our databases we will need to partition our data. Horizontal partitioning (aka Sharding) can be a good first step. We can use partitions schemes such as:

- Hash-Based Partitioning
- List-Based Partitioning
- Range Based Partitioning
- Composite Partitioning

The above approaches can still cause uneven data and load distribution, we can solve this using Consistent hashing.

For more details, refer to Sharding and Consistent Hashing.

Mutual friends

For mutual friends, we can build a social graph for every user. Each node in the graph will represent a user and a directional edge will represent followers and followees. After that, we can traverse the followers of a user to find and suggest a mutual friend. This would require a graph database such as Neo4j or ArangoDB.

This is a pretty simple algorithm, to improve our suggestion accuracy, we will need to incorporate a recommendation model which uses machine learning as part of our algorithm.

Metrics and Analytics

Recording analytics and metrics is one of our extended requirements. As we will be using Apache Kafka to publish all sorts of events, we can process these events and run analytics on the data using Apache Spark which is an open-source unified analytics engine for large-scale data processing.

Caching

In a social media application, we have to be careful about using cache as our users expect the latest data. So, to prevent usage spikes from our resources we can cache the top 20% of the tweets.

To further improve efficiency we can add pagination to our system APIs. This decision will be helpful for users with limited network bandwidth as they won't have to retrieve old messages unless requested.

Which cache eviction policy to use?

We can use solutions like Redis or Memcached and cache 20% of the daily traffic but what kind of cache eviction policy would best fit our needs?

Least Recently Used (LRU) can be a good policy for our system. In this policy, we discard the least recently used key first.

How to handle cache miss?

Whenever there is a cache miss, our servers can hit the database directly and update the cache with the new entries.

For more details, refer to Caching.

Media access and storage

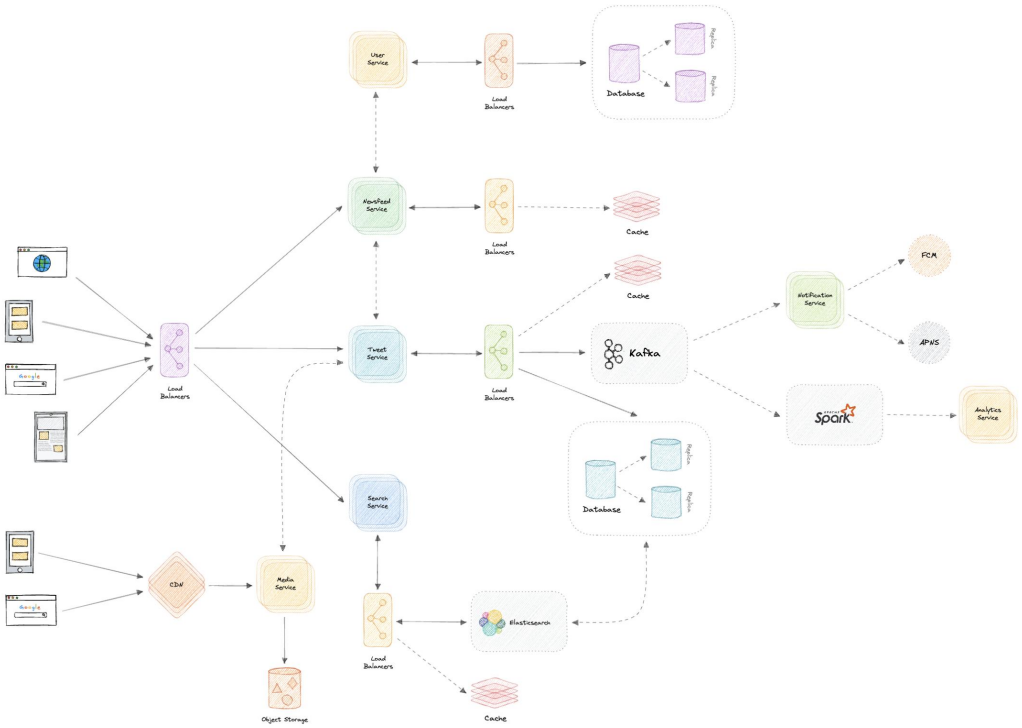
As we know, most of our storage space will be used for storing media files such as images, videos, or other files. Our media service will be handling both access and storage of the user media files.

But where can we store files at scale? Well, object storage is what we're looking for. Object stores break data files up into pieces called objects. It then stores those objects in a single repository, which can be spread out across multiple networked systems. We can also use distributed file storage such as HDFS or GlusterFS.

Content Delivery Network (CDN)

Content Delivery Network (CDN) increases content availability and redundancy while reducing bandwidth costs. Generally, static files such as images, and videos are served from CDN. We can use services like Amazon CloudFront or Cloudflare CDN for this use case.

Identify and resolve bottlenecks



Let us identify and resolve bottlenecks such as single points of failure in our design:

- "What if one of our services crashes?"
- "How will we distribute our traffic between our components?"
- "How can we reduce the load on our database?"
- "How to improve the availability of our cache?"
- "How can we make our notification system more robust?"
- "How can we reduce media storage costs"?

To make our system more resilient we can do the following:

- Running multiple instances of each of our services.
- Introducing load balancers between clients, servers, databases, and cache servers.
- Using multiple read replicas for our databases.

- Multiple instances and replicas for our distributed cache.
- Exactly once delivery and message ordering is challenging in a distributed system, we can use a dedicated message broker such as Apache Kafka or NATS to make our notification system more robust.
- We can add media processing and compression capabilities to the media service to compress large files which will save a lot of storage space and reduce cost.

```
import java.util.*;
import java.util.concurrent.*;

public class ChatSystem {

    // Data models
    static class Message {
        String senderID;
        String receiverID;
        String groupID; // Optional for group chats
        String content;
        String mediaURL; // Optional for media messages
        long timestamp;

        public Message(String senderID, String receiverID, String content, String mediaURL) {
            this.senderID = senderID;
            this.receiverID = receiverID;
            this.content = content;
            this.mediaURL = mediaURL;
            this.timestamp = System.currentTimeMillis();
        }
    }

    static class User {
        String userID;
        String name;
        String email;
```

```
        public User(String userID, String name, String email) {
            this.userID = userID;
            this.name = name;
            this.email = email;
        }
    }

    // Storage: In-memory cache for simulation
    static Map<String, List<Message>> messageStorage = new ConcurrentHashMap<>();
    static Map<String, Set<String>> groupMembers = new ConcurrentHashMap<>();
    static Map<String, User> users = new ConcurrentHashMap<>();
    static Set<String> activeUsers = ConcurrentHashMap.newKeySet();    // To track active users for push notifications

    // Simulate sending a message
    public boolean sendMessage(String senderID, String receiverID, String content, String mediaURL) {
        if (senderID == null || receiverID == null || content == null || content.isEmpty()) {
            return false;
        }

        Message message = new Message(senderID, receiverID, content, mediaURL);
        messageStorage.computeIfAbsent(receiverID, k -> new ArrayList<>()).add(message);
        activeUsers.add(receiverID); // Adding to active users for notifications
        sendNotification(receiverID, message);

        return true;
    }

    // Simulate fetching messages
    public List<Message> fetchMessages(String userID, String lastMessageID) {
        List<Message> messages = messageStorage.getOrDefault(userID, Collections.emptyList());
        // Filter messages after the last message ID (pagination logic)
        return messages;
    }
```

```
    // Simulate joining a group
    public boolean joinGroup(String userID, String groupID) {
        groupMembers.computeIfAbsent(groupID, k -> new HashSet<>()).add(userID);
        return true;
    }

    // Simulate leaving a group
    public boolean leaveGroup(String userID, String groupID) {
        Set<String> members = groupMembers.get(groupID);
        if (members != null) {
            members.remove(userID);
        }
        return true;
    }

    // Simulate sending push notification
    private void sendNotification(String userID, Message message) {
        // Push notification simulation (e.g., to mobile app)
        if (activeUsers.contains(userID)) {
            System.out.println("Sending push notification to: " + userID + " - Message: " +
message.content);
        }
    }

    // Main method to simulate chat system
    public static void main(String[] args) {
        ChatSystem chatSystem = new ChatSystem();
        chatSystem.users.put("user1", new User("user1", "Alice", "alice@example.com"));
        chatSystem.users.put("user2", new User("user2", "Bob", "bob@example.com"));

        // Send message
        chatSystem.sendMessage("user1", "user2", "Hello Bob!", null);

        // Fetch messages for user2
        List<Message> messages = chatSystem.fetchMessages("user2", null);
```

```
            messages.forEach(msg -> System.out.println("Message from " + msg.senderID + ": " +
msg.content));
        }
}
```

"How would you design a system like Twitter". Well, let's design Twitter then!

Let's look at the requirements to start with.

Functional Requirements

- Tweet - should allow you to post text, image, video, links, etc
- Re-tweet - should allow you to share someone's tweets
- Follow - this will be a directed relationship. If we follow Barack Obama, he doesn't have to follow us back
- Search

Non Functional Requirements

- Read heavy - The read to write ratio for twitter is very high, so our system should be able to support that kind of pattern
- Fast rendering
- Fast tweet.
- Lag is acceptable - From the previous two NFRs, we can understand that the system should be highly available and have very low latency. So when we say lag is ok, we mean it is ok to get notification about someone else's tweet a few seconds later, but the rendering of the content should be almost instantaneous.
- Scalable - 5k+ tweets come in every second on twitter on an average day. On peak times it can easily double up. These are just tweets, and as we have already discussed read to write ratio of twitter is very high i.e. there will be an even higher number of reads happening against these tweets. That is a huge amount of requests per second.

So how do we design a system that delivers all our functional requirements without compromising the performance? Before we discuss the overall architecture, let's split

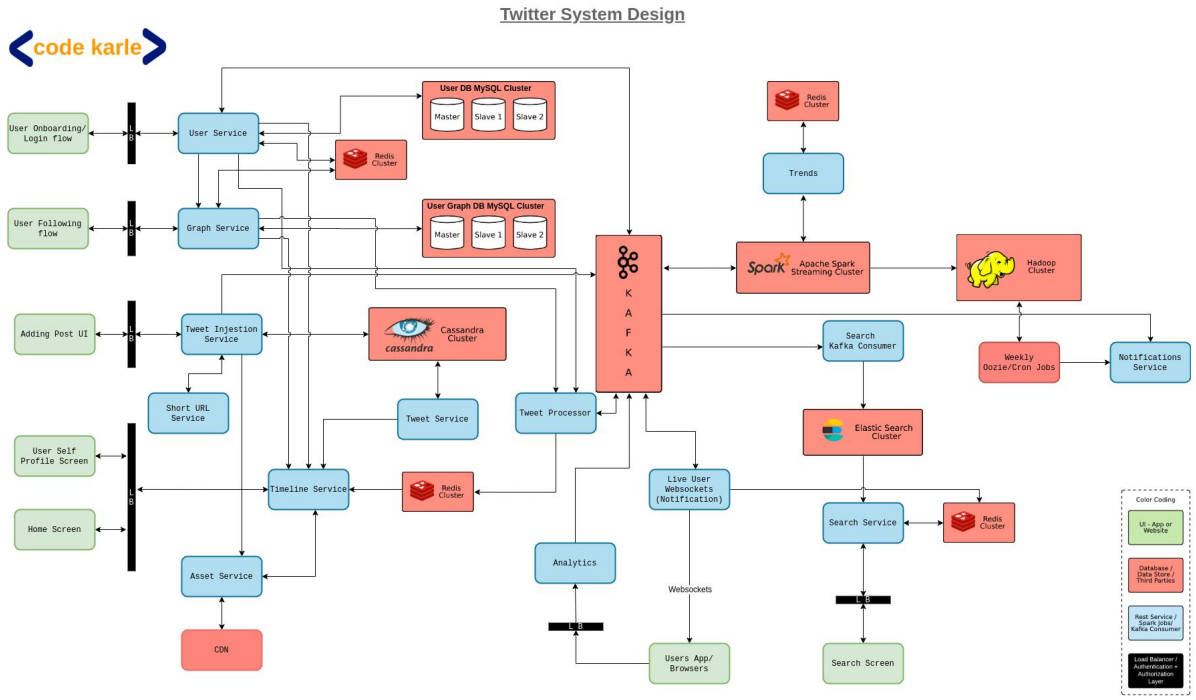
our users into different categories. Each of these categories will be handled in a slightly different manner.

- **Famous Users:** Famous users are usually celebrities, sportspeople, politicians, or business leaders who have a lot of followers
- **Active Users:** These are the users who have accessed the system in the last couple of hours or days. For our discussion, we will consider people who have accessed twitter in the last three days as active users.
- **Live Users:** These are a subset of active users who are using the system right now, similar to online on Facebook or WhatsApp..
- **Passive Users:** These are the users who have active accounts but haven't accessed the system in the last three days.
- **Inactive Users:** These are the "deleted" accounts so to speak. We don't really delete any accounts, it is more of a soft delete, but as far as the users are concerned the account doesn't exist anymore.

Now, for simplicity, let us divide the overall architecture into three flows. We will separately look at the onboarding flow, tweet flow, and search and analytics side of the system.

Note: Remember how twitter is a very read-heavy system? Well, while designing a read-heavy system, we need to make sure that we are precomputing and caching as much as we can to keep the latency as low as possible.

System Architecture



Twitter system architecture design

Onboarding flow

We have a User Service that will store all the user-related information in our system and provide endpoints for login, register, and any other internal services that need user-related information by providing GET APIs to fetch users by id or email, POST APIs to add or edit user information and bulk GET APIs to fetch information about multiple users. This user service sits on top of a User DB which is a MySQL database. We use MySQL here as we have a finite number of users and the data is very relational. Also, the User DB will mostly power the writes and the reads related to user details will be powered by a Redis cache which is an image of User DB. When user service receives a GET request with a user id, it will firstly lookup in the Redis cache, if the user is present in the Redis it will return the response. Otherwise, it will fetch the information from User DB, store it in Redis, and then respond to the client.

User-follow flow

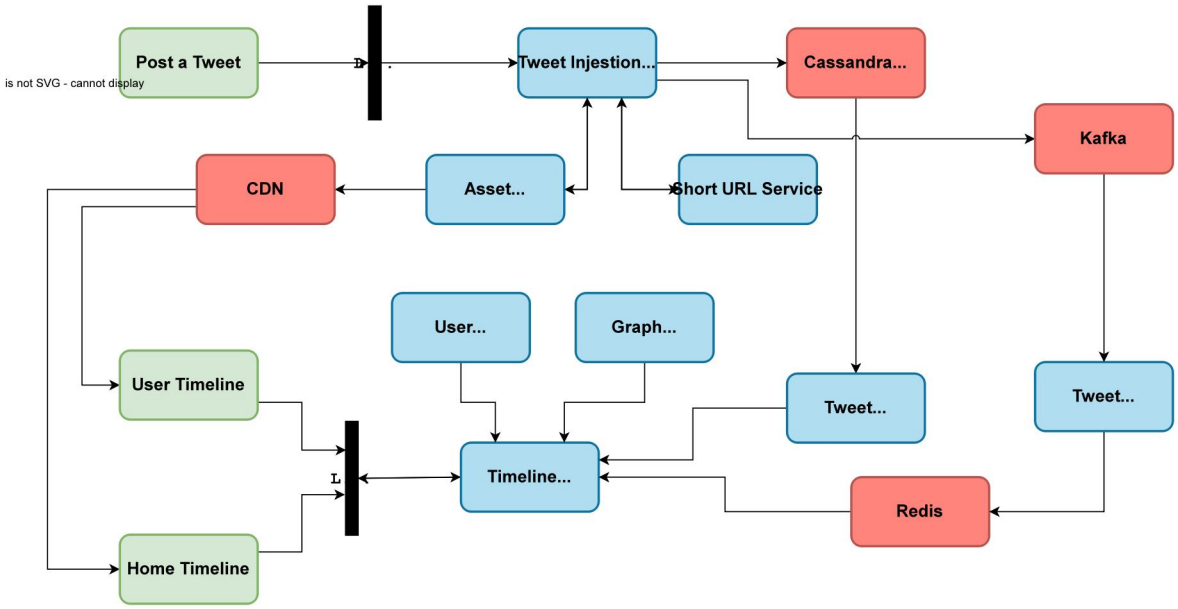
The “follow” related requests will be serviced by a Graph Service, which creates a network of how the users are connected within the system. Graph service will expose APIs to add follow links, get users followed by a certain user-id or the users following a certain user-id. This Graph service sits on top of a User Graph DB which is again a MySQL DB. Again, the follow-links won’t change too frequently, so it makes sense to cache this information in a Redis. Now in the follow flow we can cache two pieces of information - who are the followers of a particular user, and who is the user following. Similar to the user service, when graph service receives a get request it will firstly lookup in the Redis. If Redis has the information it responds to the user. Otherwise, it fetches the information from the graph DB, stores it in Redis, and responds to the user.

Now, there are some things we can conclude based on a user’s interaction with Twitter, like their interests, etc. So when such events occur, an Analytics Service puts these events in a Kafka.

Now, remember our Live users? Say U1 is a live user following U2 and U2 tweets something. Since U1 is live, it makes sense that U1 gets notified immediately. This happens through the User Live Websocket service. This service keeps an open connection with all the live users, and whenever an event occurs that a live user needs to be notified of, it happens through this service. Now based on the user’s interaction with this service we can also track for how long the users are online, and when the interaction stops we can conclude that the user is not live anymore. When the user goes offline, through the websocket service an event will be fired to Kafka which will further interact with user service and save the last active time of the user in Redis, and other systems can use this information to accordingly modify their behavior.

Tweet flow

Now a tweet could contain text, images, videos, or links. We have something called an Asset service which takes care of uploading and displaying all the multimedia content in a Tweet. We have discussed the nitty-gritty of asset service in the Netflix design article, so check that out if you are interested.



Now, we know that tweets have a constraint of 140 characters which can include text and links. Thanks to this limit we cannot post huge URLs in our tweets. This is where a URL shortener service comes in. We are not going into the details of how this service works, but we have discussed it in our Tiny URL article, so make sure to check it out. Now that we have handled the links as well, all that is left is to store the text of a tweet and fetch it when required. This is where the tweet ingestion service comes in. When a user tries to post a tweet and hits the submit button, it calls the tweet ingestion service which stores the tweet in a permanent data store. We use Cassandra here because we will have a huge amount of tweets coming in every day and the query pattern we require here is what Cassandra is best for. To know more about why we use the database solutions we do check out our article about choosing the best storage solutions.

Twitter system design post tweet flow

Now, the tweet ingestion service, as the name suggests, is only responsible for posting the tweets and doesn't expose any GET APIs to fetch tweets. As soon as a tweet is posted, the tweet ingestion service will fire an event to Kafka saying a tweet id was posted by so and so user id. Now on top of our Cassandra, sits a Tweet service that will expose APIs to get tweets by tweet id or user id.

Now, let's have a quick look at the users' side of things. On the read-flow, a user can have a user timeline i.e. the tweets from that user or a home timeline i.e. tweets from the people a user is following. Now a user could have a huge list of users they are following, and if we make all the queries at runtime before displaying the timeline it will slow down the rendering. So we cache the user's timeline instead. We will precalculate the timeline of active users and cache it in a Redis, so an active user can instantaneously see their timeline. This can be achieved with something called a Tweet processor.

As mentioned before, when a tweet is posted, an event is fired to Kafka. Kafka communicates the same to the tweet processor and creates the timeline for all the users that need to be notified of this recent tweet and cache it. To find out the followers that need to be notified of this change tweet service interacts with the graph service. Suppose user U1, followed by users U2, U3, and U4 posts a tweet T1, then the tweet processor will update the timelines for U2, U3, and U4 with tweet T1 and update the cache.

Now, we have only cached the timelines for active users. What happens when a passive user, say P1, logs in to the system? This is where the Timeline Service comes in. The request will reach the timeline service, timeline service will interact with the user service to identify if P1 is an active user or a passive user. Now since P1 is a passive user, its timeline is not cached in Redis. Now the timeline service will talk to the graph service to find a list of users that P1 follows, then queries the tweet service

to fetch tweets of all those users, caches them in the Redis, and responds back to the client.

Now we have seen the behavior for active and passive users. How will we optimize the flow for our live users? As we have previously discussed, when a tweet is successfully posted an event will be sent to Kafka. Kafka will then talk to the tweet processor which creates timelines for active users and saves them in Redis. But here if the tweet processor identifies that one of the users that need to be updated is a live user, then it will fire an event to Kafka which will now interact with the live websocket service we briefly discussed before. This websocket service will now send a notification to the app and update the timeline.

So now our system can successfully post tweets with different types of content and has some optimization built in to handle active, passive, and live users in a somewhat different manner. But it is still a fairly inefficient system. Why? Because we completely forgot about our famous users! If Donald Trump has 75 million followers, then every time Trump tweets about something, our system needs to make 75 million updates. And this is just one tweet from one user. So this flow will not work for our famous users.

Redis cache will only cache the tweets from non-famous users in the precalculated timelines. Timeline service knows that Redis only stores tweets from normal users. It interacts with graph service to get a list of famous users followed by our current user, say U1, and it fetches their tweets from the tweet service. It will then update these tweets in Redis and add a timestamp indicating when the timeline was last updated. When the next request comes from U1, it checks if the timestamp in Redis against U1 is from a few minutes back. If so, it will query the tweet service again. But if the timestamp is fairly recent, Redis will directly respond back to the app.

Now we have handled active, passive, live, and famous users. As for the inactive users, they are already deactivated accounts so we need not worry about them.

Now what happens when a famous user follows another famous user, let's say Donald Trump and Elon Musk? If Donald Trump tweets, Elon musk should be notified immediately even if the other non-famous users are not notified. This is handled by the tweet processor again. Tweet processor, when it receives an event from Kafa about a new tweet from a famous user, let's say, Donald Trump, updates the cache of the famous users that follow Trump.

Now, this looks like a fairly efficient system, but there are some bottlenecks. Like Cassandra - which will be under a huge load, Redis - which needs to scale efficiently as it is stored completely in RAM, and Kafka - which again will receive crazy amounts of events. So we need to make sure that these components are horizontally scalable and in case of Redis, don't store old data that just unnecessarily uses up memory.

Now coming to **search and analytics!**

Remember the tweet **ingestion service** we discussed in the previous section? When a tweet is added to the system, it fires an event to Kafka. A search consumer listening to Kafka stores all these incoming tweets into an Elasticsearch database. Now when the user searches a string in **Search UI**, which talks to a Search Service. **Search service** will talk to elastic search, fetch the results, and respond back to the user.

Now assuming an event occurred and people are tweeting or searching about it on Twitter, then it is safe to assume that more people will search for it. Now we shouldn't have to query the same thing again and again on elasticsearch. Once the search service gets some results from elasticsearch, it will save them in Redis with a time-to-live of 2-3 minutes. Now when the user searches something, Search service will firstly lookup in Redis. If the data is found in Redis it will be responded back to the user, otherwise, the search service will query elasticsearch, get the data, store it in Redis, and respond back to the user. This considerably reduces the load on elasticsearch.

Let’s go back to our Kafka again. There will be a spark streaming consumer connected to Kafka which will keep track of trending keywords and communicate them to the Trends service. This could further be connected to a Trend UI to visualize this data. We don’t need a permanent data store for this information as trends will be temporary but we could use a Redis as a cache for short-term storage.

Now you must have noticed we have used Redis very heavily in our design. Now even though Redis is an in-memory solution, there is still an option to save data to disk. So in case of an outage, if some of the machines go down, you still have the data persisted on the disk to make it a little more fault-tolerant.

Now, other than trends there is still some other analytics that can be performed like what are people from India talking about. For this, we will dump all the incoming tweets in a Hadoop cluster, which can power queries like the most re-tweeted posts, etc. We could also have a weekly cron job running on the Hadoop cluster, which will pull in the information about our passive users and send out a weekly newsletter to them with some of the most recent tweets that they might be interested in. This could be achieved by running some simple ML algorithms that could tell the relevance of tweets based on the previous searches and reads by the users. The newsletters can be sent via a notification service that can talk to user service to fetch the email ids of the users.