Twitter

Thursday, 29 February 2024 1:08 AM

Lets design a twitter like social media service, similar to facebook, instagram etc

What is "Twitter"?

— social media service where users can read or post short messages (upto 280 characters) called tweets.
— available on web and mobile platforms such as android and IOS.

Requirements

Functional
— 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
— high availability with minimal latency
— system should be scalable and efficient

Extended
— metrics and analytics
— retweet functionality
— favourite tweets

Estimation and Constraints

Traffic
This will be a read-heavy system.
— lets assume, we have 1 billion users with 200 million daily active users (DAU), and on average, each user tweets 5 times a day.
200 million X 5 tweets = 1 billion/day
— tweets can contain media such as images, or videos. Assume 10% of tweets are media files shared by users.
10 percent * 1 billion = 100 million/day files to store

Request per second for our system?
1 billion requests per day translate into 12K requests per second.
1 billion / (24 hrs * 3600 seconds) =~ 12K requests/second

Storage
— assume each message on average is 100 bytes,
1 billion * 100 bytes =~ 100 GB/day
— Also, 10% of daily messages (100 million) are media files, each file to be 50Kb on average
100 million * 50 kbytes =~ 5 TB/day
— For 10 years, we require about 19PB of storage.
(5TB + 0.1TB) * 365 days * 10 years = ~ 19 PB

Bandwidth
— as system is handling 5.1TB of ingress every day
5.1TB / 24hrs * 3600 seconds =~ 60 MB/second (minimum bandwidth)

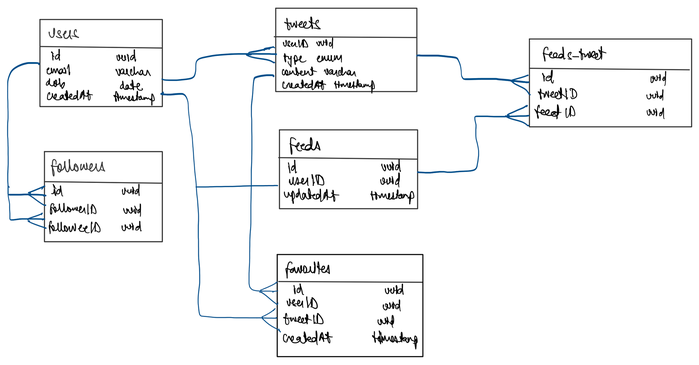
High level Estimate

Type	Estimate
Daily Active Users	100 million
Requests per second (RPS)	12k/s
storage (per day)	~5.1TB
storage (10 years)	~19 PB
bandwidth	~60MB/s

Data Model Design

Tables
users — this table will contain a user's information such as name, email, dob and other details.
tweets — this table will store tweets and their properties such as type (text, image, video etc), content, etc we will also store corresponding userId.
favourites — this table maps tweets with users for the favourite tweet 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.
feed_tweets — this table maps tweets and feed (N:M relationship)

Kind of database
— while data model seems quite relational, we don't necessarily need to store everything in a single db, as this can limit our scalability and quickly become a bottleneck.
— split data between different services each having ownership over a particular table.
— we can use relational db such as postgreSQL or a distributed NoSQL db.



API Design

— Post a tweet — this API will allow user to post a tweet on the platform.
postTweet (userID: uuid, content: string, mediaURL?: string): boolean

— Follow or unfollow a user — this API will allow user to follow or unfollow another user
follow (followerID: uuid, followeeID: uuid): boolean
unfollow (followerID: uuid, followeeID: uuid): boolean

— Get newsfeed — this API will return all the tweets to be shown within a given newsfeed
getNewsfeed (userID: uuid): Tweet[]

High Level Design

Architecture — we will be using a microservices architecture since it will make it easier to horizontally scale and decouple our services.
— each service will have ownership of the own data model.

Core Services
— user service : handles user-related concerns such as authentication and user information.
— newsfeed service : handles generation and publishing of user newsfeed. (discussed seperately).
— tweet service : handles tweet-related use cases such as posting a tweet, favourite etc.
— search service : responsible for handling search related functionality. (discussed seperately).
— media service : handle the media (image, videos, files etc) uploads. (discussed seperately).
— notification service : simply push notification to the user.
— analytics service : used for metrics and analytics use cases.

What about inter-service communication and service discovery?
— 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 use a service mesh that enables managed, observable and secure communication between individual services.

Detail Discussion

Newsfeed — when it comes to newsfeed, it seems easy enough to implement, but there are lot of things that can make or break this feature. Lets divide the problem into two parts:

Generation
Lets assume we want to generate the feed for the user A, steps followed:
1. Retrieve the IDs of all 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, we have three 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 no. of write operations on our database.
— downside — users will not be able to view recent feeds unless they "pull" 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 follower's feeds immediately. This prevents the system from having to go through a user entire followers list to check for updates.
downside — it would increase the no. of write operations on the database.

→ Hybrid Model (between the pull and push model)
— combines beneficial features of the above two models and tries to provide a balanced approach between the two.
— allows only users with a lower no. of followers to use the push model. For users with a higher no. of followers such as celebrities, the pull model is used.

Ranking Algorithm — as discussed, we need a ranking algorithm to rank each tweet according to the relevance to each specific user.

for eg: facebook used to utilize an edgeRank algorithm. Here rank of each feed item is described by:

Rank = Affinity * Weight * Decay

where affinity → closeness of user to the creator of the edge. If a user frequently likes, comments or messages the edge creator, then value of affinity will be higher, resulting in a high rank for the post.
weight → value assigned according to each edge. A comment can have 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
— it is one of our extended requirements.
— To implement this feature, we can simply create a new tweet with the userId 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 eg: type enum property can be of type tweet, similar to text, video, image etc and content can be the id of the original tweet.

— this is a very basic implementation. To improve this, we can create a seperate 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 real 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 engines for all types of data, including textual, numerical, geospatial, structured, and unstructured. 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 records 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 messaging queue or a messaging broker such as 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.

Detailed Design

Discussion of design decision in detail:

Data Partitioning
— Horizontal partitioning (sharding) can be a good first step. We can use partition schemes such as:
— hash-based — list-based — range-based — composite partitioning
— Above approaches can still cause uneven data and load distribution, we can solve this using consistent hashing.

Mutual Friends
— For mutual friends, we can build a social graph for each user.
— each node in graph will represent a user and 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 will 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 sort 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 app, we have to be careful about using a cache as our users expect latest data.
— so to prevent usage spikes from our resources we can cache the top 20% of the tweets.

Cache-eviction policy
— we can use solutions like Redis or Memcached and cache 20% of the daily traffic but what kind of cache eviction policy?
— LRU can be a good choice. We discard 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 new entries.

Media access and storage
— our media service will be handling both access and storage of user media files.
— object storage — 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 use distributed file storage such as HDFS or GlusterFS.

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

Identify and resolve bottlenecks
— 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 cost?

To make system more resilient:—
— running multiple instances of each of our services.
— introducing load balancers between clients, servers, databases and cache servers.
— using multiple read replicas for our distributed caches.
— exactly once delivery and message ordering is challenging in a distributed system, we can use a dedicated message broker such as NATS/Kafka 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.