Twitter System Design

Twitter is a social media platform where users can post short messages called "tweets" and interact with each other. These tweets can be up to 280 characters long and can include text, photos, or links. Users can also follow other users and receive notifications when they post new tweets. In this article, we will discuss how to design a system similar to Twitter. First, we will outline the key requirements for our system.

Key Requirements

Before designing a system like Twitter, it's important to identify the essential features and the requirements we want to meet with our design. In this article, we will cover the following features:

- Posting tweets: Users should be able to create and post tweets, which may include text, photos, or links.
- Tweet delivery: The service should be able to push new tweets to the followers of the user who posted them and send push notifications to alert followers of the new tweets.
- Timeline view: Users should be able to view their own timeline (a feed of their own tweets) as well as the home timeline (a feed of tweets from the accounts they follow).
- High availability: The service should be highly available, meaning it should be able to handle a large number of users and requests without experiencing downtime.

Capacity Estimation

Assuming that our Twitter system has:

- 0.5 billion total users.
- 100 million daily active users (DAU).
- 50 million new tweets are created every day.

The system would generate a total of 4 billion tweet views per day, assuming that, on average, a user visits their timeline 1 time a day, visits 3 other people's pages, and each page has 10 tweets. This is calculated using the formula: The total number of tweet views/day = Number of DAU * (Number of timeline visits per day + Number of other people's pages visited per day) * Number of tweets per page = 100 million * (1 + 3) * 10 = 4 billion.

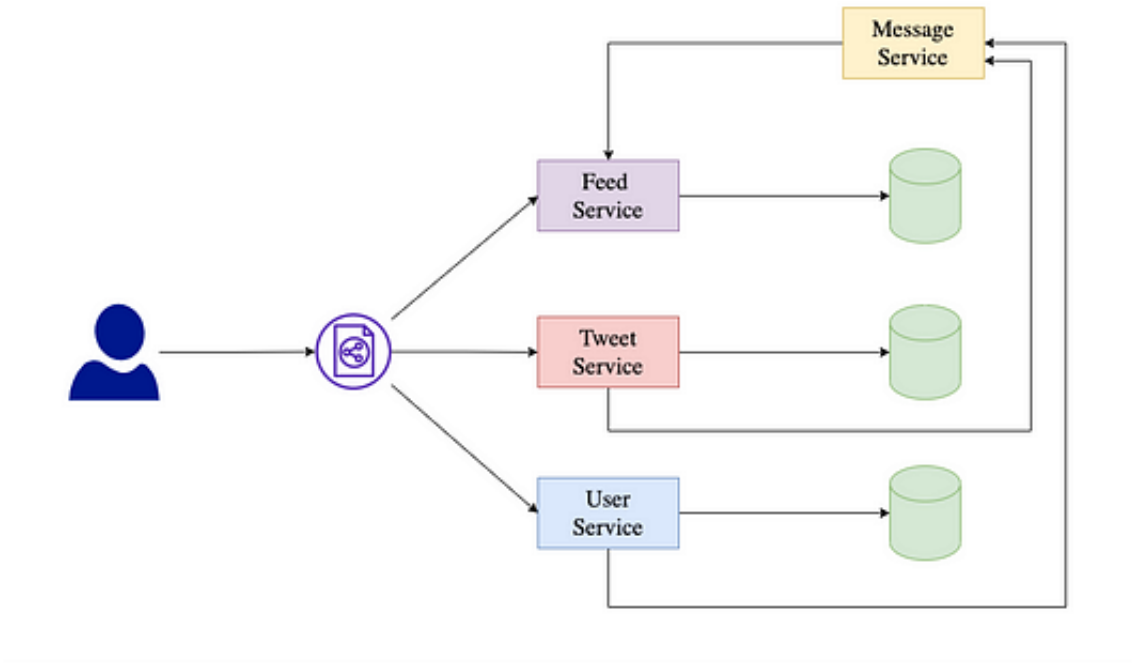
So our twitter system is more read-heavy than write-heavy, meaning it receives more requests to read tweets than to create them. Therefore, it is critical to optimize for fast reads of tweets.

Assuming that each tweet is 10 KB, the total daily storage needed would be approximately 5 TB. This calculation is done using the formula: Total storage needed per day = Tweet size * Number of tweets per day = 10KB * 50 Million = 5 TB. For 10 years of storage, we need 10 years * 365 days/year * 5 TB/day = 18.5 PB.

High Level Design

At a high level, a Twitter-like service requires a system with multiple application servers to handle client requests. These servers are placed behind load balancers, which distribute incoming traffic evenly among the servers. On the backend, a powerful database is needed to store tweets and support a high number of reads. In addition, the system needs file storage to store photos and videos

Our platform’s general high-level architecture is as follows:



The platform offers a range of services that are designed to handle specific functions. To support all the intended use cases, each service provides a set of APIs that can be used by client apps or other services. These services also have their own data stores for storing relevant data.

When a service receives a request, such as a new tweet or a request to follow a user, it validates the request and stores the data in its data store. In addition, the service sends a message in the form of an event to the messaging service, which can be used by other services to update their state. In this way, the platform is able to handle a wide range of use cases and ensure that data is stored and processed efficiently.

Data model and storage

When deciding on a data storage solution for a large platform with many users, it is important to consider both SQL and NoSQL options. However, due to scalability issues with SQL solutions, a NoSQL solution may be more suitable. In this case, we can use a wide column data store like Cassandra to store data such as tweets, comments, and likes.

For entities like users and their followers, we may consider using a graph-based data store solution like Neo4j or Cassandra. By using the appropriate data store for each type of data, we can ensure that the

system is able to handle a large volume of data and scale effectively.

Detailed Components Design

Our platform’s major system components are as follows:

User service: This provides APIs for managing users and their interactions, such as following or unfollowing other users. In some cases, it may be beneficial to create a separate User Follower Service to handle these interactions. This separation of responsibilities can help to ensure that the system is scalable and easy to maintain.

Tweet service: This provides APIs for creating and storing tweets in the data store and managing comments and likes on tweets. Depending on the needs of the application, this service may be further divided into Tweet Comments and Tweet Like Services to improve organization and scalability.

Feeds service: This is responsible for calculating the tweets that will appear on a user's timeline. It has an API that returns these tweets so that the timeline can be displayed, and it also supports pagination to allow users to browse their timeline in chunks. This makes it easier for users to navigate through their timeline and view the tweets that are most relevant to them. The service also has the ability to calculate the timeline quickly and efficiently, ensuring that the system can scale to support a large number of users.

Timeline generation: To display a user's timeline, the system must show the most recent posts from the user's followers. Generating the timeline on demand can be slow for users with many followers, as the algorithm must query, merge, and rank a large number of tweets. To improve performance, the system should pre-generate the timeline and store it in memory. This allows the timeline to be retrieved quickly when the user loads the page, rather than having to generate it on demand.

To ensure that the pre-generated timelines are regularly updated, the

system can use dedicated servers to constantly create and store them in memory. By using this approach, users' timelines are not compiled on demand, but rather regularly updated and stored in memory for quick retrieval.

Timeline updates: If the system generates timelines for all users in the same way, users with many followers may experience delays in receiving new postings. To address this issue, the system can prioritize users with new updates when generating timelines. When a new tweet is added to the message queue, it is picked up by timeline generator services, which re-generate the timeline for all followers. This approach allows users to receive new postings quickly, even if they have a large number of followers.

There are several approaches that can be used to publish new posts to users:

- **Pull model:** In this approach, clients can request data regularly or whenever they need it manually. The disadvantage of this method is that new information is not displayed until the client submits a pull request. Additionally, most pull requests will return an empty response if there is no new data, which can waste resources.
- **Push model:** With this approach, the system sends a notification to all of a user's followers as soon as they post a tweet. However, this technique can be challenging when a user has millions of followers, as the server must simultaneously transmit updates to a large number of individuals.
- **Hybrid model:** A combination of the pull and push models can be used to balance the benefits and drawbacks of each approach. For example, the system could provide updates directly to users with only a few hundred or thousands of followers, while celebrities with a larger number of followers could rely on their fans to disseminate updates. This hybrid approach can help to ensure that the system is able to scale and support a large number of users efficiently.

System Optimization

In this section, we will explore various optimization strategies to improve the performance, scalability, redundancy, and other non-functional aspects of our architecture. These approaches will help us ensure that the system is able to meet the needs of a large number of users and provide a reliable, high-quality experience.

Database sharding

Given the large number of users and tweets, it is not feasible to store all of the data for a service on a single machine. To address this challenge, it is important to define a partitioning strategy to divide the data into smaller sections, or partitions, which can be replicated and stored on different cluster hosts. This process, known as data sharding, helps to improve the speed of data nodes by reducing the number of nodes that must be searched in order to find a particular document.

There are several different approaches that can be used to split the data, including partitioning by user, by tweet, or by some other criterion. By carefully considering the needs of the application and the characteristics of the data, it is possible to design an effective partitioning strategy that helps to ensure that the system is able to scale and support a large number of users efficiently.

Sharding based on UserID

One possible approach to data partitioning is to link each user to a specific server based on their UserID. This strategy keeps all of a user's tweets, likes, and followers, among other things, on the same server. However, this approach can be problematic when some users (such as celebrities) are particularly popular, as it can result in a disproportionate amount of data and access being concentrated on a subset of servers. To address this issue, it may be necessary to adopt a different partitioning strategy that is better suited to the characteristics of the data and the needs of the application.

Sharding based on TweetID

Another approach to data partitioning is to map each tweet to a specific server based on its TweetID. This technique stores the tweet information on the server that is responsible for that particular tweet. To find tweets, the system must query all servers and receive a collection of tweets from each one. While this approach helps to avoid the problem of hot users by distributing the data evenly across servers, it can increase latency because all servers must be queried in order to find a particular tweet. As a result, this technique may not be the best option for applications that require fast search performance.

Sharding based on tweet creation time

Storing tweets based on the time they were created allows us to quickly retrieve the most recent tweets by querying only a small number of servers. However, this approach can result in uneven traffic load distribution. For example, when writing new tweets, all incoming traffic will be directed to a single server, while the other servers will be idle. On the other hand, when reading tweets, the server holding the most recent data may experience higher load compared to servers holding older data, leading to performance issues and inefficiency in handling large amounts of traffic.

Database caching

Caching is a useful technique for improving the performance and scalability of a system. It works by storing precomputed data in a cache, which can be accessed quickly in response to requests for that data.

The Feed Service is a good candidate for introducing caching in order to improve the speed of the system. One option for storing cached data is to use a solution such as Redis or Memcache. These solutions can be used to cache the chronology for each user, potentially storing the entire user timeline or a significant portion of it.

By preloading the cache with this data, it is possible to reduce the latency

when a user accesses their feed, improving their experience. To further improve performance, the system can preemptively request and cache the next batch of feed data for a user as they are browsing their feed. This can help to reduce the time it takes to load new tweets and enhance the overall user experience.

Load balancing

To ensure that incoming requests are evenly distributed across multiple data centers, we can use DNS load balancing. This is particularly useful for applications that receive requests from a variety of geographic regions. Each service should have a load balancer in front of it to distribute requests to various service nodes based on capacity estimates.

There are several load balancing methods that can be used, such as round-robin and least connection, to determine how to distribute the load across the available nodes. Additionally, because our services are stateless, we can add or remove nodes from the cluster without losing any data, making it easier to scale the system as needed.

Replication and Fault Tolerance

To optimize for read-heavy workloads, we can use multiple slave database servers for each partition of the database. These slave servers will only be used for handling read traffic, while all write operations will be directed to the master database and then replicated to the slave database. This approach not only helps to distribute read traffic, but also provides fault tolerance.

Conclusion

In this blog post, we discussed how to design a system similar to Twitter. We covered topics such as how to handle client requests, how to store and manage data, and how to optimize the system for performance and scalability. We hope you enjoyed reading this blog post and learned something new.

Thanks to Suyash Namdeo for his contribution in creating the first version of this content. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!