Interview Daemon

Follow 9 Followers About

System Design: Twitter

Social Blogging Website

Interview Daemon Jan 2 · 8 min read

Source: *interviewdaemon.com (Link)*

Twitter is a microblogging and social networking service on which users post and interact with messages known as “tweets”. Registered users can post, like and retweet tweets, but unregistered users can only read them.

Introduction

This system will allow users to post tweets with other users. Additionally, users can follow other users based on follow request and they can see other user’s timeline.

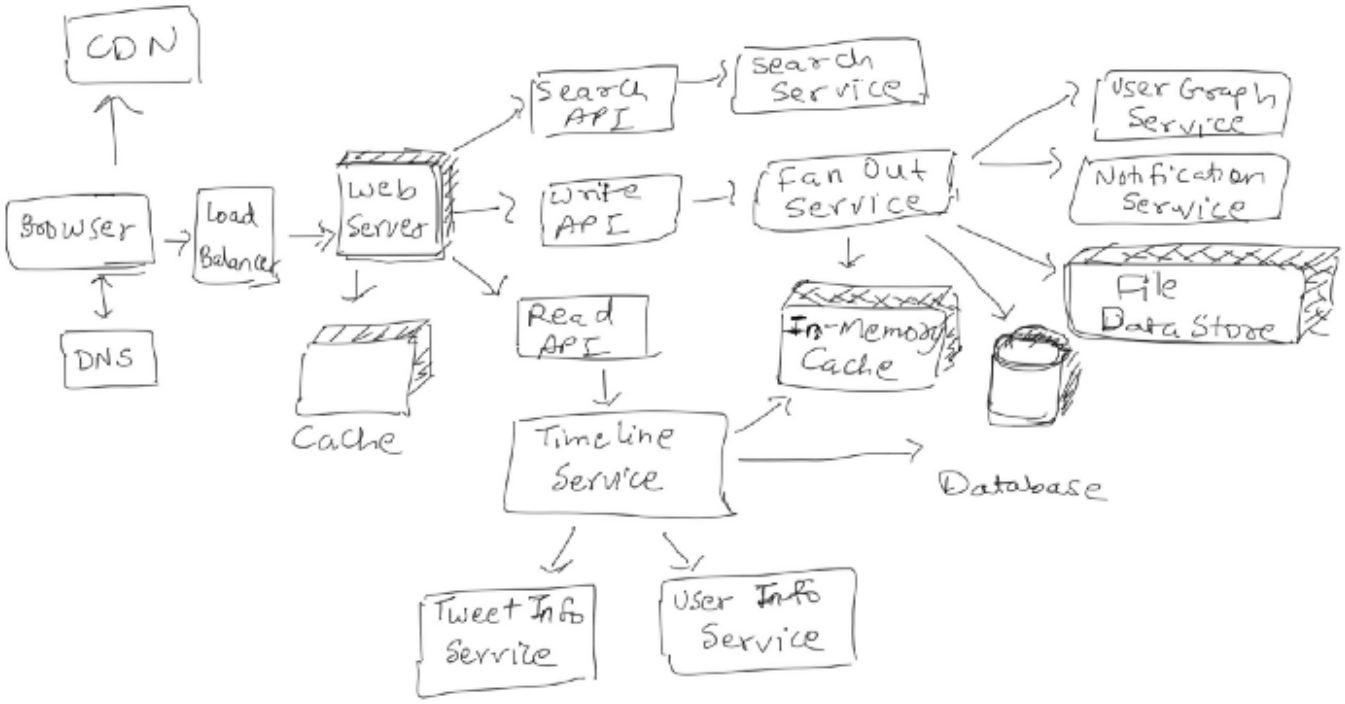
Requirements

- Users must be able to **post a tweet.**
- Users must be able view their **own timeline (Home Timeline).** Home Timeline is activity from people the user is following.
- Users must be able to **view other timeline (User Timeline).** User Timeline is activity from other user when you visit their page.

- Users must be able to **search keywords**.
- User must be able to view **search timeline (Search Timeline)**. Search Timeline displays search result based on tags or search keyword.
- System should be **highly available**
- System should be **highly reliable**
- System should be **durable**
- System should be **resilient**
- Service will be both **write-heavy and read-heavy**.
- Service will be **consistent** (Eventual Consistency is fine) and reliable which means there should not be any data loss.
- Service will be durable which means all piece of system should exists until they are delete manually.

High Level System Design

At a high-level, we need to support multiple micro-services in the current system.



API Design

We can use REST or SOAP to serve our APIs. Basically, there will be below important API's of photo and video sharing service system.

1. postTweet

```
postTweet(api_dev_key, tweetDescription, user_id, media_id)

curl -X POST --data '{ "user_id": "123", "auth_token":  "ABC123",
"status": "hello world!", "media_id": "ABC987" }'
https://twitter.com/api/v1/tweet
```

PostTweet will responsible for posting tweet. api_dev_key is the API developer key of a registered account. We can eliminate hacker attacks with api_dev_key. This API returns HTTP response. (202 accepted if success)

2. getUserTimeline

```
getTimeline(api_dev_key, user_id, start_offset, limit = 20)
```

Return JSON containing information about the list of tweets.

3. searchQuery

```
searchQuery(api_dev_key, query, start_offset, limit = 20)
```

Return JSON containing information about the list of tweets based on query keywords.

Database

Twitter uses MySQL (Relational Database) as primary database. It stores information such as user data, photo, tags, meta-tags etc. As the platform grew bigger they moved to more microservice model keeping the database as SQL but started to shard(partition) the database across multiple servers. (Database Sharding)

- We can use **Redis** to store in-memory database is used to store the recent user tweets and timelines, sessions & other app's real-time data. (Twitter uses Redis)
- **Memcache/Redis** can be used for distributed memory caching system is used for caching throughout the service. (Twitter uses Memcache and Redis)
- We can store photos in a distributed file storage like HDFS or S3 (Twitter user **Blobstore**)
- **MetricsDB** for storing platform data metrics
- **FlockDB** for storing social graph

Component Design

1. Tweet Posting Service

Tweet can be posted using the postTweet API which is stored in MYSQL database Tweet table. Once the postTweet API is used to post a tweet it is later rendered at couple of places. Tweets are displayed in Home Time of the user, Other User Timeline which follows the current user and if someone search for same keyword or hashtag. However getting the tweet from database is a costlier operation and so we have to find a mechanism to store the tweets In-Memory Cache (Redis) or NoSQL database.

The new tweet would be placed **In-Memory Cache**, which populates the user's Home Timeline (activity from people the user is following).

2. Home Timeline Service

- The **Client** posts a home timeline request to the **Web Server**
- The **Web Server** forwards the request to the **Read API** server
- The **Read API** server contacts the **Timeline Service**, which does the following:
- Gets the timeline data stored in the **Memory Cache**, containing tweet ids and user ids — (Constant Lookup Time: O(1))
- Queries the **Tweet Info Service** with a multiget to obtain additional info about the tweet ids — (Lookup Time with N Tweets: O(n))
- Queries the **User Info Service** with a multiget to obtain additional info about the user ids — (Lookup Time with K Users: O(k))

Celebrity Problem:

If a person with millions of the followers **(Celebrity**) on Twitter tweets, We need to update millions of lists which is not very scalable. We have to update the Memory Cache for all the followers of that particular celebrity so that they have the tweet in their timeline.

Solution:

- Precomputed Home Timeline of User A with everyone except celebrity tweet(s) from the In-Memory Cache
- Get the list of celebrity from Database which user is following and get their tweets from In-Memory Cache or Database. Merge other user tweets with celebrity tweet at load time.
- For every user have the mapping of celebrities list and mix their tweets at runtime when the request arrives and cache it.

3. User Timeline Service

The REST API for User Timeline would be similar to the Home Yimeline, except all tweets would come from the user as opposed to the people the user is following.

- The **Web Server** forwards the request to the **Read API** server
- The **Read API** retrieves the user timeline from the **SQL Database**

4. Search Service or Top Trends

- The **Client** sends a search request to the **Web Server**
- The **Web Server** forwards the request to the **Search API** server
- The **Search API** contacts the **Search Service**, which does the following:
- Parses/tokenizes the input query, determining what needs to be searched

Removes markup

- Breaks up the text into terms
- Fixes typos
- Normalizes capitalization
- Converts the query to use boolean operations
- Queries the **Search Cluster** (ie Lucene) for the results:
- Scatter gathers each server in the cluster to determine if there are any results for the query
- Merges, ranks, sorts, and returns the results

4. Account Service

This service is solely responsible for creation and deletion of user details. This can call MySQL and update the User table. Can also user the "**PROFILE_VISIBILITY**" boolean to

make user profile public/private based on user preferences.

5. Follow User Service

This service is responsible to keep track of followers a user has and profiles which a user follows. **UserFollow** table is used which keeps track of people a user follows. We can also keep this in Cassandra or any key value stores.

6. Notification Service

Asynchronous tasks such as sending notifications to the users & other system background processes are done using task queue and message broker.

- We can use an asynchronous task queue based on distributed message communication, focused on real-time operations. It supports scheduling too.
- A message broker use to push the content in queue and pull it later.

7. Monitoring

With so many instances powering the service, monitoring plays a key role in ensuring the health & availability of the service.

Network & infrastructure monitoring tool used by Twitter to track metrics across the service & get notified of any anomalies.

Data Sharding (Data Partitioning)

Twitter does generate its own unique id and also shards the database.

Database Partitioning

1. Partition based on User_ID

Here we can keep all tweets of a user on the same shard. So we'll find the shard number by **User_ID % 10** and then store the data there. To uniquely identify any tweet in our system, we can append the shard number with each tweet_id.

Problems:

- How would we handle hot users? Several people follow such hot users, and a lot of other people see any tweet they post.

- Some users will have a lot of tweets compared to others, thus making a non-uniform distribution of storage.
- What if we cannot store all tweets of a user on one shard? If we distribute tweets of a user onto multiple shards, will it cause higher latencies?
- Storing all tweets of a user on one shard can cause issues like unavailability of all of the user's data if that shard is down or higher latency if it is serving high load etc.

2 Partition based on Tweet_ID

If we can generate unique Tweet_ID's first and then find a shard number through "Tweet_ID % 10", the above problems will have been solved. We would not need to append ShardID with Tweet_ID in this case, as Tweet_ID will itself be unique throughout the system.

This approach solves the problem of hot users, but, in contrast to sharding by UserID, we have to query all database partitions to find tweets of a user, which can result in higher latencies.

3. Sharding based on Tweet creation time

Storing tweets based on creation time will give us the advantage of fetching all the top tweets quickly and we only have to query a very small set of servers. The problem here is that the traffic load will not be distributed, e.g., while writing, all new tweets will be going to one server and the remaining servers will be sitting idle. Similarly, while reading, the server holding the latest data will have a very high load as compared to servers holding old data.

4. Sharding by TweetID and Tweet creation time

If we don't store tweet creation time separately and use TweetID to reflect that, we can get benefits of both the approaches. This way it will be quite quick to find the latest Tweets. For this, we must make each TweetID universally unique in our system and each TweetID should contain a timestamp too.

We can use epoch time for this. Let's say our TweetID will have two parts: the first part will be representing epoch seconds and the second part will be an auto-incrementing

sequence. So, to make a new TweetID, we can take the current epoch time and append an auto-incrementing number to it.

If we assume our current epoch seconds are “1483228800,” our TweetID will look like this:

1483228800 000001
1483228800 000002
1483228800 000003
1483228800 000004

Cache and Load Balancing

We can introduce a cache for metadata servers to cache hot database rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system. (We can use Memcache / Redis)

How can we build a more intelligent cache? If we go with the eighty-twenty rule, i.e., 20% of daily read volume for tweets is generating 80% of the traffic, which means that certain tweets are so popular that most people read them. This dictates that we can try caching 20% of the daily read volume of tweets and metadata.

References:

- DonneMartin
- Naren Gowda — Medium
- Naren Gowda — Youtube

Interview Daemon System Design Interview System Design Twitter