Designing Twitter Search

Twitter is one of the largest social networking service where users can share photos, news, and text-based messages. In this chapter, we will design a service that can store and search user tweets. Similar Problems: Tweet search. Difficulty Level: Medium

---

1. What is Twitter Search?

Twitter users can update their status whenever they like. Each status (called tweet) consists of plain text and our goal is to design a system that allows searching over all the user tweets.

2. Requirements and Goals of the System

- Let's assume Twitter has 1.5 billion total users with 800 million daily active users.
- On average Twitter gets 400 million tweets every day.
- The average size of a tweet is 300 bytes.
- Let's assume there will be 500M searches every day.
- The search query will consist of multiple words combined with AND/OR.

We need to design a system that can efficiently store and query tweets.

3. Capacity Estimation and Constraints

Storage Capacity: Since we have 400 million new tweets every day and each tweet on average is 300 bytes, the total storage we need, will be:

400M * 300 => 120GB/day

Total storage per second:

120GB / 24hours / 3600sec ~= 1.38MB/second

4. System APIs

We can have SOAP or REST APIs to expose functionality of our service; following could be the definition of the search API:

Parameters:

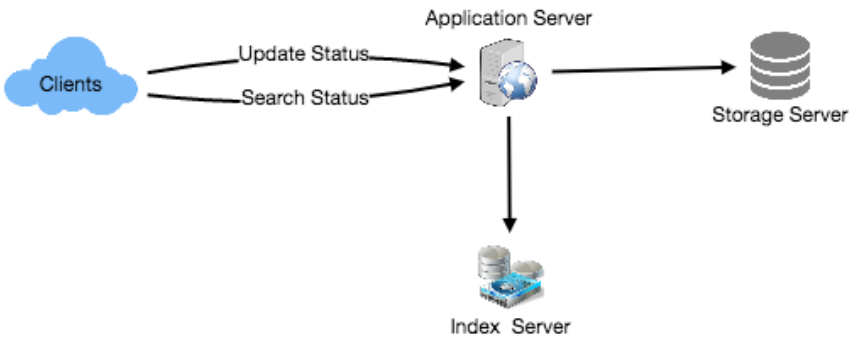
api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.
search_terms (string): A string containing the search terms.
maximum_results_to_return (number): Number of tweets to return.
sort (number): Optional sort mode: Latest first (0 - default), Best matched (1), Most liked (2).
page_token (string): This token will specify a page in the result set that should be returned.

Returns: (JSON)

A JSON containing information about a list of tweets matching the search query. Each result entry can have the user ID & name, tweet text, tweet ID, creation time, number of likes, etc.

5. High Level Design

At the high level, we need to store all the statues in a database and also build an index that can keep track of which word appears in which tweet. This index will help us quickly find tweets that users are trying to search.



High level design for Twitter search

6. Detailed Component Design

1. Storage: We need to store 120GB of new data every day. Given this huge amount of data, we need to come up with a data partitioning scheme that will be efficiently distributing the data onto multiple servers. If we plan for next five years, we will need the following storage:

120GB * 365days * 5years ~= 200TB

If we never want to be more than 80% full at any time, we approximately will need 250TB of total storage. Let's assume that we want to keep an extra copy of all tweets for fault tolerance; then, our total storage requirement will be 500TB. If we assume a modern server can store up to 4TB of data, we would need 125 such servers to hold all of the required data for the next five years.

Let's start with a simplistic design where we store the tweets in a MySQL database. We can assume that we store the tweets in a table having two columns, TweetID and TweetText. Let's assume we partition our data based on TweetID. If our TweetIDs are unique system-wide, we can define a hash function that can map a TweetID to a storage server where we can store that tweet object.

How can we create system-wide unique TweetIDs? If we are getting 400M new tweets each day, then how many tweet objects we can expect in five years?

400M * 365 days * 5 years => 730 billion

This means we would need a five bytes number to identify TweetIDs uniquely. Let's assume we have a service that can generate a unique TweetID whenever we need to store an object (The TweetID discussed here will be similar to TweetID discussed in Designing Twitter). We can feed the TweetID to our hash function to find the storage server and store our tweet object there.

2. Index: What should our index look like? Since our tweet queries will consist of words, let's build the index that can tell us which word comes in which tweet object. Let's first estimate how big our index will be. If we want to build an index for all the English words and some famous nouns like people names, city names, etc., and if we assume that we have around 300K English words and 200K nouns, then we will have 500k total words in our index. Let's assume that the average length of a word is five characters. If we are keeping our index in memory, we need 2.5MB of memory to store all the words:

500K * 5 => 2.5 MB

Let's assume that we want to keep the index in memory for all the tweets from only past two years. Since we will be getting 730B tweets in 5 years, this will give us 292B tweets in two years. Given that each TweetID will be 5 bytes, how much memory will we need to store all the TweetIDs?

292B * 5 => 1460 GB

So our index would be like a big distributed hash table, where 'key' would be the word and 'value' will be a list of TweetIDs of all those tweets which contain that word. Assuming on average we have 40 words in each tweet and since we will not be indexing prepositions and other small words like 'the', 'an', 'and' etc., let's assume we will have around 15 words in each tweet that need to be indexed. This means each TweetID will be stored 15 times in our index. So total memory we will need to store our index:

(1460 * 15) + 2.5MB ~= 21 TB

Assuming a high-end server has 144GB of memory, we would need 152 such servers to hold our index.

We can shard our data based on two criteria:

Sharding based on Words: While building our index, we will iterate through all the words of a tweet and calculate the hash of each word to find the server where it would be indexed. To find all tweets containing a specific word we have to query only the server which contains this word.

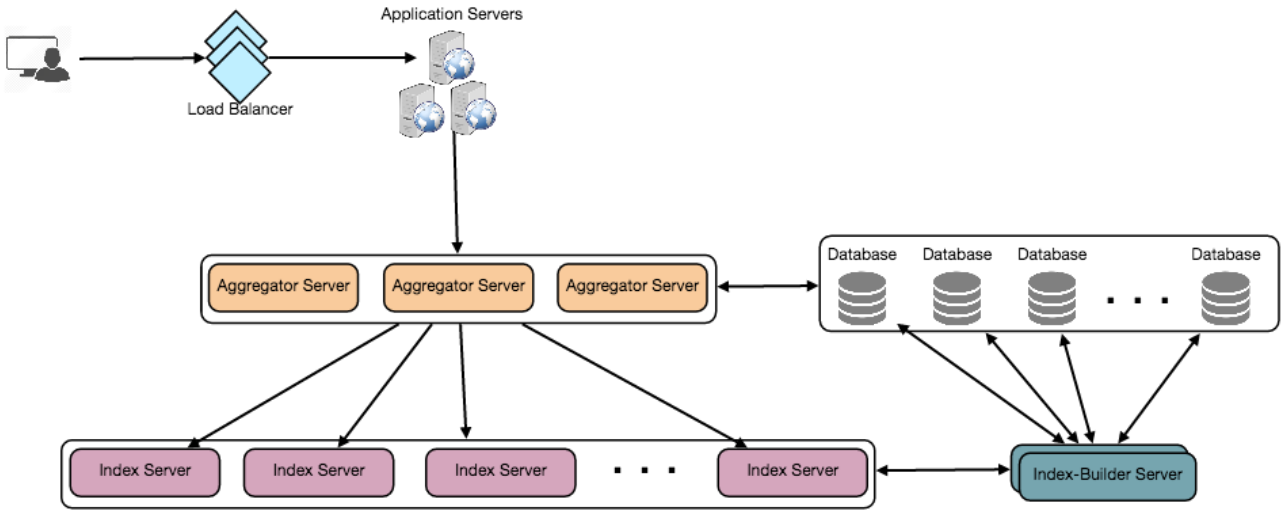
We have a couple of issues with this approach:

1. What if a word becomes hot? Then there will be a lot of queries on the server holding that word. This high load will affect the performance of our service.
2. Over time, some words can end up storing a lot of TweetIDs compared to others, therefore, maintaining a uniform distribution of words while tweets are growing is quite tricky.

To recover from these situations we either have to repartition our data or use Consistent Hashing.

Sharding based on the tweet object: While storing, we will pass the TweetID to our hash function to find the server and index all the words of the tweet on that server. While querying for a particular word, we have to query all the servers, and each server will

return a set of TweetIDs. A centralized server will aggregate these results to return them to the user.



Detailed component design

7. Fault Tolerance

What will happen when an index server dies? We can have a secondary replica of each server and if the primary server dies it can take control after the failover. Both primary and secondary servers will have the same copy of the index.

What if both primary and secondary servers die at the same time? We have to allocate a new server and rebuild the same index on it. How can we do that? We don't know what words/tweets were kept on this server. If we were using 'Sharding based on the tweet object', the brute-force solution would be to iterate through the whole database and filter TweetIDs using our hash function to figure out all the required tweets that would be stored on this server. This would be inefficient and also during the time when the server was being rebuilt we would not be able to serve any query from it, thus missing some tweets that should have been seen by the user.

How can we efficiently retrieve a mapping between tweets and the index server? We have to build a reverse index that will map all the TweetID to their index server. Our Index-Builder server can hold this information. We will need to build a Hashtable where the 'key' will be the index server number and the 'value' will be a HashSet containing all the TweetIDs being kept at that index server. Notice that we are keeping all the TweetIDs in a HashSet; this will enable us to add/remove tweets from our index quickly. So now,

whenever an index server has to rebuild itself, it can simply ask the Index-Builder server for all the tweets it needs to store and then fetch those tweets to build the index. This approach will surely be fast. We should also have a replica of the Index-Builder server for fault tolerance.

8. Cache

To deal with hot tweets we can introduce a cache in front of our database. We can use Memcached, which can store all such hot tweets in memory. Application servers, before hitting the backend database, can quickly check if the cache has that tweet. Based on clients' usage patterns, we can adjust how many cache servers we need. For cache eviction policy, Least Recently Used (LRU) seems suitable for our system.

9. Load Balancing

We can add a load balancing layer at two places in our system 1) Between Clients and Application servers and 2) Between Application servers and Backend server. Initially, a simple Round Robin approach can be adopted; that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is LB will take dead servers out of the rotation and will stop sending any traffic to it. A problem with Round Robin LB is it won't take server load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about their load and adjust traffic based on that.

10. Ranking

How about if we want to rank the search results by social graph distance, popularity, relevance, etc?

Let's assume we want to rank tweets by popularity, like how many likes or comments a tweet is getting, etc. In such a case, our ranking algorithm can calculate a 'popularity number' (based on the number of likes etc.) and store it with the index. Each partition can sort the results based on this popularity number before returning results to the

aggregator server. The aggregator server combines all these results, sorts them based on the popularity number, and sends the top results to the user.

Have questions?

Get help on educative DISCUSS

Completed

← *Previous* Designing an API Rate Limiter *Next* → Designing a Web Crawler

Send feedback 24 Recommendations