Functional Requirement

User should be able to search tweets hashtags

The system should give results matching the search keywords.

Non-Functional Requirement

The Twitter search should be highly available to provide search service.

The search service should have minimum latency.

Capacity Estimation

Storage Capacity:

Let assume that 600 Million searches will happen every day for tweets

Total Tweets For 5 years = 600 * 365 * 5 million = 1095 billon

Avg Size of Each Tweets – 150 bytes(Assuming 1 character is 1 bytes)

Total Storage required for tweets → 600M * 150 => 90 GB/day

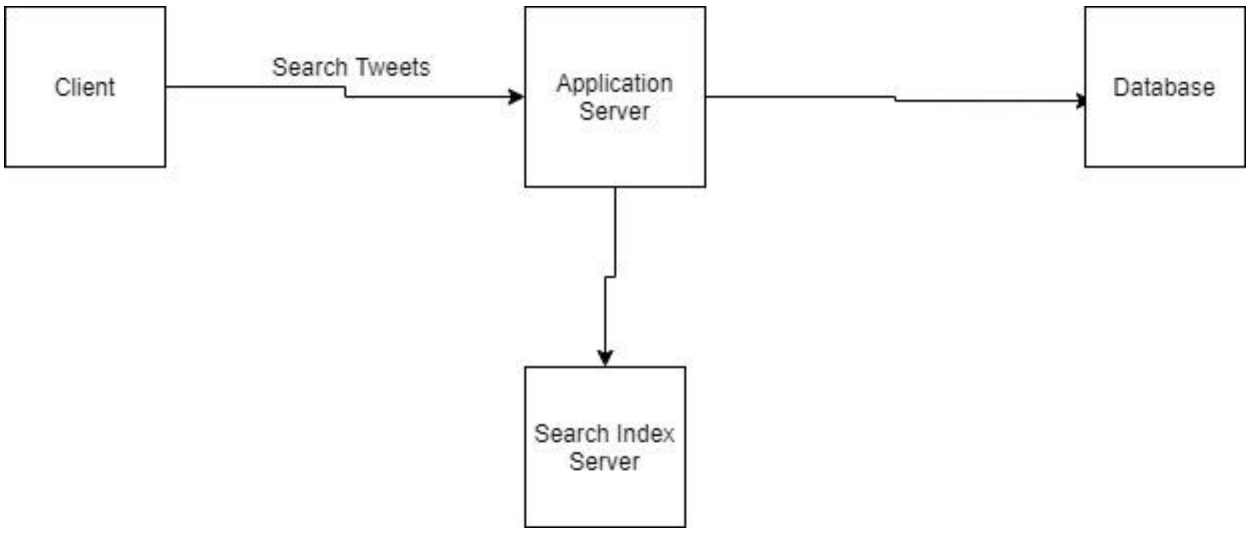
Total Storage for 5 years = 90 * 365 * 5 = 164.5 TB

Total storage per second: 90 GB / 24hours / 3600sec ~= 1.04 MB/seconds

System API

search(search terms, maximum_results_to_return, sort, page token)

High Level Design



Database Design

- Require storage for billions of records
- Smaller Object Size(less than 1K)
- No relation exists between tweets.
- Service is read heavy

Tweet

Field	Type
Tweet id	Integer
Tweet content	String
Created data	timestamp
Total number of favorites	Number

Keep above nature of data, we propose to use NoSQL database such as DynamoDB, Mongo DB, Cassandras for our service.

Basic System Design

Index Server

Index server is most import component of our system. Indexes are used in database for faster retrieval of data. Indexed data is stored separate from the database and is maintained in memory

We can build index on all the English words and all the nouns of th .

Total Estimate for Estimating Data =

Average size of word in tweet = 5

Total English words= 171460 + 1500 = 172960 words

Total Memory to store all words = 172960 * 5 = 0.85 MB

Total storage for indexing data = 1095 * 5 = 5475 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 10 words in each tweet that need to be indexed. This means each TweetID will be stored 10 times in our index. So total memory we will need to store our index:

(5475 * 10) + 0.85MB ~= 21 TB

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

Data Partitioning and Replication

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 on Tweet

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.

Caching

We can introduce a cache for database servers to cache hot tweets and users. Also Application servers, before hitting database, can quickly check if the cache has desired tweets. Based on clients’ usage patterns we can determine how many cache servers we need. Keeping above requirements ,we propose to utilize Memcache for our system.

Intelligent Cache

80-20 rule can help us build intelligent cache. It means 20% of tweets is generating 80% of daily traffic which means certain tweets are most popular and liked by majority people. So, we can cache these 20% of daily read of volume of data and tweets to build intelligent cache. Our service can benefit from this approach.

Our cache would be like a hash table where ‘key’ would be ‘OwnerID’ and ‘value’ would be a doubly linked list containing all the tweets from that user in the past three days. Since we want to retrieve the most recent data first, we can always insert new tweets at the head of the linked list, which means all the older tweets will be near the tail of the linked list. Therefore, we can remove tweets from the tail to make space for newer tweets.

Load Balancing

We can add a Load balancing layer at three places in our system:

1. Between Clients and Application servers
2. Between Application Servers and database servers
3. Between Application Servers and Indexing Servers

We can use simple round robin mechanism for load balancing ,it does not take the server load into consideration. So, we can use Agent-Based Adaptive Load Balancing approach for load balancing Each server in the pool has an agent that reports on its current load to the load balancer. This real-time information is used when deciding which server is best placed to handle a request. This is used in conjunction with other techniques such as Weighted Round Robin and Weighted Least Connection.