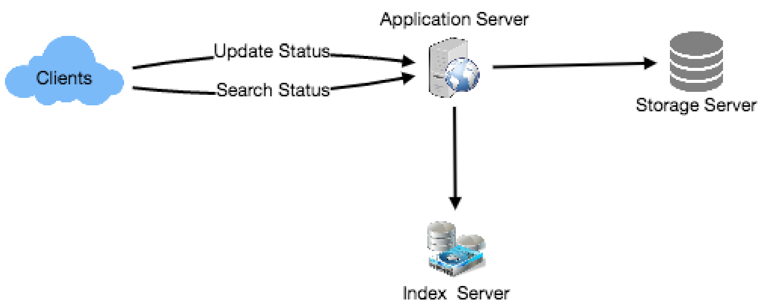
I. Designing Twitter Search

1. twitter search-tweet search service
2. requirements and goal of the system
    a. Build an effective tweet storage / navigation system
3. capacity estimations and constraints
    a. assume
        i. average incoming tweets : 400M / day
        ii. average size of tweet : 300 byte / tweet
    b. estimation
        i. storage
            - 400M * 300byte = 120GB / day
            - 120GB / 86400sec ~= 1.38MB / sec
4. system apis - soap or rest api
    a. search (api_dev_key, search_terms, maximum_result_to_return, sort, page_token)
        i. sort (number) : optional, sort mode
            - Latest first (0 - default)
            - Best matched (1)
            - Most liked (2)
        ii. page_token (string): Specify the page to be returned in resultSet
        iii. return (json)
            - Returns tweets that match search_terms
            - UserID, Name, TweetText, TweetID, CreationDate, Likes, etc
5. high level design



   a. We need to create an index to store all the states in the db and track which words are in which tweets.
   b. index Purpose: To improve tweet search speed
6. component design
   a. storage
      i. 120GB of new data is stored every day-> Considering massive data, a data partitioning scheme that can be efficiently stored on a distributed server is required
      ii. What are your plans for the next five years?
         - 120GB * 365 * 5 ~= 200TB
         - Assuming that only 80% of HDD capacity is used, 250 TB HDD capacity is required
         - Assume to store a copy of all tweets for fault tolerance-> Hard Require 250KB * 2 = 500TB
         - Assuming a modern server stores 4TB of data, 125 servers are required
      iii. tweet storage workflow
         - Stored in rdbms-> Stored in a table with two columns, TweetID and TweetText-> Map to storage server with TweetID based Sharding-> hash function (TweetID% number of server)
      iv. How can I create a unique TweetID?
         - If you store 400M new tweets every day, how many tweets can you expect in 5 years?
            ✓ 400M * 365 * 5 ~= 730B
         - Assuming that there is a server that can generate a unique TweetID when saving a tweet-hash function is TweetID% number of servers
   b. index
      i. The tweet query is composed of word units, so an index is created to indicate which word is in which tweet.
      ii. assume

- Create indexes for famous (highly used) nouns such as all English words, people's names, city names, etc.
- About 300K English words and 200K nouns
- Average word length: 5 characters

iii. estimation

- 500K * 5 = 2.5MB when storing all words in memory
- Suppose that index of all tweets in the last 2 years is kept in memory
  - ✓ 292B (730B / 5 * 2) tweet / 2years
  - ✓ Assuming TweetID is 5 bytes, 292B * 5 = 1460GB
- index Similar to a large distributed hash table
  - ✓ key : word
  - ✓ value : TweetID
  - ✓ Assume the average number of words per tweet is 40 words / tweet-> 15 words / tweet because prepositions and small words such as 'an' and 'and' are not indexed
  - ✓ 1460GB * 15 + 2.5MB ~= 21TB
  - ✓ Assuming that the server has 144GB memory, 152 (21000/144) servers are required

c. database partitioning

i. word based sharding

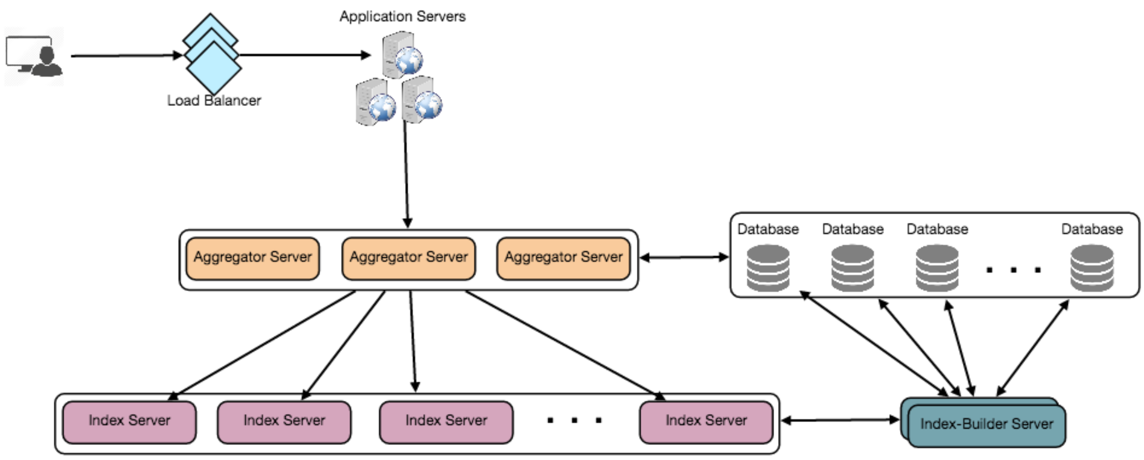
- Search all words in tweet-> calculate hash for each word and store it in the corresponding index server
- To find all tweets that contain a specific word, just query the server where the word is stored
- Disadvantages

- ✓ Hot word processing problem: The server that holds the word has a lot of load-> This load affects the entire system (slow)
- ✓ Non-uniformity of distribution: Different weight of each word-> Difficult to maintain uniform distribution

- Solution
    - ✓ consistent hashing (**일관된 해싱**)
    - ✓ repartition (from scratch again)

ii. tweet object based sharding

- Find the server by passing the TweetID to the hash function-> indexing all the words in the tweet on that server-> you must query all the servers to query a specific word-> each server returns the TweetID Set-> the user after the central server counts Return to



7. fault tolerance
    a. index server What happens when I die?
        i. If it is composed of master-slave, slave acts as master
        ii. Both master and slave must have the same index copy
    b. So what happens if both master and slave die at the same time?

i. Service Suspension
ii. Restore
    - Assign a new server and rebuild the same index-because I don't know what word / tweet is kept on that server
    - When using tweet object based sharding
        - ✓ brute-force solution : Search all the entire db again, and identify all tweets to be stored on the server through the hash function with TweetID.
    - Disadvantages
        - ✓ Inefficient
        - ✓ Service unavailable when server is rebuilding
    - How can I search the mapping between Tweet and index server efficiently?
        - ✓ Create a reverse index that maps all TweetIDs to the index server-> The information is stored by the index build server
        - ✓ key : index server
        - ✓ value : Maintained in the index server
        - ✓ TweetID hash table to create a hash set that includes all of them.
        - ✓ Operates a replica of the index build server for fault tolerance

8. cache
    a. Introduced cache before db for hot tweet processing
    b. If you use Memcached, all tweets can be stored in memory.
    c. Application server quickly checks whether there is a tweet stored in the cache before reaching the backend db (system speed is greatly improved)
    d. The number of required cache servers can be adjusted according to the client's usage pattern
    e. cache eviction policy : LRU (Least Recently Used)

9. load balancing
    a. Location
        i. Between client and web server
        ii. Between web server and application server
        iii. Between application server and db server
    b. Initial Policy: round-robin system
        i. Advantages
            - Simple implementation and no overhead
            - Even distribution of requests between backend servers
            - Stop sending traffic without replacing the dead server
        ii. Disadvantages
            - server load check failed.
            - Do not stop sending traffic even if a particular server is overloaded
        iii. Solution
            - Check the load periodically on the backend server and deploy intelligent LB solution to adjust traffic based on this

10. ranking
    a. How can I rank search results by social graph distance, popularity, relevance, etc.?
        i. Assume that you rank tweets based on the popularity and number of comments
        ii. ranking algorithm
            - Create and index popular number fields based on the number of people you like and store them
            - The aggregator server aggregates all these results, sorts them (popularity) and sends the best results to the user