System Design — Twitter Search

source: quanta magazine

> *Design a system for implementing Search functionality for a site like Twitter*

Similar Questions

- Design Autocomplete Search
- Design Google Search
- Design graph search for a social networking website

Functional Requirements

- Users must be able to search tweets— show the top 100 results.
- Search results need not be personalized i.e. it is expected that the same search query will return the same results for 2 different users, simulteneously querying the system.
- Search results should show the latest and trending tweets.
- Search results should be ranked in reverse chronological order.

301 | 8

- **Availability** — Search functionality should be always available
- **Low Latency** — Search results should be returned with very low latency (50ms)
- **Reliability** — Latest tweets should be available for search within 10s of creation.

Traffic and Throughput Requirements

- Number of DAU = 500 million
- Number of search queries per second = 1 million
- Average length of search queries = 10
- Number of tweets per second = 10K
- Peak number of tweets per second = 1 million
- Maximum length of tweets = 140 characters

The "On My Computer" Approach

Idea is to create an **inverted index of the terms** in the tweets mapping to the Tweet ids.

Assuming that we have a service for storing and retrieving tweets, when a new tweet comes in, first insert it into the tweet store, retrieve the tweet_id as well as the original tweet, then tokenize the tweet into words or terms.

Create a HashMap A: term -> [list of tweet_ids]

> *Update the list of tweet_ids by appending the latest tweet at the end of the list corresponding to the term.*
>
> *For a given search query, again tokenize it into terms, lookup HashMap A to retrive multiple lists of tweet_ids, one corresponding to each term.*
>
> *Then do an union operation on the tweet_ids and return 100 tweet_ids to the tweet service which will find the tweets corresponding to these tweets.*

How to ensure time sorted list of tweets ?

> *The tweet_id is comprised of 2 components: UNIX Timestamp in Milliseconds (41 bits) + 10 bit auto-incrementing integer.*

Since there could be 1 million tweets per second i.e. 1000 tweets per second, thus in the same millisecond there could be 1000 different tweets. We can store them in a 10 bit (1024 possible integers) integer.

Thus tweet_ids are already sorted by time in the above list.

What is the data structure for storing the list of tweet_ids ?

Note that tweets can be also deleted by accounts.

One simple data structure that can allow O(1) insertion and deletion is doubly linked list.

> *But in Linked List memory is not contiguously assigned (random memory access) and the memory overhead for creating a Node pointer object is high.*

To avoid referring the term in multiple places. We will maintain an auto-incrementing term_id. Thus we have the following HashMaps:

> *A: term -> term_id*
>
> *B: term_id -> [id_0 -> id_1 -> id_2 -> .... -> id_M] (doubly linked list)*
>
> *C: term_id -> DLL Tail Address*
>
> *D: hash(term_id+tweet_id) -> DLL Node Address*

The last HashMap is required in order to delete a Node in O(1) time complexity from the inverted index.

The 3rd HashMap is required to get tweet ids in reverse chronological order.

The other option is using **arrays** which allows sequential memory access which is faster as compared to random access.

> *But the drawback with array based approach, is that we do not know beforehand how many tweet_ids will be there for a term. If we pre-allocate a small array, then if the number of tweet_ids is more, then we have to either increase the length of the array or create a new array to store the future elements.*

If the size of the arrays are small then we have to resize too often. All reads and writes are blocked during the time we will be migrating to a larger array. But if the size of the array is too large then there will be lots of wasted memory addresses which will have to be cleaned up explicitly with a GC before we run into OOM.

One strategy is to create array slices of size 1, 2, 4, 8, 16, … and so on.

> *For each term, we have K slices. If the term occurs for 1st time, then we create the first slice of size 1. If the term occurs next time, then we create a slice of size 2 and insert the new tweet_id in that slice. Similarly if the term occurs in more than 7 tweets, we create a new slice of slice 8 and insert the new tweet_id in the first empty slot from the beginning and so on.*
>
> *Note that the slices need not be contiguous in memory as they are created as and when required wherease each slice is contiguously laid out in memory.*

To track which slice and which index in the slice to insert the new tweet_id, we can maintain a HashMap: term -> (curr_slice_number, curr_position)

The slice_number and position are updated with each insertion of term in the inverted index.

> *A: term -> term_id*
>
> *B: term_id -> [[id_0], [id_1, id_2], [id_3, id_4, id_5, NULL]]*
>
> *C: term_id -> (curr_slice_number, curr_position)*

But deletion from inverted index is not efficient with this strategy.

In order to support efficient deletion and editing of tweets, we can go ahead with the DLL approach.

What happens when a tweet is edited ?

When a term is changed in the tweet, for the new term, standard insertion process is followed.

For the terms removed, for each of those terms and tweet_id, lookup the HashMap:

hash(old_term_id+tweet_id) -> DLL Node Address

To fetch the node pointers in the inverted index. Then delete those nodes from the DLLs to update the inverted index.

> *This multistep process for a single term is a single transaction and thus for any failure, we have to undo all previous changes accordingly.*

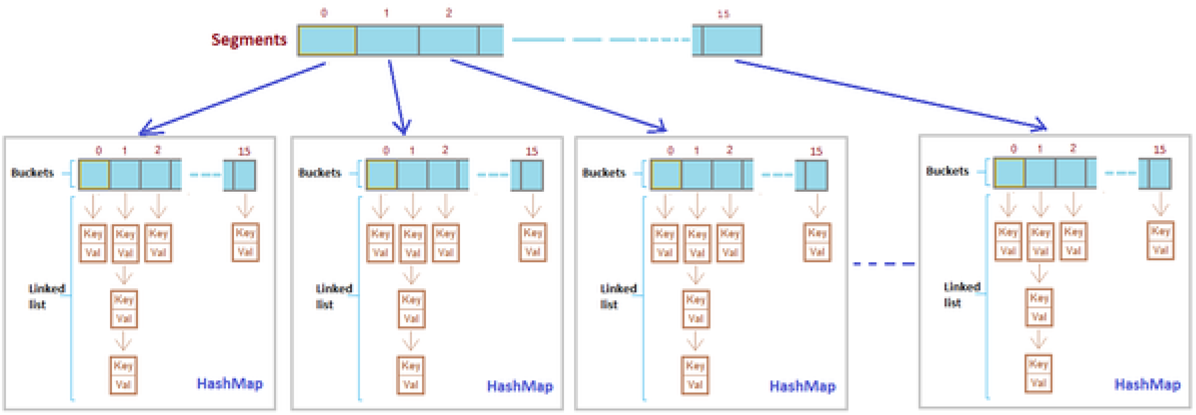
How to handle concurrent read-write ?

If one process is updating the inverted index for a term and another process wants to read the term simulateneously, there are few edge cases:

- New tweet_id was added by the write process but **term_id-> DLL Tail Address** is not updated yet, thus read process will read from old tail backwards and will miss the newly added node.
- Tweet edited and before write process could update inverted index, read process reads the removed node in the inverted index and assumes that the term is present.

> *Since mutexes are acquired at the HashMap level and not at individual key level, thus the entire Map becomes unavailable during writes. Assuming we have 10K writes per second, that would slow down all reads.*

We can use something similar to the **ConcurrentHashMap** implementation in JAVA.



Segments in Concurrent HashMap

Divide the HashMap arrays into **equal sized segments**.

> *For e.g. if the HashMap array is of length 1024, then we can have 32 segments of size 32 each. These segments are independent.*
>
> *During write only one segment will be in use by a thread and thus just acquire lock for that segment only whereas all the other 31 segments are free to be read/write by the other threads.*

Thus for an index j in the HashMap array, the segment is obtained by performing int(j/32) and the index within the segment is j%32.

During addition of a new tweet_id, for each term_id acquire write lock on the segment int(hash(term_id)/32).

> *During insertion, the lock is acquired on the hashmap "term_id-> DLL Tail Address" in the appropriate segment before writing the inverted index so that all threads trying to get the tail for term_id will wait for the lock to release, meanwhile any other threads that accesses segments other than int(hash(term_id)/32), will be able to read/write without waiting.*

```
lock segment int(hash(term_id)/32) of hashmap term_id-> DLL Tail
Address {
    get tail node
    update inverted_index by adding new node and update tail
    update "term_id-> DLL Tail Address" hashmap with new tail
    release_lock()
}
```

During edit of a tweet_id, for each old_term_id acquire write lock on the segment int(hash(old_term_id)/32) for all the following hashmaps:

> *term_id -> [id_0 -> id_1 -> id_2 -> …. -> id_M]*
>
> *term_id -> DLL Tail Address*
>
> *hash(term_id+tweet_id) -> DLL Node Address*

Note: we need to lock all the hashmaps because the tail may also be updated during edits.

```
lock segment int(hash(old_term_id)/32) of all hashmaps {
    update tweets table for tweet_id
    get relevant node address from hash(old_term_id+tweet_id)
    delete node in inverted index
    update tail node if required in "term_id-> DLL Tail Address"

    remove node from hash(old_term_id+tweet_id)
    release_lock()
}
```

What are the drawbacks of the above approach ?

If the system crashes all HashMaps are lost thus system becomes unavailable.

Number of tweets per second = 10K

Number of tweets for 5 years = 1.6 trillion

> *Assuming a power law distribution of terms i.e. say if the highest occuring term occurs in x tweets, then the 2nd highest occurs in m*x tweets for m < 1, the next highest in m^2*x tweets and so on.*

Thus $x + xm + xm^2 + xm^3 + \ldots + xm^{(k-1)}$= 1.6 trillion for k terms

$x*(1-m^k)/(1-m)$ = 1.6 trillion

For large enough k, x/(1-m) = 1.6 trillion

For x = 1 million (we can always exclude all terms which occurs in more than 1 million tweets), we have m = 0.999999375

Also xm^(k-1) ≥ 1 (i.e. at-least one tweet) or k is approximately 22 million

Thus number of unique terms is approximately equal to 22 million (i.e. term_id is of 25 bits)

Assuming that average number of terms per tweet is 15.

> *Size of the inverted index = Number of terms*25 bits + Number of tweets*15*51 bits = 22 million*25 bits + 1.6 trillion*15*51 bits = 140TB.*

140TB is very huge to hold on a single RAM.

Distributed Hash Tables

How many instances of the inverted index are required to store 140TB ?

Assuming 16GB machine, total partitions = 140*1024/16 = 8960.

How to partition the inverted index ?

> *Assign integers 0 to K-1 to the K partitions. Then based on the value of P = hash(term_id) % K, assign term_id to partition P.*

How many replicas for the inverted index ?

Using replicas serves 2 purposes:

1. Fault tolerance — Each partition has multiple copies thus even if one copy (machine) crashes other copies can still serve the request.
2. Load Balancing — With high throughput of 1 million QPS, one machine for a partition may not be sufficient. If one machine can handle X QPS, then with M machines we can handle M*X QPS.

The expected search QPS is 1 million at peak. Assuming that we have 8 core CPU machines and search latency of 50ms. Then QPS served from one machine is:

QPS single machine = 8*1000/50 = 160

> *To serve 1 million QPS, we need minimum of 1 million/160 machines i.e. 6250 machines.*

But earlier we have seen that to store 140TB we need at-least 8960 machines (one partition of 16GB in each machine). Thus we have 8960 partition and in order to achieve fault tolerance, lets say we have 3 replicas for each partition thus in total we have 8960*3 machines = 27K

> *With 27K machines, throughput we can achieve is 27K*160=4.3 million QPS which is good enough.*

What is the replication strategy for the inverted index ?

> *For each partition we should have a single master and 2 replicas.*
>
> *During writes, all writes happen at the master node, which is then replicated to the 2 replicas. All reads happen from both master and replicas.*

If we have all master nodes (multi-master) i.e. writes can happen in any of the 3 nodes, then there could be inconsistencies. It could happen that 2 concurrent requests for the same term_id sees different states (in different master nodes) and thus each process will update based on their master state and this could lead to conflicts.

But indexing is faster with multi-master since the tweets can be indexed into the nearest inverted index server from the user. But given a delay of 10s, we can live with single master for now.

Although within a single master node we can handle concurrency as seen above.

With a single master node, all writes are consistent and there are no issues related to conflicts. We can say that a write is successful only if the master is updated successfully and any one replica node is updated also.

During read, we read from any 2 nodes (master+replica1 or master+replica2 or replica1+replica2). Then based on whichever has the higher last_updated timestamp, we consider results from that instance.

> *The choice of at-least 2 nodes for write and 2 nodes for reads is that during write even if one node is not updated successfully (maybe it went offline), during reads at-least one node is there which was successfully updated during writes.*

How tweets are ingested ?

> *Tweets are ingested into a Kafka queue for processing it into the inverted index. The use of queue is to make the* ***indexing flow asynchronous*** *because clients need not have to wait for the inverted index to be sucessfully updated before getting a response that their tweet is sucessful.*

With an expected delay of 10s between tweet ingested and tweet indexed, using some queueing theory, it can be shown that the delay is expressed as:

delay = 1/(QPS of consumer-QPS of producer)

Assuming QPS of producer is 10K (10K writes per second)

QPS of consumer = 1/delay + 10K/s = 10K/s

> *Assuming each consumer can handle 1000 QPS, we need 10 consumers to handle 10K tweets per second.*

How to generate IDs in distributed manner ?

Note that tweet ids are being generated by distributed system instead of a single machine. With our earlier approach of using IDs of the form:

> *UNIX Timestamp in Milliseconds (41 bits) + 10 bit auto-incrementing integer.*

It might happen that 2 machines generate the same ID. This could happen if 2 machines writing the new tweet at the same millisecond and both reads the highest auto-incrementing integer as X before both computing X+1 instead of one of them computing X+2.

This could be resolved by using another field in the ID formula which is the machine ID. We will keep 13 bits for the machine ID i.e. we are assuming that we will need maximum 8192 machines for distributed ID generation.

> *UNIX Timestamp in Milliseconds (41 bits) + 10 bit auto-incrementing integer + 13 bit machine ID = Total 64 bits.*

Thus if 2 machines generate the same ID one of them will have higher machine ID than the other thus breaking the tie.

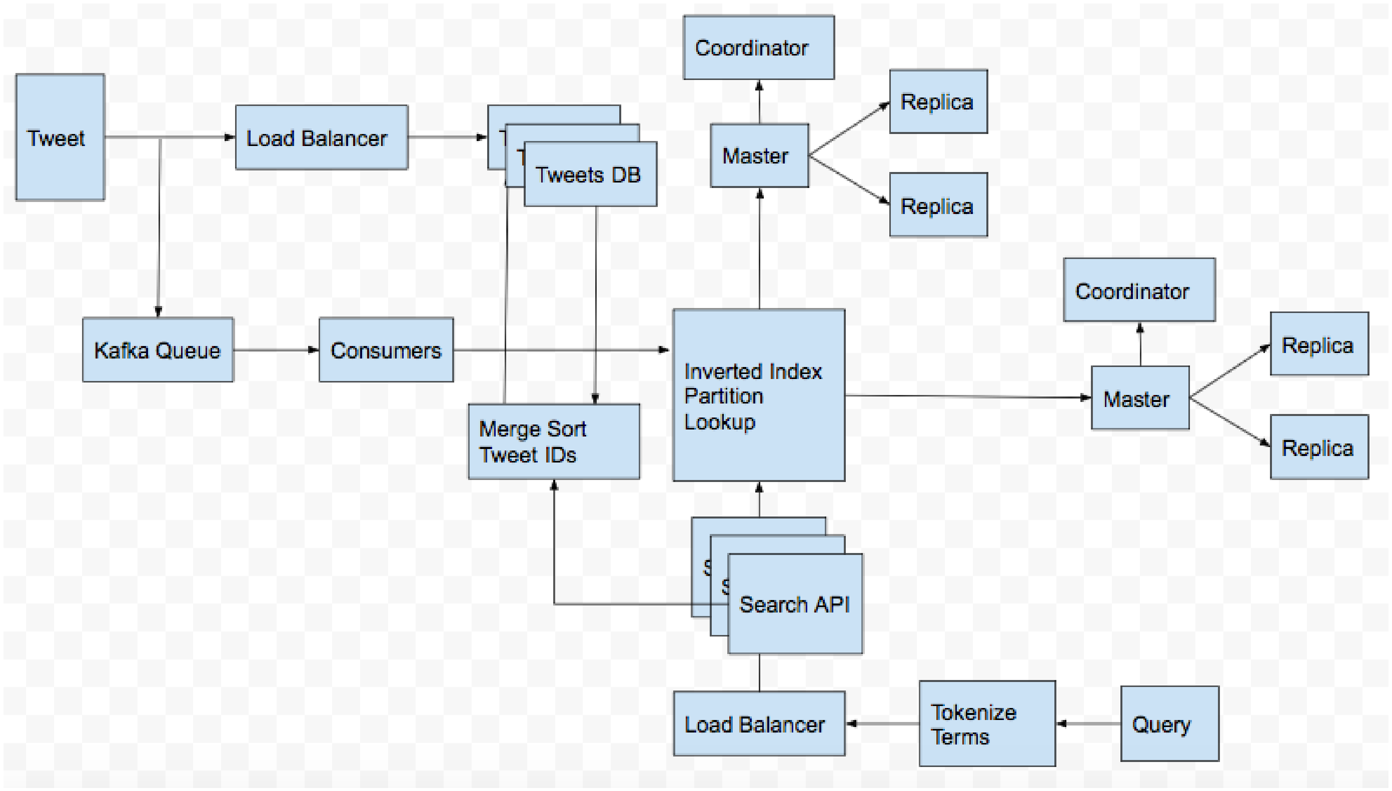
How to persist the inverted index and the other maps ?

Although we have taken care of fault tolerance by the use of replication, but it could still happen that all replicas for a partition crashes leading to loss of data for search.

To overcome this, we can consider the following approaches:

- Recreate the partition from tweets table. This can take take somewhere from few hours to few days depending on the size of data. Thus the system will not work correctly during this period.
- Run perioding jobs to persist the partition in a filesystem or DB. The use case for DB is not really there as we need to scan all rows to recreate the partition. Thus storing each row of partition as a row in a file seems more cost effective as well as useful.

The Pipeline



Useful Resources

- http://notes.stephenholiday.com/Earlybird.pdf
- https://www.javainuse.com/java/javaConcurrentHashMap
- https://dzone.com/articles/how-concurrenthashmap-works-internally-in-java
- https://web.mit.edu/modiano/www/6.263/lec5-6.pdf