Functional Requirements

- When user starts type word, system should return top 10 results matching the prefix of typed words.
- Search should work in real time and require minimum latency.

Non-Functional Requirements

- Search results should take 200ms results to provide results
- Since search result should work in real time and takes less than 200ms , we should design the system to be highly available. If the system is not available, then performance might degrade. Therefore, system should be highly available.
- Consistency is not important because multiple users will be using our system.

Storage and Capacity Estimation

Capacity Estimation

Let assume 3-6 billion queries per day for estimation for search for our system.

Total Queries per day(Assuming average size) = 5 billion

Average length of the query – 5

Average word size → 5 words

Average Query Size = 5 * 5 = 25 characters

Total Queries per day = 25 * 5 = 125 billion

Storage Estimation:

Let assume total byte store for 1 character = 1 byte

Total storage for one query/Average size of query = 30 * 1 = 30 byte

Total Storage for Queries per day = 125 * 10^9 * 30 = 3.750 GB per day

Assuming 10 % of search queries are new i.e.

Total storage for 1 year = 3.75 + (3.75 * 10 * 365) = 140.625 GB

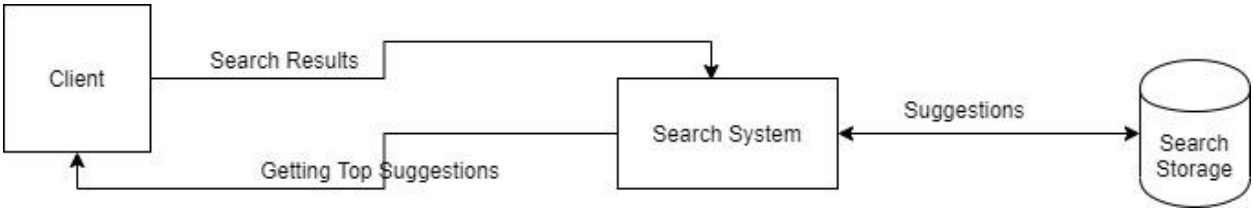
Total storage for 5 years = 140.63 * 5 = 703.12 GB

System API

Get Top Suggestion (current Query)

Update Suggestions(search Term)

High Level Design



Database Design

Word/Phrase/Term

Field	Type
Phrase	String
Weighted Frequency	Number
Created time	Timestamp

In addition to maintaining trie, we should search phrases in database. Since we will be storing millions of phrases per day, we propose to utilize Cassandra NoSQL to store our phrases.

Basic System Design and Algorithm

We need to store all string in such a way that our system can return top suggestions based on prefix match for give user query .We can use a Trie to store all string and return the top suggestions. Trie serves best purpose for string matching and pattern algorithms, A Trie is a tree like data structure used to store words . Each node of the Trie stores one character in sequential manner.

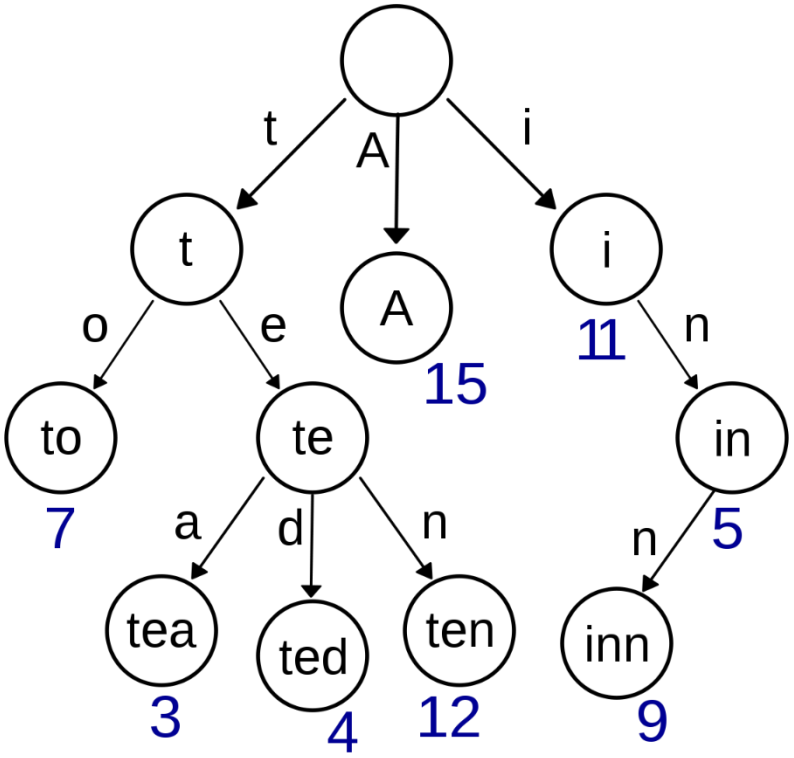


Figure 1 A Trie depicting storing multiple words

A problem with above problem is that it can lead to wastage to space to store millions of characters for billion queries .To avoid such space waste, we can merge node

Top Suggestion

Store count of searches that terminated at each node so that always top 10 suggestions will be returned.

Time Complexity for Search for Prefix

We can expect a huge tree given the number of queries and string data stores.

Average query will take searching 20-30 levels deep which might not work well given our strict latency requirement. If we store 10 suggestions with every node in our trie, it will work for real time suggestion, but it will require huge storage resulting in requirement of additional storage space in addition to the storage estimates. Instead we can store the reference of terminal nodes. To find the suggested terms we need to traverse back using the parent reference from the terminal node. We will also need to store the frequency with each reference to keep track of top suggestions.

Building Trie

We can efficiently build our trie bottom up. Each parent node will recursively call all the child nodes to calculate their top suggestions and their counts. Parent nodes will combine top suggestions from all their children to determine their top suggestions.

Update Trie

Assuming five billion searches every day, which would give us approximately 60K queries per second. Update trie for every query will be extremely resource intensive and hamper our read requests. One solution to periodic update of trie .This updation should be done offline.

As the new queries come in we can log them and track their frequencies. We can utilize Map-Reduce (MR) set-up to process all the logging data periodically i.e. hourly. These MR jobs will calculate frequencies of all searched terms in the past hour We can then update frequencies of all searched terms in the past hour. We can then update our trie with this new data. We can take the current snapshot of the trie and update it with all the new terms and their frequencies. We should do this offline as we do not want our read queries to be blocked by update trie requests. We can have two options:

1. Replicate the trie on each server for offline update. Once updation process is completed ,we can discard the old one and start using new .
2. We can utilize master-slave configuration for each trie server. We can update slave while the master is serving traffic. Once the update is complete, we can make the slave our new master. We can repeat this process for all the trie

Update Suggestion/Query Frequency

Since we are storing frequencies of our typeahead suggestions with each node, we need to update them too! We can update only differences in frequencies rather than recounting all search terms from scratch. If we are keeping count of all the terms searched in last 10 days, we will need to subtract the counts from the period no longer included and add the counts for the new time being included. We can add and subtract frequencies based on Exponential Moving Average (EMA) of each term. In EMA, we give more weight to the latest data. It is also known as the exponentially weighted moving average.

After inserting a new term in the Trie, we will go to the terminal node of the phrase and increase its frequency. Since we are storing the top 10 queries in each node, it is possible that this search term jumped into the top 10 queries of a few other nodes. So, we need to update the top 10 queries of those nodes then. We must traverse back from the node to all the way up to the root. For every parent, we check if the current query is part of the top 10. If so, we update the corresponding frequency. If not, we check if the current query's frequency is high enough to be a part of the top 10. If so, we insert this new term and remove the term with the lowest frequency.

Data Partitioning and Replication

a. Range Based Partitioning: What if we store our phrases in separate partitions based on their first letter. So, we save all the terms starting with the letter 'A' in one partition and those that start with the letter 'B' into another partition and so on. We can even combine certain less frequently occurring letters into one partition. We should come up with this partitioning scheme statically so that we can always store and search terms in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers, for instance, if we decide to put all terms starting with the letter 'E' into one partition, but later we realize that we have too many terms that start with letter 'E' that we can't fit into one partition.

b. Partition based on the maximum capacity of the server: In this scheme, we partition our trie based on the maximum memory capacity of the servers. We can keep storing data on a server as long as it has memory available. Whenever a sub-tree cannot fit into a server, we break our partition there to assign that range to this server and move on the next server to repeat this process.

We can have a load balancer in front of our trie servers which can store this mapping and redirect traffic. Also, if we are querying from multiple servers, either we need to merge the results on the server side to calculate the overall top results or make our clients do that If we prefer calculate the overall top results or make our clients do that. If we prefer to do this on the server side, we need to introduce another layer of

servers between load balancers and trie severs (let us call them aggregator). These servers will aggregate results from multiple trie servers and return the top results to the client.

Partitioning based on the maximum capacity can still lead us to hotspots, e.g., if there are a lot of queries for terms starting with 'cap', the server holding it will have a high load compared to others.

c. Partition based on the hash of the term: Each term will be passed to a hash function, which will generate a server number and we will store the term on that server. This will make our term distribution random and hence minimize hotspots. The disadvantage of this scheme is, to find typeahead suggestions for a term we must ask all the servers and then aggregate the results.

Caching

Cache is extremely beneficial to store frequently access word/phrases/terms. .We can use Memcache for caching in our system We can store frequently access words/phrases/terms and their suggestion in key-pair store. This not only reduces burden on database servers due to concurrent querying, but greatly enhance the system performance.

Load Balancing

Load balance can help us distribute the traffic to the shard and database servers. We can use different load balancing algos to distribute the traffic. Some of techniques are explained below :

Weighted Round Robin

Weighted Round Robin builds on the simple Round Robin load balancing method. In the weighted version, each server in the pool is given a static numerical weighting. Servers with higher ratings get more requests sent to them.

Least Connection

Neither Round Robin nor Weighted Round Robin take the current server load into consideration when distributing requests. The Least Connection method does take the current server load into consideration. The current request goes to the server that is servicing the least number of active sessions at the current time.

Agent-Based Adaptive 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.

Handling Edge cases in given system

- For fast retrieval, CDN can be used
- To save network space, the service can periodically send the in-progress requests.
- Clients can store the recent history of suggestions locally. Recent history has a very high rate of being reused