Overview

Design typeahead search system or k most searched queries.

Functional Requirements

- Support the typeahead search.
- Prefix searches autocomplete. If I type an, it should suggest top k search which have prefix an. Supported language is English.
- Top items will be updated weekly/daily (clear with interviewer)
- Query might have spaces.

Non-Functional Requirements

- Low latency
- Highly available
- Relevant and Sorted
- Scalable

Capacity Planning

- Daily active users = 10 million.
- Avg search by the user = 10.
- Assuming each query is 5 character long and total has 4 words. Total size will be 20 bytes (assuming ascii encoding).
- Total request sent to backend for each search term= 5 character long * 4 words = 20. Total request for each user = 10*20.
- QPS = 10million*10*20/ (86400) ~= 1million*2000/10^5 = 20000 QPS.

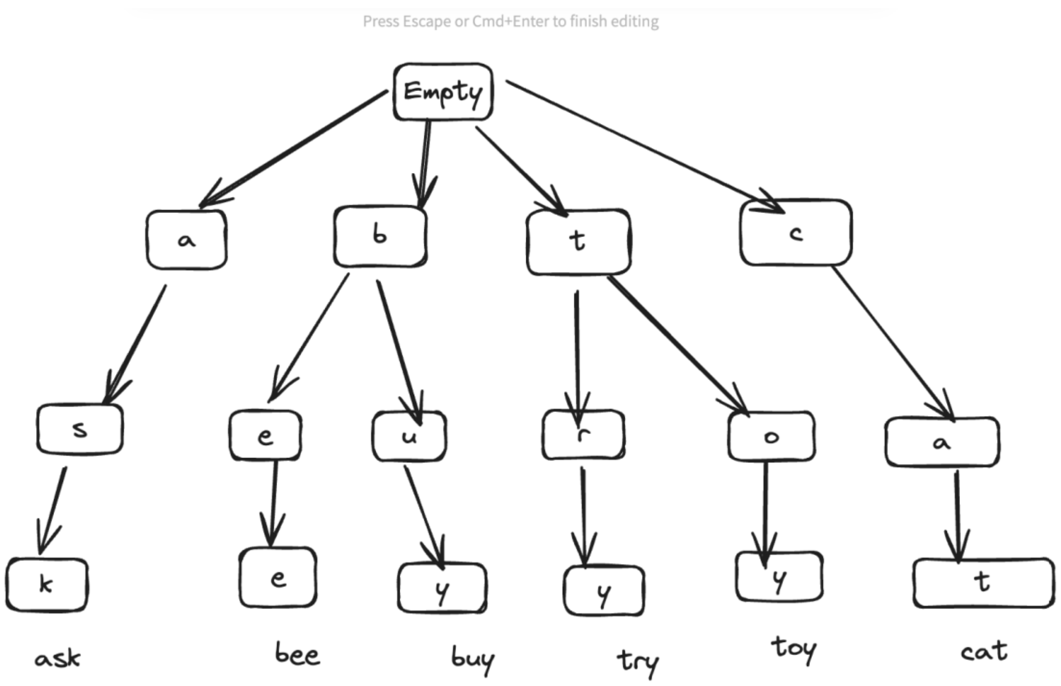
- Peak QPS = 2*QPS= 40k QPS.
- Assuming 10 per of search query are new. Total ingestion size =10 per of (10*10million = 100million) = 10million. Assuming 20 bytes = 10million*20=200million bytes = 0.2 GB.

Architecture

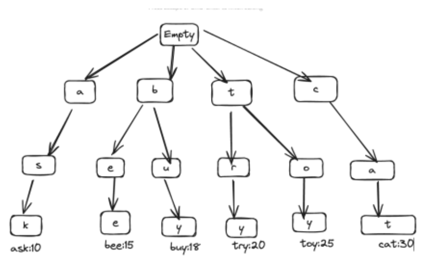
Basic Design

- Multiple servers will ingest and query the DB to get the data. Query will be something like

  select * from table where query like prefix% order by frequency desc limit k;
- If the data can fit in memory, we can use trie.



The above Trie stores all the aggregated string. It'll also need to store the frequency at the leaf.



What will the time complexity of traversing the Trie and gathering the result.
For a prefix of length L, if the number of child (direct and indirect) nodes is N.
Time complexity will be O(L+N) + sorting all the result. If number of child node is X.
Complexity will be Xlog(X). Total time complexity will be O(L+N) + Xlog(X).

How can we optimize it ?

We can optimize the traversing if we cache the top k result on each node. In this way we avoid O(L+N) factor above. Also, since the k value is small, storing and subsequent update operation of each node will be fast.

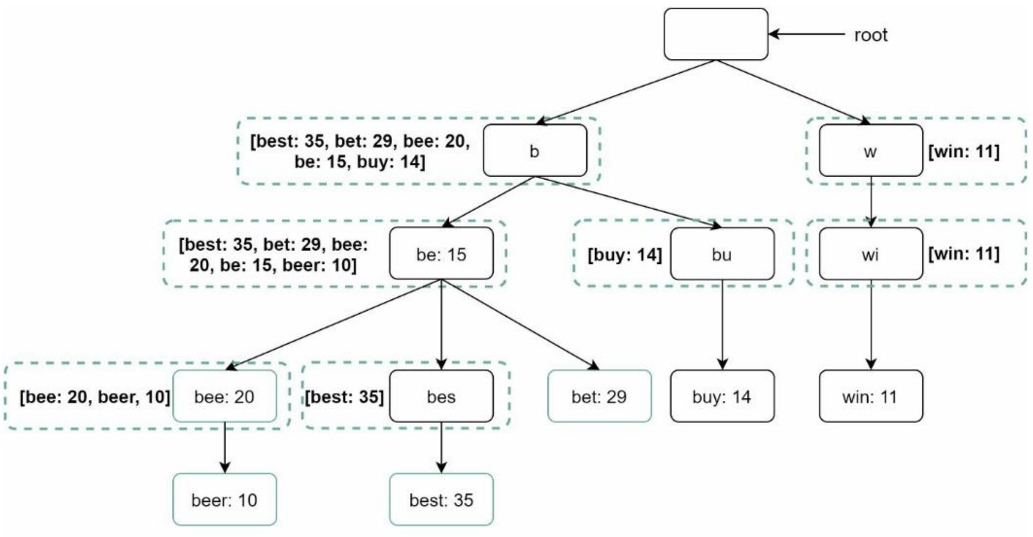
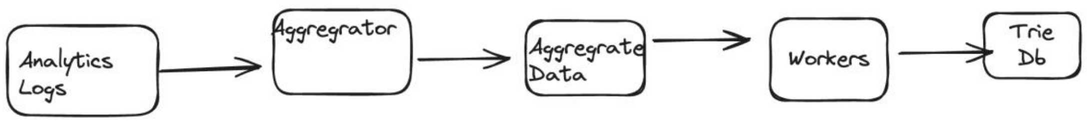


Figure 13-8

Fetching will be O (L) for finding the prefix node. Since top k are cached, all good on this side.
Downside of this -> Memory footprint. We can use dictionary compression (which takes the advantage of repeated words and assigned them a unique integer).
Write will be slower as now you need to update the top k for every node (which is okk because of offline processing).

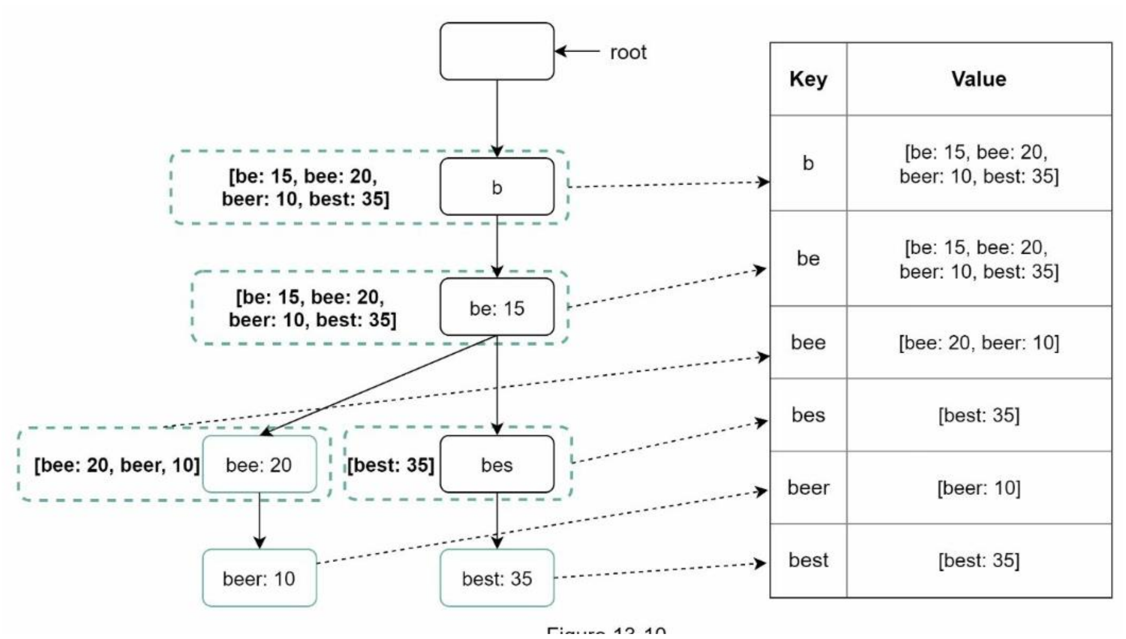
Data Gathering Component

- We will rely on the offline data processing.
  Architecture



- Analytics Logs -> Dump of all the search terms fired.
- Aggregator -> Might be spark job which will do the cleaning and aggregate the search term and dump this.
- Workers -> Another spark job which reads the aggregate data .After aggregating, data will look like search string along with frequency. workers is going to generate the prefix and get list of top k items and update the DB.
  E.g if the search term contains anuj and anay for e.g, worker will have top k items for a -> anuj,anay.
- DB ->Prefix can be represented in the form of key value pair. Each key is going to be prefix and value is going to be list of top k string along with its frequency.
  We can use Cassandra (column wide with custom sharding and sorting on

prefix) in this case because of its high read and write throughput.

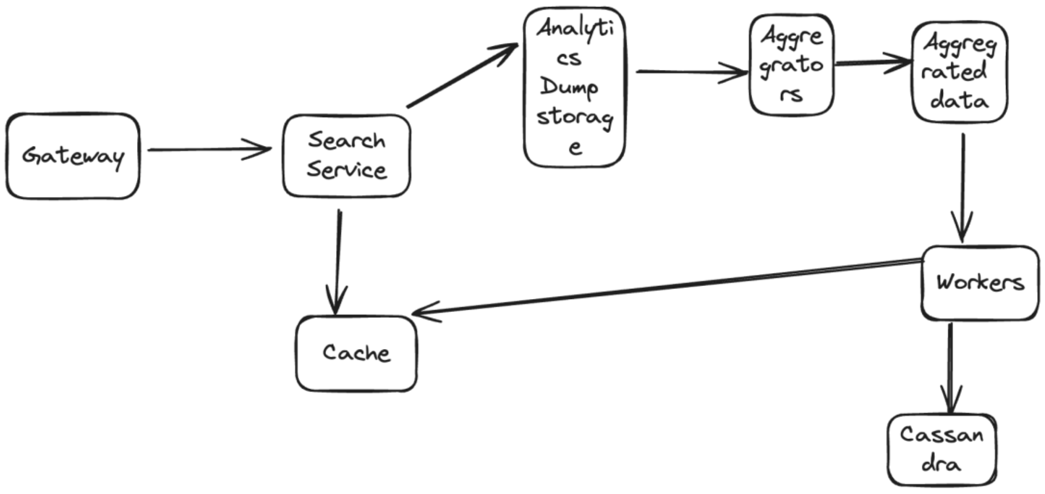

root
b
[be: 15, bee: 20, beer: 10, best: 35]
be: 15
[be: 15, bee: 20, beer: 10, best: 35]
bee: 20
[bee: 20, beer, 10]
bes
[best: 35]
beer: 10
best: 35

Key	Value
b	[be: 15, bee: 20, beer: 10, best: 35]
be	[be: 15, bee: 20, beer: 10, best: 35]
bee	[bee: 20, beer: 10]
bes	[best: 35]
beer	[beer: 10]
best	[best: 35]

Figure 13-10

- Can we also use cache ?
  To speed up the read request, we can use distributed key value cache like Redis with the required TTL. This can be taken care by the workers above. If we don't want to store all the data, put the logic in workers for the same.
- ~~Trie operation -> How does the update of Trie happen? Because we are using Cassandra which support high write throughput, we can directly update the DB.~~

Final Architecture



Sharding strategy

Here the hash based sharding will not work. Problem being there are more words in c and b than z and b and c can both on the same machine.
We must use range base sharding. Each server can have some group of characters which will have low + high density. We can maximum create 26 shards in this case.
If you are still facing the hotspoting issue, we might need to do the sharding based on more level of character set. E.g ca will go in this shard and cx will go in another.