Features:
This is the first part of any system design interview, coming up with the features which the system should support. As an interviewee, you should try to list down all the features you can think of which our system should support. Try to spend around 2 minutes for this section in the interview. You can use the notes section alongside to remember what you wrote.
Q: How many typeahead suggestions are to be provided?
A: Let's assume 5 for this case.
Q: Do we need to account for spelling mistakes ?
A: Example : Should typing *mik* give michael as a suggestion because michael is really popular as a query?
Lets assume we need not account for spelling mistakes, and assume that the suggestions will have the typed phrase as the strict prefix.
Q: What is the criteria for choosing the 5 suggestions ?
A: As the question suggests, all suggestions should have the typed phrase/query as the strict prefix. Now amongst those, the most relevant would be the most popular 5. Here, popularity of a query can be determined by the frequency of the query being searched in the past.
Q: Does the system need to be realtime (For example, recent popular events like "Germany wins the FIFA worldcup" starts showing up in results within minutes).
A: Let's assume that it needs to be realtime.
Q: Do we need to support personalization with the suggestions? (My interests / queries affect the search suggestions shown to me).
A: Let's assume that we don't need to support personalization
Estimations:
This is usually the second part of a design interview, coming up with the estimated numbers of how scalable our system should be. Important parameters to remember for this section is the number of queries per second and the data which the system will be required to handle.
Try to spend around 5 minutes for this section in the interview.

There are essentialy 2 parts to this system :

- Clients can query my system for top 5 suggestions given a query prefix.
- Every search query done should feed into the system for an update.

Lets estimate the volume of each.
Q: How many search queries are done per day?
A: Assuming the scale of Google, we can expect around 2-4 Billion queries per day.
Q: How many queries per second should the system handle?
A: We can use the estimation from the last question here.
Total Number of queries : 4 Billion
Average length of query : 5 words = 25 letters (Since avg length of english word is 5 letters).
Assuming, every single keystroke results in a typeahead query, we are looking at an upper bound of 4 x 25 = 100 Billion queries per day.
Q: How much data would we need to store?
A: Lets first look at the amount of new data we generate every day. 15% of the search queries are new for Google (~500 Million new queries). Assuming 25 letters on average per query, we will 12.5G new data per day.

Assuming, we have accumulated queries over the last 10 years, the size would be 12.5 * 365 * 10 G which is approximately 50TB.
Design Goals:
Latency - Is this problem very latency sensitive (Or in other words, Are requests with high latency and a failing request, equally bad?). For example, search typeahead suggestions are useless if they take more than a second.
Consistency - Does this problem require tight consistency? Or is it okay if things are eventually consistent?
Availability - Does this problem require 100% availability?
There could be more goals depending on the problem. It's possible that all parameters might be important, and some of them might conflict. In that case, you'd need to prioritize one over the other.
Q: Is Latency a very important metric for us?
A: A big Yes. Search typeahead almost competes with typing speed and hence needs to have a really low latency.
Q: How important is Consistency for us?
A: Not really important. If 2 people see different top 5 suggestions which are on the same scale of popularity, its not the end of the world. I, as a product owner, am happy as long as the results become eventually consistent.
Q: How important is Availability for us?
A: Very important. If search typeahead is not available, the site would still keep working. However, it will lead to a much degraded experience.
Skeleton of the design:
The next step in most cases is to come up with the barebone design of your system, both in terms of API and the overall workflow of a read and write request. Workflow of read/write request here refers to specifying the important components and how they interact. Try to spend around 5 minutes for this section in the interview.
Important : Try to gather feedback from the interviewer here to indicate if you are headed in the right direction.

As discussed before, there are essentially 2 parts to this system :

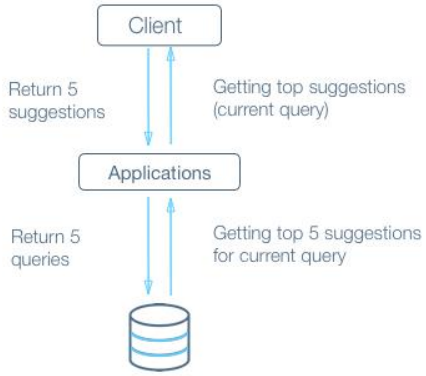
- Given a query, give me 5 most frequent search terms with the query as strict prefix
- Given a search term, update the frequencies.

Q: What would the API look like for the client?
A:
Read: List(string) getTopSuggestions(string currentQuery)
Write: void updateSuggestions(string searchTerm)
Q: What is a good data structure to store my search queries so that I can quickly retrieve the top 5 most popular queries?
A: For this question, we need to figure out top queries with another string as strict prefix. If you have dealt with enough string questions, you would realize a prefix tree (or trie) would be a perfect fit here.

Devil however lies in the details. We will dig deeper into the nitty gritty of this in the next section.
Q: How would a typical read query look like?
A: Components:

- Client (Mobile app / Browser, etc) which calls getTopSuggestions(currentQuery)
- Application server which interprets the API call and queries the database for the corresponding top 5 queries.
- Database server which looks up the top queries in the trie.

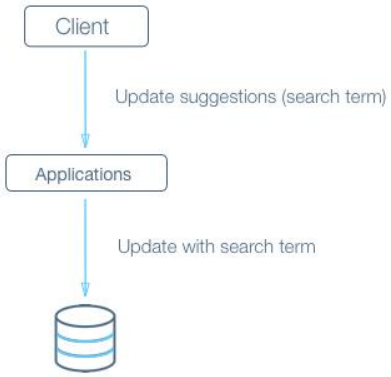
HIGH LEVEL DESIGN (READ)



Q: How would a typical write query look like?
A: Components:

- Client (Mobile app / Browser, etc) which calls updateSuggestions(searchTerm)
- Application server which interprets the API call and forwards the searchTerm to database for update.
- Database server which updates its trie using the searchTerm

HIGH LEVEL DESIGN



Deep Dive:
Lets dig deeper into every component one by one. Discussion for this section will take majority of the interview time(20-30 minutes).

Lets dig deeper into every component one by one.

Application layer:

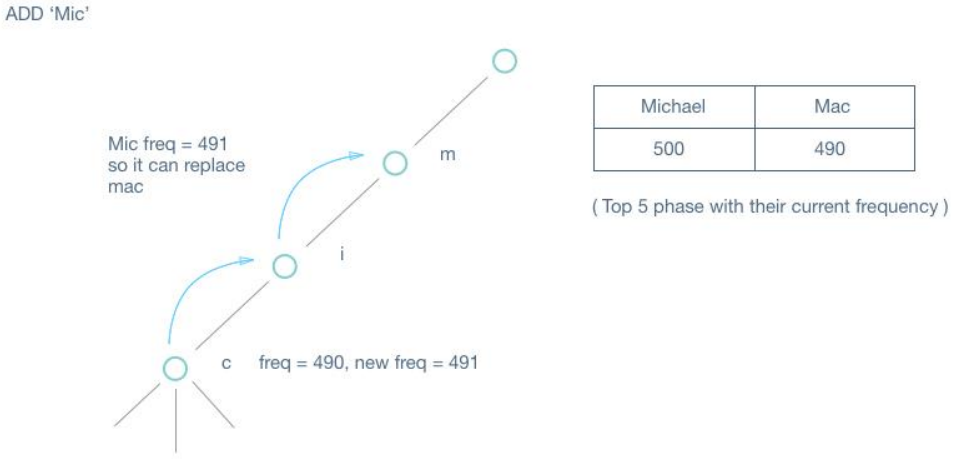
Think about all details/gotchas yourself before beginning.
Q: How would you take care of application layer fault tolerance?
Q: How do we handle the case where our application server dies?
A: The simplest thing that could be done here is to have multiple application server. They do not store any data (stateless) and all of them behave the exact same way when up. So, if one of them goes down, we still have other application servers who would keep the site running.
Q: How does our client know which application servers to talk to. How does it know which application servers have gone down and which ones are still working?
A: We introduce load balancers. Load balancers are a set of machines (an order of magnitude lower in number) which track the set of application servers which are active (not gone down). Client can send request to any of the load balancers who then forward the request to one of the working application servers randomly.
A: If we have only one application server machine, our whole service would become unavailable. Machines will fail and so will network. So, we need to plan for those events. Multiple application server machines along with load balancer is the way to go.

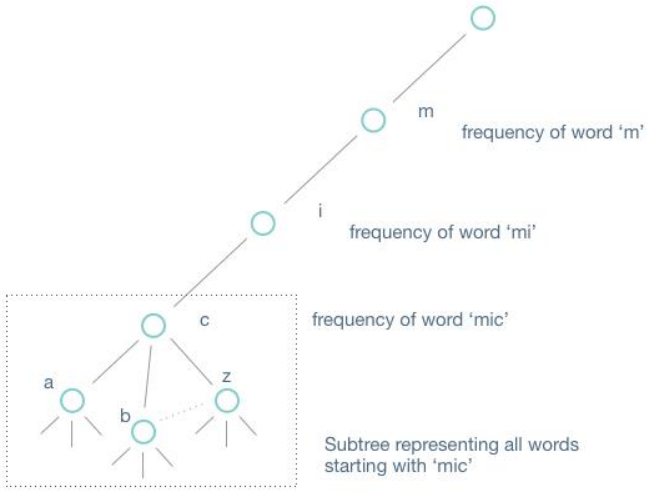
Database layer:

Let's first dig deeper into the trie we talked about earlier.

Q: How would a read query on the trie work?
A: The read query would require us to fetch the top 5 results per query. A traditional trie would store the frequency of the search term ending on the node n1 at n1. In such a trie, how do we get the 5 most frequent queries which have the search term as strict prefix. Obviously, all such

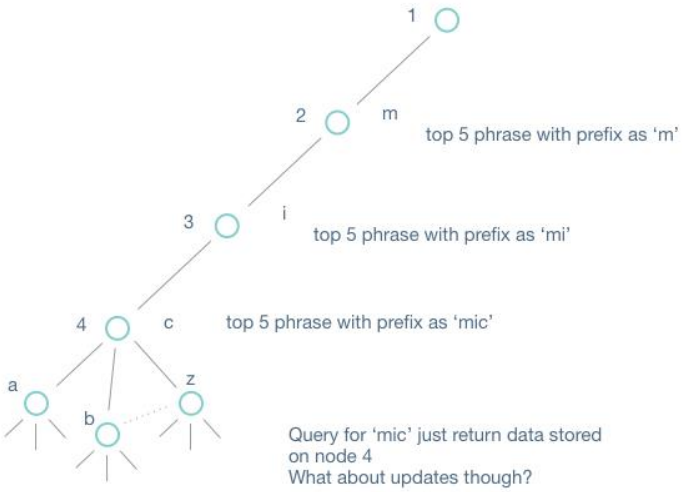
frequent queries would be the terms in the subtree under n1 (as shown in diagram).
So, a bruteforce way is to scan all the nodes in the subtree and find the 5 most frequent. Lets estimate the number of nodes we will have to scan this way.
Lets say, the user just typed one letter ‘a’. In such a case, we would end up scanning all queries which begin with ‘a’. As we discussed earlier, this would mean scanning terabytes of data which is clearly very time taking and inefficient. Also, the high latency does not align with our design goals.





Q: How can we modify the trie so that reads become super efficient?
Hint : Store more data on every node of the trie.
A: Storage is cheap. Lets say we were allowed to store more stuff on each node. How would we use the extra storage to reduce the latency of answering the query.
A good choice would be storing the top 5 queries for the prefix ending on node n1 at n1 itself. So, every node has the top 5 search terms from the subtree below it. The read operation becomes fairly simple now. Given a search prefix, we traverse down to the corresponding node and return

the top 5 queries stored in that node.



Q: How would a typical write work in this trie?
A: So, now whenever we get an actual search term, we will traverse down to the node corresponding to it and increase its frequency. But wait, we are not done yet. We store the top 5 queries in each node. Its possible that this particular search query jumped into the top 5 queries of a few other nodes. We need to update the top 5 queries of those nodes then. How do we do it then? Truthfully, we need to know the frequencies of the top 5 queries (of every node in the path from root to the node) to decide if this query becomes a part of the top 5.
There are 2 ways we could achieve this.

- Along with the top 5 on every node, we also store their frequency. Anytime, a node's frequency gets updated, we traverse back from the node to its parent till we reach the root. For every parent, we check if the current query is part of the top 5. If so, we replace the corresponding frequency with the updated frequency. If not, we check if the current query's frequency is high enough to be a part of the top 5. If so, we update the top 5 with frequency.
- On every node, we store the top pointer to the end node of the 5 most frequent queries (pointers instead of the text). The update process would involve comparing the current query's frequency with the 5th lowest node's frequency and update the node pointer with the current query pointer if the new frequency is greater.

Q: Can frequent writes affect read efficiency?
A: Yes, potentially. If we are updating the top 5 queries or the frequencies very frequently, we will need to take a lock on the node to make sure the reader thread does not get an inconsistent value. As such, writes start to compete with reads.
Q: What optimizations can we do to improve read efficiency?
Q: Can we use sampling?
A: Yes. If we assume Google's scale, most frequent queries would appear 100s of times in an hour. As such instead of using every query to update, we can sample 1 in 100 or 1 in 1000 query and update the trie using that.
Q: Offline update?

A: Again if we assume that most queries appearing in the search typeahead would appear 100s of times in an hour, we can have an offline hashmap which keeps maintaining a map from query to frequency. Its only when the frequency becomes a multiple of a threshold that we go and update the query in the trie with the new frequency. The hashmap being a separate datastore would not collide with the actual trie for reads.
A: As mentioned earlier, writes compete with read. Sampling writes and Offline updates can be used to improve read efficieny.
Q: What if I use a separate trie for updates and copy it over to the active one periodically?
A: Not really, there are 2 major problems with this approach.

- You are not realtime anymore. Lets say you copy over the trie every hour. Its possible a search term became very popular and it wasn't reflected for an hour because it was present in the offline trie and did not appear till it was copied to the original trie
- The trie is humungous. Copying over the trie can't be an atomic operation. As such, how would you make sure that reads are still consistent while still processing incoming writes?

Q: Would all data fit on a single machine?
A: Refer to estimations section. We would need to store more than 50TB of data.
Ideally, we would want most of it in memory to help with the latency. Thats a lot to ask from a single machine. We will go with a "No" here.
Q: Alright, how do we shard the data then?
Q: Would we only shard on the first level?
A: The number of shards could very well be more than the number of branches on first level(26). We will need to be more intelligent than just sharding on first level.
Q: What is the downside of assigning one branch to a different shard?
A: Load imbalance. Storage imbalance. Some letters are more frequent than the others. For example, letters starting with 'a' are more likely than letters starting with 'x'. As such, we can run into cases of certain shards running hot on load. Also, certain shards will have to store more data because there are more queries starting with a certain letter. Another fact in favor of sharding a little more intelligently.
A: Lets say we were sharding till the second or third level and we optimize for load here. Lets also say that we have the data around the expected load for every prefix.
We keep traversing the 2 letter prefixes in order ('a', 'aa', 'ab', 'ac',...) and break when the total load exceeds an threshold load and assign that range to a shard.
We will need to have a master which has this mapping with it, so that it can route a prefix query to the correct shard.
Q: How would we handle a DB machine going down?
A: As we discussed earlier, availability is more important to us than consistency. If thats the case, we can maintain multiple replica of each shard and an update goes to all replicas. The read can go to multiple replicas (not necessarily all) and uses the first response it gets. If a replica goes down, reads and writes continue to work fine as there are other replicas to serve the queries.
The issue occurs when this replica comes back up. There are 2 options here :

- If the frequency of the replica going down is lower or we have much higher number of replicas, the replica which comes back up can read the whole data from one of the older working replica while keeping the new incoming writes in a queue.

- There is a queue with every server which contains the changelog or the exact write query being sent to them. The replica can request any of the other replicas in its shard for all changelog since a particular timestamp and use that to update its trie.