System Design - Case study 2 (Typeahead)

Agenda

System Design - Getting Started

Typeaheads
- Design Problem
- Minimum Viable Product
- Estimate Scale
- Design Goals
- APIs

Building a Trie
- getSuggestions API
- updateFrequency API
- Sampling Approach
- HashMap Approach

How to split or shard the Trie?
- Disproportionate Load Problem
- HashMap Approach

Recency Factor

System Design - Getting Started

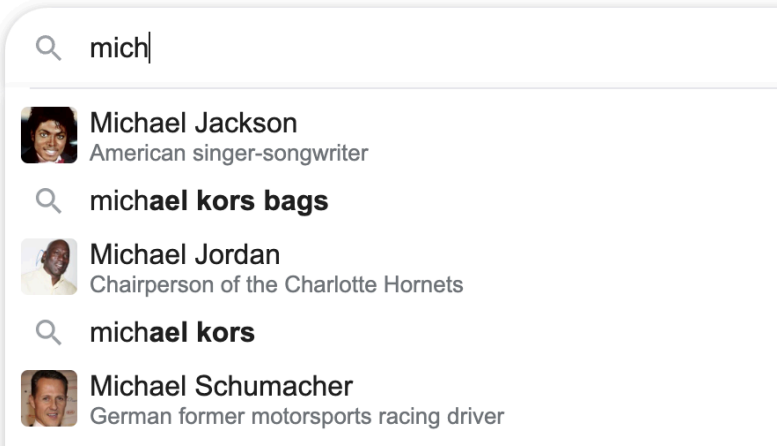
Before you jump into design you should know what you are designing for. The design solution also depends on the scale of implementation. Before moving to the design, ponder over the following points in mind:

- Figure out the MVP (Minimum Viable Product)
- Estimate Scale
    - Storage requirements (Is sharding needed?)
    - Read-heavy or write-heavy system
        - Write operations block read requests because they acquire a lock on impacted rows.
        - If you are building a write-heavy system, then the performance of reads goes down. So, if you are building both a read and write heavy system, you have to figure out how you absorb some of the reads or writes somewhere else.

        - Query Per Second (QPS)
            - If your system will address 1 million queries/second and a single machine handles 1000 queries/second, you have to provision for 1000 active machines.
- Design Goal
    - Highly Consistent or Highly Available System
    - Latency requirements
    - Can you afford data loss?
- How is the external world going to use it? (APIs)
    - The choice of sharding key may depend on the API parameters

Typeaheads

Typeaheads refers to the suggestions that come up automatically as we search for something. You may have observed this while searching on Google, Bing, Amazon Shopping App, etc.



Design Problem

- How to build a Search Typeahead system?
- Scale: Google

Minimum Viable Product

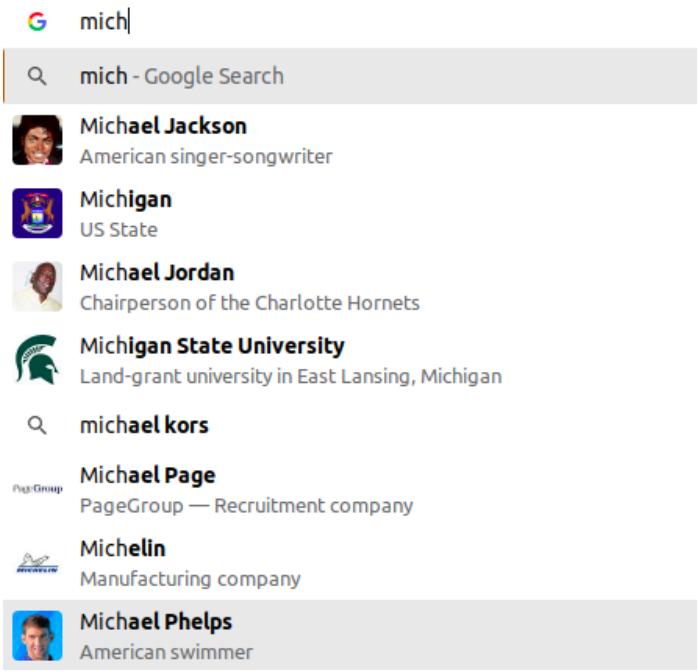
Consider Anshuman as the CEO of Google and he comes to Swaroop asking for building a typeahead system. Questions from the **Engineering Architect** (Swaroop):

- Maximum number of suggestions required?
    - Let's say five.
- Which suggestions? How to rank suggestions?
    - Choose the most popular ones. Next question-> Definition of Popularity.

    - Popularity of a search phrase is essentially how frequently do people search for that search phrase. It's combination of frequency of search, and recency. For now, assume popularity of a search term is decided by the number of times the search phrase was searched.
    - Strict prefix
- Personalisation may be required. But in MVP, it can be ignored. (In a real interview, check with the interviewer)
- Spelling mistakes not entertained.
- Keep some minimum number of characters post which suggestions will be shown.
    - Let's say 3.
- Support for special characters not required at this stage

Note:

- MVP refers to the functional requirements. Requirements such as latency, etc. are non-functional requirements that will be discussed in the Design Goal section.
- The algorithm to rank suggestions should also consider **recency** as a factor.
    - For example, Roger Binny has the highest search frequency: 1 million searches over the last 5 years. On a daily basis, it receives 1000 searches.
    - But, yesterday Roger Federer won Wimbledon and he has received 10000 queries since then. So, the algorithm should ideally rank Roger Federer higher.
    - However, for now let's move forward with frequency only.



Estimate Scale

Assumptions:

- **Search terms** or **Search queries** refers to the final query generated after pressing Enter or the search button.
- Google receives 10 billion search queries in a day.
- The above figure translates to 60 billion typeahead queries in a day if we assume each search query triggers six typeahead queries on average.

Need of Sharding?

Next task is to decide whether sharding is needed or not. For this we have to get an estimate of how much data we need to store to make the system work.

First, let's decide what we need to store.

- We can store the search terms and the frequency of these search terms.

Assumptions:

- 10% of the queries received by Google every day contain new search terms.
    - This translates to 1 billion new search terms every day.
    - Means 365 billion new search terms every year.
- Next, assuming the system is working past 10 years:
    - Total search terms collected so far: 10 * 365 Billion
- Assuming one search term to be of 32 characters (on average), it will be of 32 bytes.
- Let's say the frequency is stored in 8 bytes. Hence, total size of a row = 40 bytes.

Total data storage size (in 10 years): 365 * 10 * 40 billion bytes = 146 TB (Sharding is needed).

Read or Write heavy system

- 1 write per 6 reads.
    - This is because we have assumed 10 billion search queries every day which means there will be 10 billion writes per day.
    - Again each search query triggers 6 typeahead queries => 6 read requests.
- Both a read and write-heavy system.

Note: Read-heavy systems have an order of magnitude higher than writes, so that the writes don't matter at all.

Design Goals

- Availability is more important than consistency.
- Latency of getting suggestions should be super low - you are competing with typing speed.

APIs

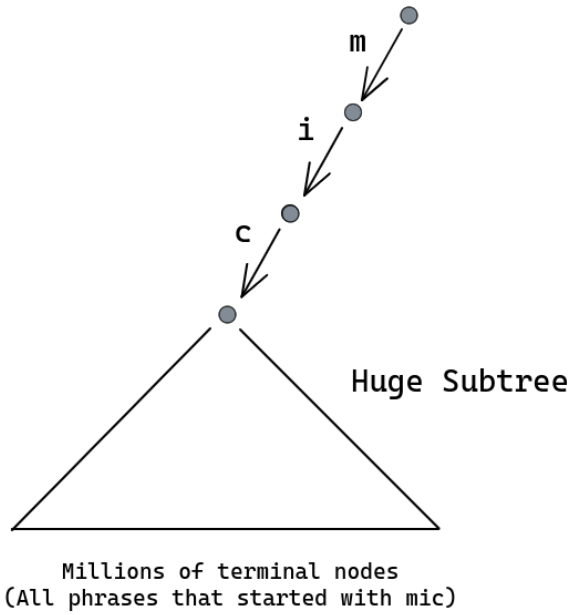
- getSuggestion(prefix_term, limit = 5)
- updateFrequency(search_term)
    - Asynchronous Job performed via an internal call
    - The Google service which provides search results to a user's query makes an internal call to Google's Typeahead service to update the frequency of the search term.

Trie Approach

- Construct a trie where each node is an English alphabet (or alphanumeric if digits are also considered)
- Each node has utmost 26 children i.e. 26 letters of the English alphabet system
- Each terminal node represents a search query (alphabets along **root** -> **terminal node**).

getSuggestions API

Consider **"mic"** as the search query against which you have to show typeahead suggestions. Consider the diagram below: The subtree can be huge, and as such if you go through the entire subtree to find top 5 suggestions, it will take time and our design goal of low latency will be violated.



Solution

* Store the top five suggestions in each node

* The getSuggestions API becomes super fast

* Fetch top five suggestions in O(prefix_term)

updateFrequency API

With the trie design suggested above where we are storing top five suggestions in every node, how will updateFrequency API work?

- In case of **updateFrequency** API call, let **T** be the terminal node which represents the **search_term**.

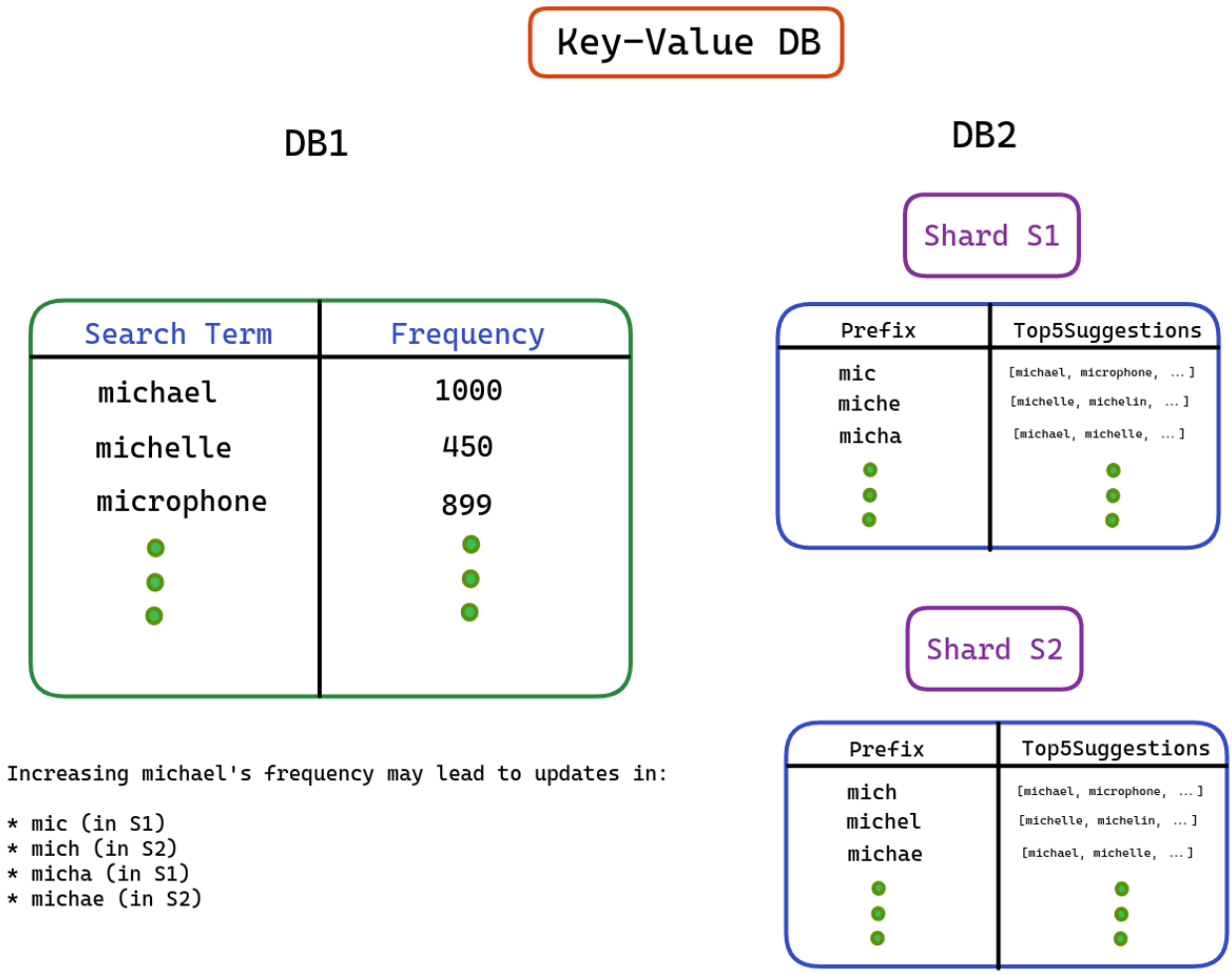
- Observe that only the nodes lying in the path from the **root to T** can have changes. So you only need to check the nodes which are **ancestors** of the terminal node **T** lying the path from **root to T**.

Summarizing, a single node stores:

- The frequency of the search query. [This will happen if the node is a terminal node]
- Top five suggestions with **string "root -> this node"** as prefix

HashMap Approach

- We can maintain two **HashMaps** or **Key-Value** store as follows:
    - **Frequency** HashMap stores the frequency of all search terms as a key-value store.
    - **Top5Suggestions** HashMap stores the top five suggestions corresponding to all possible prefixes of search terms.
- **Write:** Now, when **updateFrequency** API is called with a search term **S**, only the prefixes of the search term may require an update in the **Top5Suggestions** key-value store.
- **Write:** These updates on the **Top5Suggestions** key-value store need not happen immediately.
    - These updates can happen asynchronously to the shards storing the prefixes of the search term.
    - You can also maintain a queue of such updates and schedule it to happen accordingly.
- **Read:** In this system, if the search query is "**mich"**, through consistent hashing, I can quickly find the shard which stores the top five suggestions corresponding to "mich" key and return the same.
    - Consistent Hashing does not guarantee that "**mic**" and "**mich**" will end up on the same machine (or shard).
- In a Key-Value DB, Sharding is taken care of by the database itself. The internal sharding key is the key itself.
    - In any generic key-value store, the sharding happens based on the key.
    - However, you can specify your own sharding key if you want to.



Optimize writes (Read and write heavy -> Read Heavy)

Reads and writes compete with each other. In the design above, a single frequency update is leading to multiple writes in the trie (on the ancestral nodes) / hashmap. If writes are very large in number, it will impact the performance of reads and eventually **getSuggestions** API will be impacted.

Can we reduce the number of writes?

- Notice that exact frequency of the search query is not that important. Only the relative popularity (frequencies) of the search queries matter.

Threshold Approach

How about you buffer the writes in a secondary store. But with buffer, you risk not updating the trending search queries. Something that just became popular. How do we address that? How about a threshold approach (detailed below):

- Maintain a separate HashMap of additional frequency of search terms. This is to say, updateFrequency does not go directly to the trie or the main hashmap. But this secondary storage.
- Define a threshold and when this threshold is crossed for any search term, update the terminal trie node representation of the search term with **frequency = frequency + threshold**.
    - Why? You don't really care about additional frequency of 1 or 2. That is the long tail of new search terms. The search terms become interesting when their frequency is reasonably high. This is your way of filtering writes to only happen for popular search term.
    - If you set the threshold to 50 for example, you are indicating that something that doesn't even get searched 50 times in a day is not as interesting to me.
- As soon as one search item frequency is updated in the trie, it goes back to zero in the HashMap.
- Concerned with the size of the HashMap?
    - Let's estimate how big can this HashMap grow.
        - We have 10 billion search in a day. Each search term of 40 bytes.
        - Worst case, that amounts to 400GB. In reality, this would be much lower as new key gets created only for a new search term. A single machine can store this.
    - If you are concerned about memory even then, flush the HashMap at the end of the day
- So basically we are creating a write buffer to reduce the write traffic and since you do not want to lose on the recency, a threshold for popularity is also maintained.

Sampling Approach

Think of exit polls. When you have to figure out trends, you can sample a set of people and figure out trends of popular parties/politicians based on the results in the sample. Very similarily, even here you don't care about the exact frequency, but the trend of who the most popular search terms are. Can we hence sample?

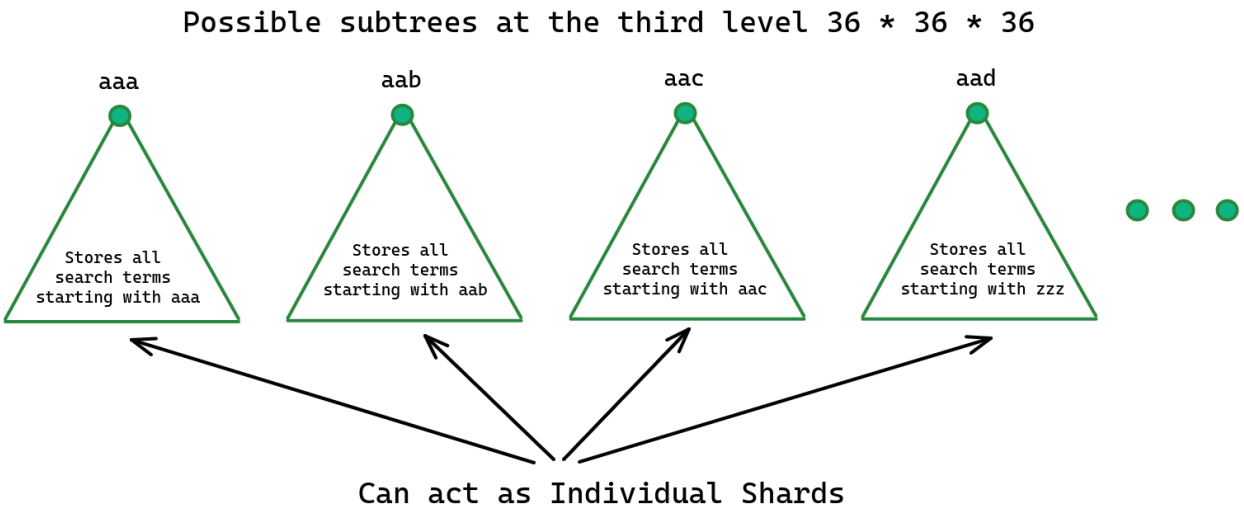
- Let's not update the trie/hashmap on every occurrence of a search term. We can assume that every 100th occurrence of a search term is recorded.
- This approach works better with high frequency counts as in our case. Since you are only interested in the pattern of frequency of search items and not in the exact numbers, Sampling can be a good choice.

Sharding

Sharding the trie:
Sharding key: Trie prefix

- The splitting or sharding should be on the basis of prefixes of possible search terms.

- Let's say someone has typed 3 characters. What are the possible subtrees at the third level?
    - The third level consists of 26 * 26 * 26 subtrees or branches. These branches contain prefix terms "aaa", "aab", "aac", and so on.
    - If we consider numbers as well, there will be 36 * 36 * 36 branches equivalent to around 50k subtrees.
    - Hence, the possible number of shards required will be around 50000.



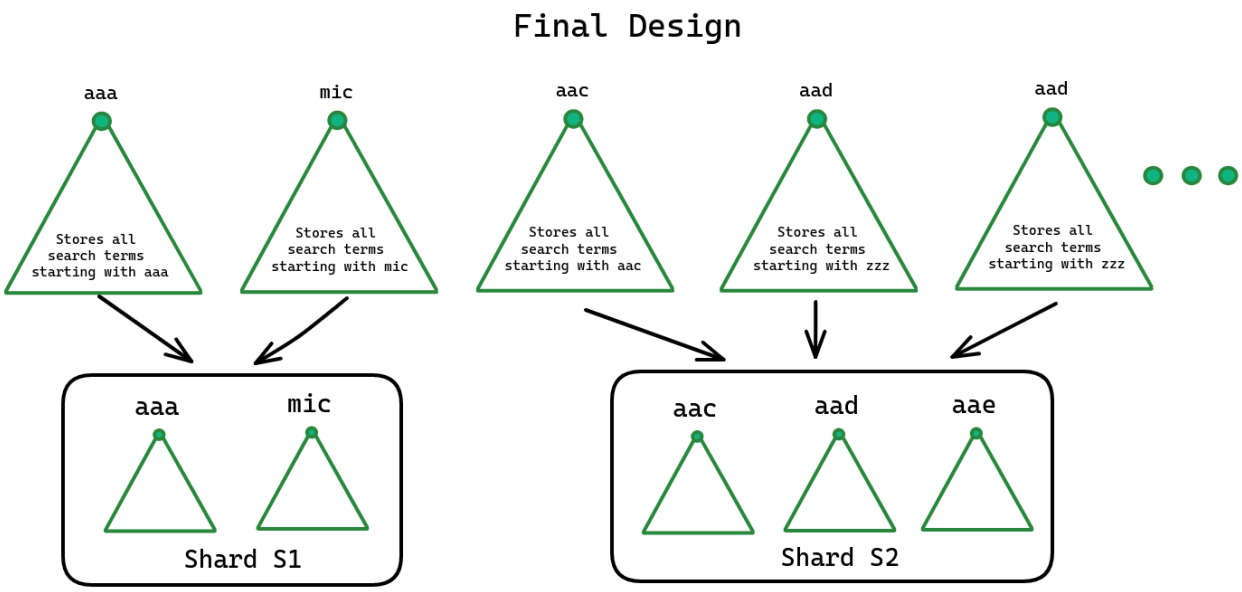
Disproportionate Load Problem

- The problem with the design above is that some shards or branches have high traffic while others are idle.
    - For example, the shard representing **"mic"** or **"the"** will have high traffic. However, the **"aaa"** shard will have low traffic.
- To solve this issue, we can group some prefixes together to balance load across shards.
    - For example, we can direct search terms starting with "**aaa"** and "**mic"** to the same shard. In this way we can better utilize the shard assigned to **"aaa"** somewhere else.
- So, we are sharding on the first three characters combined.

Example:

- Let's say the search query is "**mich"**, you find the right shard based on the hash generated for the first three characters "**mic"** and direct the request there.

Consistent Hashing code will map to the correct shard by using the first three characters of the search term.



Sharding the Hashmap DB

- Hashmap DB is easier to shard. It's just a collection of key and value.
- You can choose any existing key value DB and it automatically takes care of sharding
    - Consistent Hashing with sharding key as the key itself.

Recency Factor

How to take recency of search queries into consideration? How to reduce the weightage of search queries performed earlier as we move ahead?
For example, "Shania Twain" might have been a very popular search term 5 years back. But if no one searches for it today, then it's unfair to keep surfacing it as the most popular search term in suggestions (Less likelihood of people selecting that suggestion).

- One idea is to decrease a fixed number (absolute decrement) from each search term every passing day.
    - This idea is unfair to search terms with lower frequency. If you decide to decrease 10 from each search term:
        - Search term with frequency two is already on the lower side and further decreasing will remove it from the suggestions system.
        - Search terms with higher frequency such as 1000 will have relatively no effect at all.
- To achieve this, we can apply the concept **Time Decay.**
    - Think in terms of percentage.
    - You can decay the frequency of search terms by a constant factor which is called the **Time Decay Factor**.
    - The more quickly you want to decay the frequencies, the higher the **TDF**.

    - Every day, **Freq = Freq/TDF** and when **updateFrequency** is called, **Freq++**.
        - New frequency has a weight of one.
        - Frequency from yesterday has a weight of half.
        - Frequency from the day before yesterday has a weight of one-fourth and so on.
    - According to this approach, if a search term becomes less popular, it eventually gets kicked out of the system.
- Using the concept of Time Decay, every frequency is getting decreased by the same percentage.

Summarizing the class:

- Tries cannot be saved in databases, unless you implement one of your own. Hence, you can look at the Hashmap approach as a more practical one (All key value stores work).
- Reads and writes compete, hence you need to think of caching reads or buffering writes. Since, consistency is not required in this system, hence buffering writes is a real option.
- Sampling cares only about the trend and not about the absolute count. A random sample exhibits the same trend.
    - It is like if a whole city is fighting, you pick 1% of the city randomly, even if it would be fighting as well.
    - Best example is the Election Exit Poll. Based on the response of a random set of population, we determine the overall trend.
    - Hence, the same trend is exhibited by a random sample of search queries. Using this approach, the number of writes gets reduced.
    - Basically, if you choose to sample 1% of the queries, the number of writes got reduced by 100x.
- Time Decay factor reduces the weightage of search queries performed in the past in an exponential fashion.