Interview Daemon

Follow 9 Followers About

System Design: AutoComplete System

(Text Typeahead System)

Interview Daemon Jan 8 · 6 min read

Source: interviewdaemon.com (Link)

Autocomplete is a technical term used for the search suggestions you see when searching. This feature increases text input speed. In fact, it is intended to speed up your search interaction by trying to predict what you are searching for when typing.

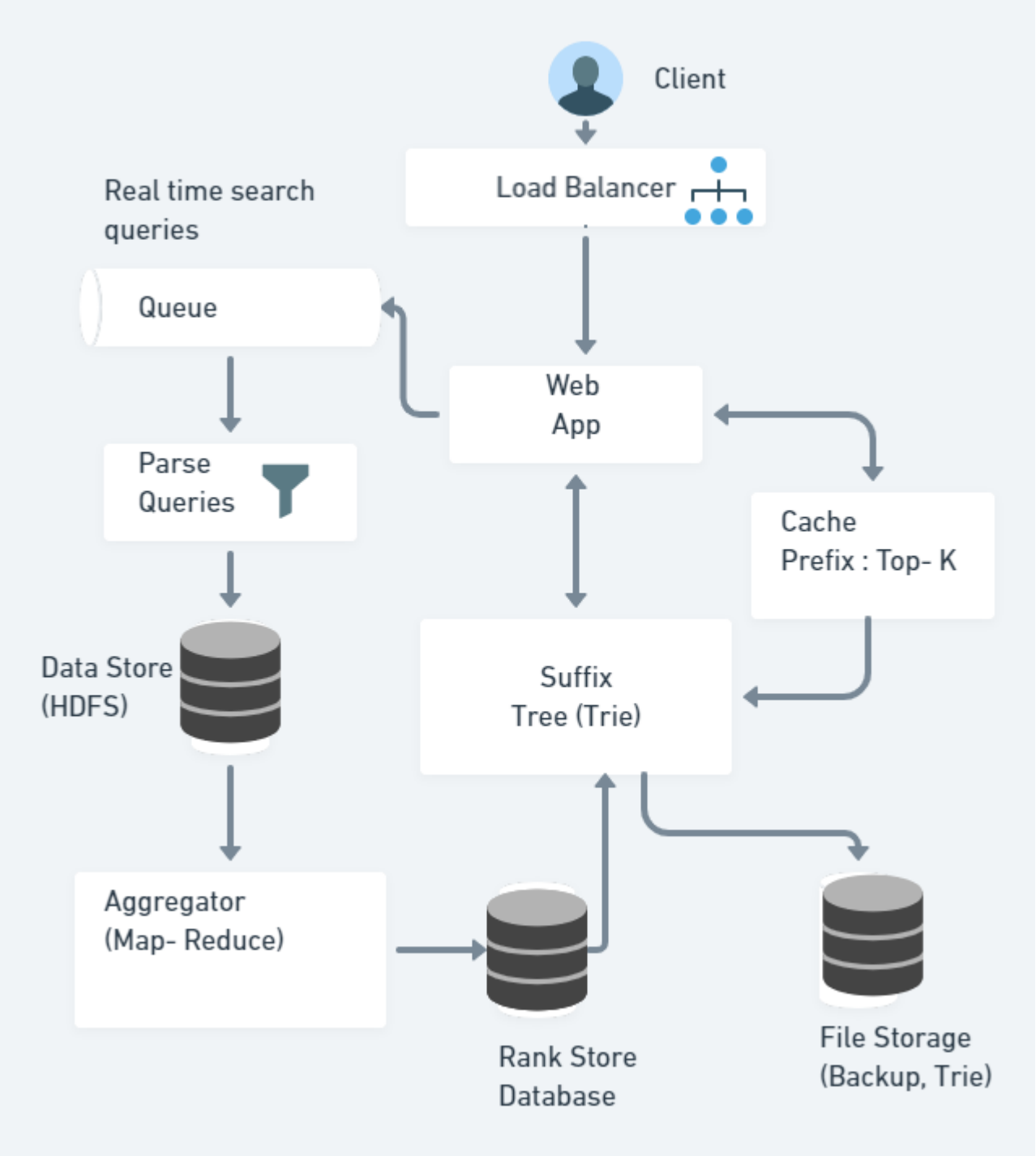
Introduction

Design a autocomplete system given the word suggests five typeahead suggestions based on the prefix and top-ranking search words.

Requirements

- We return **five typeahead suggestions.**
- Suggestions ordered by **some ranking score**
- The top-ranking **score is real-time** based on the most recent searched words.
- The system should be **High Performance**

At a high-level, we need to support multiple micro-services in the current system.



Source: interviewdaemon.com

API Design

1. getTopSuggestions

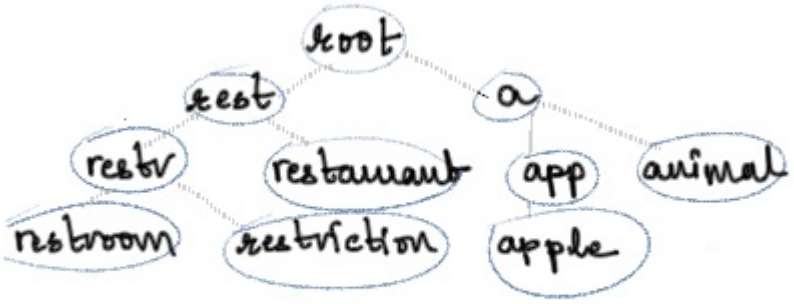
Returns the top suggestions based on query.

```
String getTopSuggestions(string currentQuery)
```

Search Algorithms

1. Trie DataStructure

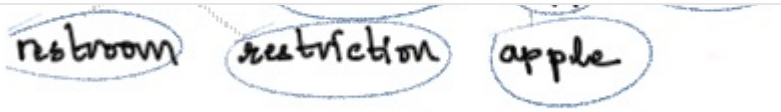
Trie is a data retrieval data structure. A trie is a tree-like data structure used to store phrases where each node stores a character of the phrase in a sequential manner. Trie can search the key in O(M) time. The storage can be an **in-memory cache** (Redis or Memcached), a database, or even a file.



- Implement in Java (Link)

2. Suffix-Tree Algorithm

A suffix-tree is a compressed **trie** of all the suffixes of a given string. It is such a **trie** that can have a long path without branches. The better approach is reducing these long paths into one way, and the advantage of this is to minimize the trie's size significantly.



There is an algorithm designed by **Ukkonen** for making a suffix tree for **s** in linear time in terms of the length of **s**.

Suffix trees can solve many complicated problems because it contains so many data about the string itself. For example, in order to know how many times a pattern **P** occurs in **s**, it is sufficient to find **P** in **T** and return the size of a subtree corresponding to its node. Another well-known application is finding the number of distinct substrings of **s**, and it can be solved easily with a suffix tree.

3. Minimal Deterministic Finite Automata

A prefix tree is an instance of a class of more general data structures called acyclic deterministic finite automata (DFA). There are algorithms for transforming a DFA into an equivalent DFA with fewer nodes. Minimizing a prefix tree DFA reduces the size of the data structure.

The **Myhill-Nerode theorem** gives us a theoretical representation of the minimal DFA in terms of string equivalence classes. A minimal DFA fits in the memory even when the vocabulary is large.

- Minimization of DFA (Link)

Component Design

1. Cache (In-Memory Store)

{‘re’: [‘restriction’:6, ‘restaurant’:5], ‘a’: [‘apple’:11, ‘animal’:5], ‘ap’: [‘apple’:11, ‘app’:2]}

Normally trie is stored in Database / (Key-Valye) Object Store. which ensures its persistency and good to scale for large tries. We can store some of sorted ranked results

All the search queries besides checking into the cache can also be pushed into a queue, which can later be consumed by a real-time log/search parser processing these queries and dumping them into the Highly Distributed File Storage System.

3. Aggregator (Offline Processing — Map Reduce — Rank Score Source)

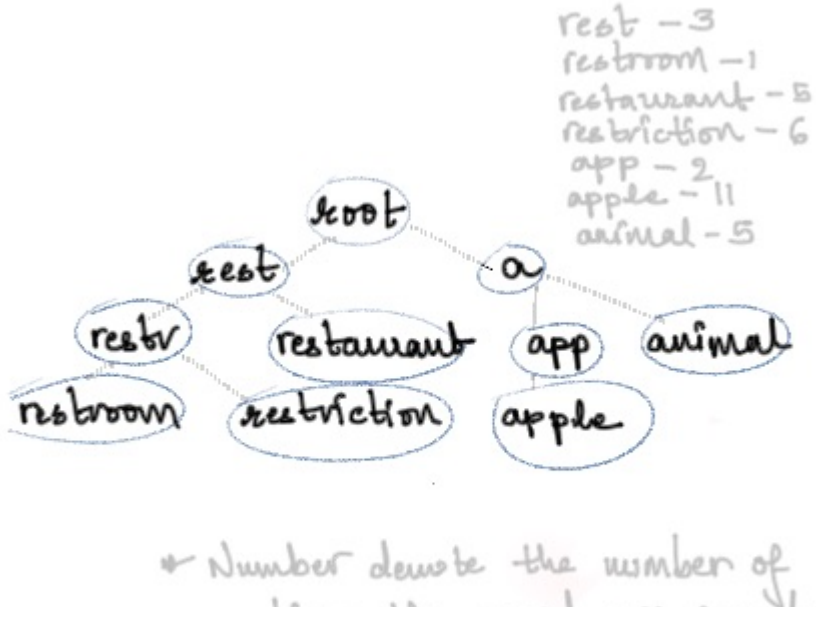
If we wish to recommend users with the most frequently and newest searched terms, we'll need to have an aggregator to calculate words' score with real-time data so that trie could be kept updated.
First of all, we want to sample user search, together with a timestamp, send to map-reduce job.

In order to do this, first of all, we want to sample user search, together with timestamp, send to map reduce job.

MapReduce job then calculates frequency rank score of all sampled search terms in a specific amount of time and send the result to an aggregator (no need to include those low-frequency results cause we know they won't be ranked top K in the trie).

We can store the rank store table in the KeyValue store of Relational database so that it can later be referenced to build a trie where each node stores the latest updated ranking.

4. Trie



- **Naive Approach**: 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.
- **Optimized Approach**: 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.

Update/Write to Trie

Write or update we also have to update the top 5 queries at each node. For example if we are adding a new word `applied`. We also have to update top 5 results of `app`, `appl` and `applied`. There are 2 ways we can achive this

1. 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. for exampe on adding applied we will traverse to root via `applied` -> `appl` -> `app` -> `root` and update all the top 5 results of each node.
2. 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. Here we go from root to applied. We compare lowest 5th frequent element with applied count

Frequent Reads / Impacting Batch Update

- If we are updating the top 5 queries or the frequencies very frequently. For example most frequent queries would appear 100s of times in an hour. So we may have to update our trie more often. We can batch all read and update the trie together.
- We can do offline processing as mentioned in step 1.

Cache and Load Balancing

We can introduce a cache for metadata servers to cache hot database rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system. (We can use Memcache / Redis)

How can we build a more intelligent cache? If we go with the eighty-twenty rule, i.e., 20% of daily read volume for search is generating 80% of the traffic, which means that certain search words are so popular that most people query them

References:

- DingDingWang — Medium
- Dzone
- GeeksForGeeks,