Back To Course Home

Grokking the System Design Interview

0% completed

Search Course

System Design Problems

- System Design Interviews: A step by step guide
- Designing a URL Shortening service like TinyURL
- Designing Pastebin
- Designing Instagram
- Designing Dropbox
- Designing Facebook Messenger
- Designing Twitter
- Designing Youtube or Netflix
- Designing Typeahead Suggestion
- Designing an API Rate Limiter
- Designing Twitter Search
- Designing a Web Crawler
- Designing Facebook's Newsfeed
- Designing Yelp or Nearby Friends
- Designing Uber backend
- Design Ticketmaster
- Additional Resources

Glossary of System Design Basics

- System Design Basics
- Key Characteristics of Distributed Systems
- Load Balancing
- Caching
- Data Partitioning
- Indexes
- Proxies
- Redundancy and Replication
- SQL vs. NoSQL
- CAP Theorem
- Consistent Hashing
- Long-Polling vs WebSockets vs Server-Sent Events

Appendix

- Contact Us
- Other courses

Course Certificate

Mark Course as Completed

Designing Typeahead Suggestion

Let's design a real-time suggestion service, which will recommend terms to users as they enter text for searching.

Similar Services: Auto-suggestions, Typeahead search
Difficulty: Medium

We'll cover the following

- 1. What is Typeahead Suggestion?
- 2. Requirements and Goals of the System
- 3. Basic System Design and Algorithm
- 4. Permanent Storage of the Trie
- 5. Scale Estimation
- 6. Data Partition
- 7. Cache
- 8. Replication and Load Balancer
- 9. Fault Tolerance
- 10. Typeahead Client
- 11. Personalization

1. What is Typeahead Suggestion?

Typeahead suggestions enable users to search for known and frequently searched terms. As the user types into the search box, it tries to predict the query based on the characters the user has entered and gives a list of suggestions to complete the query. Typeahead suggestions help the user to articulate their search queries better. It's not about speeding up the search process but rather about guiding the users and lending them a helping hand in constructing their search query.

2. Requirements and Goals of the System

Functional Requirements: As the user types in their query, our service should suggest top 10 terms starting with whatever the user has typed.

Non-function Requirements: The suggestions should appear in real-time. The user should be able to see the suggestions within 200ms.

3. Basic System Design and Algorithm

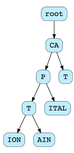
The problem we are trying to solve is that we have a lot of 'strings' that we need to store in such a way that users can search with any prefix. Our service will suggest the next terms matching the given prefix. For example, if our database contains the following terms: cap, cat, captain, or capital, and the user has typed in 'cap', our system should suggest 'cap', 'captain' and 'capital'.

As we have to serve a lot of queries with minimum latency, we need to come up with a scheme that can efficiently store our data such that it can be queried quickly. We can't depend upon some database for this; we need to store our index in memory in a highly efficient data structure.

One of the most appropriate data structures that can serve our purpose is the Trie (pronounced "try"). 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. For example, if we need to store 'cap, cat, caption, captain, capital' in the trie, it would look like:

Now if the user has typed 'cap', our service can traverse the trie to go to the node 'P' to find all the terms that start with this prefix (e.g., cap-tion, cap-ital etc).

We can merge nodes that have only one branch to save storage space. The above trie can be stored like this:



Should we have case insensitive trie? For simplicity and search use-case, let's assume our data is case insensitive.

How to find top suggestion? Now that we can find all the terms for a given prefix, how can we find the top 10 terms for the given prefix? One simple solution could be to store the count of searches that terminated at each node, e.g., if users have searched about 'CAPTAIN' 100 times and 'CAPTION' 500 times, we can store this number with the last character of the phrase. Now if the user types 'CAP' we know the top most searched word under the prefix 'CAP' is 'CAPTION'. So, to find the top suggestions for a given prefix, we can traverse the sub-tree under it.

Given a prefix, how much time will it take to traverse its sub-tree? Given the amount of data we need to index, we should expect a huge tree. Even traversing a sub-tree would take really long, e.g., the phrase 'system design interview questions' is 30 levels deep. Since we have very strict latency requirements we do need to improve the efficiency of our solution.

Can we store top suggestions with each node? This can surely speed up our searches but will require a lot of extra storage. We can store top 10 suggestions at each node that we can return to the user. We have to bear the big increase in our storage capacity to achieve the required efficiency.

We can optimize our storage by storing only references of the terminal nodes rather than storing the entire phrase. 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.

How would we build this 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 of their children to determine their top suggestions.

How to update the trie? Assuming five billion searches every day, which would give us approximately 60K queries per second. If we try to update our trie for every query it'll be extremely resource intensive and this can hamper our read requests, too. One solution to handle this could be to update our trie offline after a certain interval.

As the new queries come in we can log them and also track their frequencies. Either we can log every query or do sampling and log every 1000th query. For example, if we don't want to show a term which is searched for less than 1000 times, it's safe to log every 1000th searched term.

We can have a Map-Reduce (MR) set-up to process all the logging data periodically say every hour. These MR jobs will calculate 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 don't want our read queries to be blocked by update trie requests. We can have two options:

1. We can make a copy of the trie on each server to update it offline. Once done we can switch to start using it and discard the old one.
2. Another option is we can have a primary-secondary configuration for each trie server. We can update the secondary while the primary is serving traffic. Once the update is complete, we can make the secondary our new primary. We can later update our old primary, which can then start serving traffic, too.

How can we update the frequencies of typeahead suggestions? 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're keeping count of all the terms searched in the last 10 days, we'll need to subtract the counts from the time period no longer included and add the counts for the new time period 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's also known as the exponentially weighted moving average.

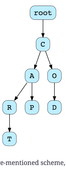
After inserting a new term in the trie, we'll go to the terminal node of the phrase and increase its frequency. Since we're storing the top 10 queries in each node, it is possible that this particular search term jumped into the top 10 queries of a few other nodes. So, we need to update the top 10 queries of those nodes then. We have to 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.

How can we remove a term from the trie? Let's say we have to remove a term from the trie because of some legal issue or hate or piracy etc. We can completely remove such terms from the trie when the regular update happens, meanwhile, we can add a filtering layer on each server which will remove any such term before sending them to users.

What could be different ranking criteria for suggestions? In addition to a simple count, for terms ranking, we have to consider other factors too, e.g., freshness, user location, language, demographics, personal history, etc.

4. Permanent Storage of the Trie

How to store trie in a file so that we can rebuild our trie easily - this will be needed when a machine restarts? We can take a snapshot of our trie periodically and store it in a file. This will enable us to rebuild a trie if the server goes down. To store, we can start with the root node and save the trie level-by-level. With each node, we can store what character it contains and how many children it has. Right after each node, we should put all of its children. Let's assume we have the following trie:



If we store this trie in a file with the above-mentioned scheme, we will have: "C2,A2,R1,T,P,O1,D". From this, we can easily rebuild our trie.

If you've noticed, we are not storing top suggestions and their counts with each node. It is hard to store this information; as our trie is being stored top-down, we don't have child nodes created before the parent, so there is no easy way to store their references. For this, we have to recalculate all the top terms with counts. This can be done while we are building the trie. Each node will calculate its top suggestions and pass it to its parent. Each parent node will merge results from all of its children to figure out its top suggestions.

5. Scale Estimation

If we are building a service that has the same scale as that of Google we can expect 5 billion searches every day, which would give us approximately 60K queries per second.

Since there will be a lot of duplicates in 5 billion queries, we can assume that only 20% of these will be unique. If we only want to index the top 50% of the search terms, we can get rid of a lot of less frequently searched queries. Let's assume we will have 100 million unique terms for which we want to build an index.

Storage Estimation: If on the average each query consists of 3 words and if the average length of a word is 5 characters, this will give us 15 characters of average query size. Assuming we need 2 bytes to store a character, we will need 30 bytes to store an average query. So total storage we will need:

$$100 \text{ million} * 30 \text{ bytes} \Rightarrow 3 \text{ GB}$$

We can expect some growth in this data every day, but we should also be removing some terms that are not searched anymore. If we assume we have 2% new queries every day and if we are maintaining our index for the last one year, total storage we should expect:

$$3\text{GB} + (0.02 * 3 \text{ GB} * 365 \text{ days}) \Rightarrow 25 \text{ GB}$$

6. Data Partition

Although our index can easily fit on one server, we can still partition it in order to meet our requirements of higher efficiency and lower latencies. How can we efficiently partition our data to distribute it onto multiple servers?

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.

We can see that the above problem will happen with every statically defined scheme. It is not possible to calculate if each of our partitions will fit on one server statically.

b. Partition based on the maximum capacity of the server: Let's say 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 to the next server to repeat this process. Let's say if our first trie server can store all terms from 'A' to 'AABC', which mean our next server will store from 'AABD' onwards. If our second server could store up to 'BXA', the next server will start from 'BXB', and so on. We can keep a hash table to quickly access this partitioning scheme:

Server 1, A-AABC
Server 2, AABD-BXA
Server 3, BXB-CDA

For querying, if the user has typed 'A' we have to query both servers 1 and 2 to find the top suggestions. When the user has typed 'AA', we still have to query server 1 and 2, but when the user has typed 'AAA' we only need to query server 1.

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 to do this on the server-side, we need to introduce another layer of servers between load balancers and trie severs (let's 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 have to ask all the servers and then aggregate the results.

7. Cache

#

We should realize that caching the top searched terms will be extremely helpful in our service. There will be a small percentage of queries that will be responsible for most of the traffic. We can have separate cache servers in front of the trie servers holding the most frequently searched terms and their typeahead suggestions. Application servers should check these cache servers before hitting the trie servers to see if they have the desired searched terms. This will save us time to traverse the trie.

We can also build a simple Machine Learning (ML) model that can try to predict the engagement on each suggestion based on simple counting, personalization, or trending data, and cache these terms beforehand.

8. Replication and Load Balancer

#

We should have replicas for our trie servers both for load balancing and also for fault tolerance. We also need a load balancer that keeps track of our data partitioning scheme and redirects traffic based on the prefixes.

9. Fault Tolerance

#

What will happen when a trie server goes down? As discussed above we can have a primary-secondary configuration; if the primary dies, the secondary can take over after failover. Any server that comes back up, can rebuild the trie based on the last snapshot.

10. Typeahead Client

#

We can perform the following optimizations on the client-side to improve user's experience:

1. The client should only try hitting the server if the user has not pressed any key for 50ms.
2. If the user is constantly typing, the client can cancel the in-progress requests.
3. Initially, the client can wait until the user enters a couple of characters.
4. Clients can pre-fetch some data from the server to save future requests.
5. Clients can store the recent history of suggestions locally. Recent history has a very high rate of being reused.
6. Establishing an early connection with the server turns out to be one of the most important factors. As soon as the user opens the search engine website, the client can open a connection with the server. So when a user types in the first character, the client doesn't waste time in establishing the connection.
7. The server can push some part of their cache to CDNs and Internet Service Providers (ISPs) for efficiency.

11. Personalization

#

Users will receive some typeahead suggestions based on their historical searches, location, language, etc. We can store the personal history of each user separately on the server and also cache them on the client. The server can add these personalized terms in the final set before sending it to the user. Personalized searches should always come before others.