Design Typeahead (Autocomplete) System

What is Typeahead?

Typeahead, or autocomplete, is a feature that allows an application to predict the rest of a word as a user is typing it. In many graphical user interfaces, the user can accept a suggestion by pressing the tab key or scroll through multiple suggestions by using the down arrow key.

Autocomplete is commonly used in search functions across a variety of platforms, including social media and search engines such as Facebook, Instagram, and Google. It is a convenient tool that can save time and improve the user experience by providing relevant suggestions as the user types.

Requirements

Functional Requirements

Our service should provide the top 10 terms that begin with the user's current query as they type. The suggestions should be sorted based on the frequency and recentness of the phrases or queries. This means that the most commonly used and recently used terms that match the user's input will be prioritized in the list of suggestions.

Non-Functional Requirements

- High availability: The system should be able to handle disruptions and continue to function without significant downtime.
- Real-time suggestions: The user should be able to see suggestions in near-instantaneous time, with a target of within 200 milliseconds.
- Scalability: The system should be able to handle a large volume of traffic without experiencing performance issues.

Estimation of Scale and Constraints

Traffic estimations

If we build a service on the same scale as Google, we can expect around 5 billion searches per day, or about 60,000 queries per second. However, we can estimate that only 20% of these searches will be unique, with many duplicates. If we only want to index the top 50% of search phrases, we can exclude less frequently searched queries.

For example, if we want to develop an index for 100 million different terms, we can estimate the storage requirements as follows:

- Average query size: 15 characters (assuming 3 words per query with an average length of 5 characters per word)
- Storage space per query: 30 bytes (assuming 2 bytes per character)
- Total storage space: 100 million * 30 bytes = 3 GB

We can expect some growth in the data over time, but we can also exclude phrases that are no longer searched. If we assume a 2% increase in new requests every day and retain our index for the previous year, the total storage would be approximately 25 GB: 3 GB + (0.02 * 3 GB * 365 days). This calculation takes into account both the initial storage requirements and the growth in data over time.

System APIs

1. **get_suggestions(prefix)**: This function takes in a prefix (i.e., the initial characters that have been typed by the user in a search query) and returns a list of suggestions based on that prefix.
2. **add*to*db(query)**: This function updates the database with a new, unique, and trending search query that has been searched above a certain threshold. The input for this function is the query that is being added to the database.

High Level Design

To meet the real-time requirement for our autocomplete system, we need to be able to quickly add new search queries to our database. This means that in addition to developing a system for providing suggestions to the user, we also need to incorporate popular search queries into our database. This will allow users to receive suggestions based on both current and popular searches, ensuring that the autocomplete feature is up-to-date and relevant.

As a result, we've divided our autocomplete into two sections:

1. **Making recommendations**: This component is responsible for taking in a search query or prefix and returning the seven most frequently searched keywords that match the input.
2. **Collecting information**: This component is responsible for collecting and organizing user input requests in real-time. While it may not be feasible to process large datasets in real-time, this approach can serve as a starting point. We will later explore a more realistic approach for handling larger data sets.

The autocomplete feature works by sending a query to the application servers every time the user types a character into the search field. The query get_suggestions() retrieves the user's top seven options from the database and displays them as suggestions.

Data Structure to Store Search Queries

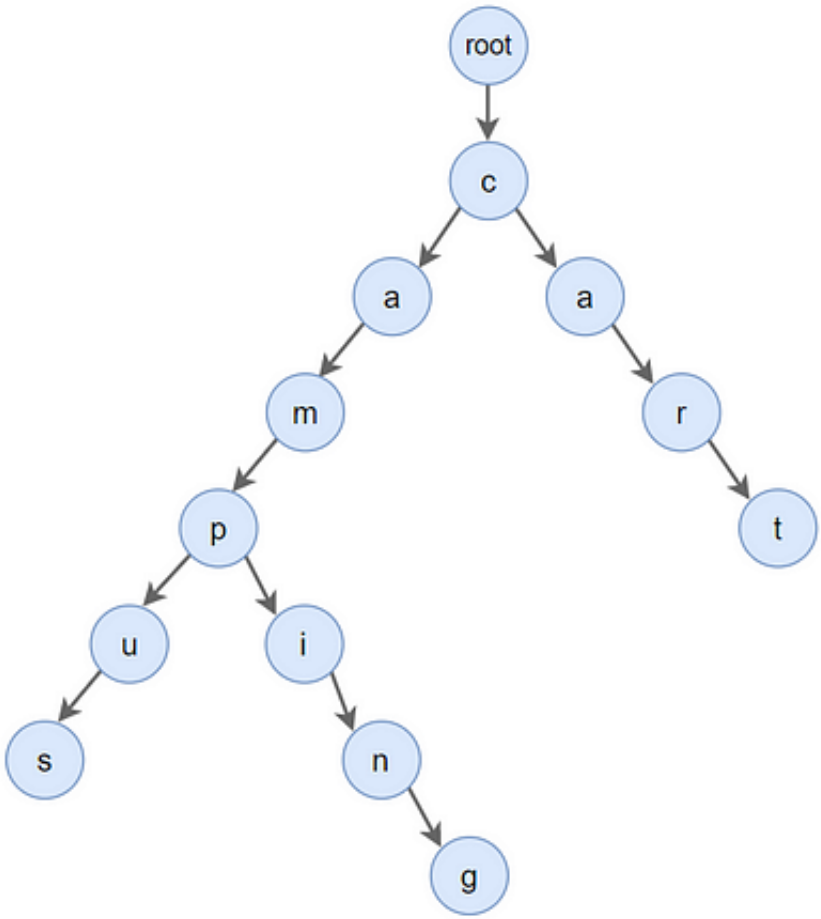
To efficiently handle a large number of queries with minimal latency, we need a system that can store our data in a way that allows for fast querying. We cannot use a traditional database for this purpose, so we need to use an in-memory data structure that is optimized for fast access.

One effective data structure for this purpose is a trie, also known as a prefix tree. A trie is a tree-like structure that stores phrases by representing each character in the phrase as a node in the tree. The root node represents an empty string, and each child node stores a character and has 26 children, one for each possible character. This allows the trie

to efficiently store and retrieve strings based on their prefixes.

In our case, the trie data structure is well-suited for storing and querying search phrases because it allows us to efficiently find and retrieve the most frequently searched terms that match a given prefix.

If we need to store ‘camping, campus, cart’ in the trie, for example, it will look like this:



If the user has typed ‘cam,’ our service can traverse the trie to the node ‘m’ to find all the terms that start with this prefix (e.g., cam-ping, cam-pus, etc.).

We need to design an efficient solution for indexing a large amount of data, as the tree we expect to create will be massive and traversing even a sub-tree may take a long time. For example, the search term "system design interview questions" has 30 levels. Our solution needs to have a high level of efficiency because we have strict latency constraints.

To improve search speed, we can save the top ten suggestions for each node and return them to the user. However, this will require a significant increase in storage capacity. Instead of saving the full phrases, we can save space by storing references to the terminal nodes and keeping track of the frequency of each reference. To find the suggested terms, we can use the parent reference from the terminal node.

To construct the trie efficiently, we can build it from the bottom up. Each parent node will first call all of its child nodes recursively to compute the top suggestions and counts. Then, the parent node will combine the top suggestions from all of its children to decide on its own top choices.

If we have a daily search volume of five billion, this translates to around 60,000 requests per second. Updating the trie on every query will use a lot of resources and slow down our read requests. One solution to this problem is to update the trie offline after a set period of time.

We can also track the frequency of new queries as they come in. We can either log every query or sample and log every 1,000th query. It is acceptable to only log every 1,000th searched term if we don't want to show terms that have been searched fewer than 1,000 times.

To regularly process all of the logging data, we can set up a Map-Reduce (MR) job. For example, we can run this MR job once an hour to calculate the frequency of all terms searched in the previous hour. Then, we can use this information to update our trie with the new data. To do this, we can update the trie's current snapshot with all of the new terms and their frequencies. Note: It is important to perform this update offline, as we do not want the process of updating the trie to obstruct our read queries.

There are two possibilities available to us:

1. We can make a copy of the trie on each server and update it. Once the update is complete, we can start using the new copy and discard the old one.
2. We can set up each trie server as a master-slave arrangement. While

the master is serving traffic, we can update the slave. Once the update is complete, we can make the slave the new master and update the old master. This way, both the new master and the updated old master will be able to serve traffic.

How can we keep the frequency of typeahead suggestions up to date?

We need to update the frequencies of our typeahead suggestions, which are stored with each node in the trie. Instead of recreating all search phrases from scratch, we can simply update the changes in frequency. For example, if we want to track all phrases searched in the last 10 days, we can adjust the counts by subtracting the counts from the time period that is no longer included and adding the counts for the new period. We can do this using the Exponential Moving Average (EMA) of each term, which gives more weight to the most recent data.

When we add a new term to the trie, we will go to the terminal node for that phrase and increase its frequency. If the search term has now entered the top 10 queries for additional nodes, we will need to update the top 10 queries for those nodes as well. To do this, we will need to travel all the way up to the root from the node, checking if the current query is in the top 10 for each parent node. If it is, we will adjust the frequency accordingly. If the current query is not in the top 10, we will check if it has a high enough frequency to be included. If it does, we will add the new phrase and remove the term with the lowest frequency.

What is the best way to delete a term from the trie?

Suppose we need to delete a term from the trie due to legal concerns, hatred, or piracy, we can handle this in two ways. First, during the normal update process, we can completely delete the offending phrases from the trie. In the meantime, we can also add a filtering layer to each server that will remove any such terms before sending them to users. This will ensure that users do not have access to the deleted terms until they are fully removed from the trie.

Different ranking criteria for suggestions

When ranking phrases, we must consider more than just a simple count. Other criteria such as freshness, user location, language, demographics, and personal history must also be taken into account. These factors can help us provide more relevant and accurate search results to users.

Caching

Caching is a performance optimization technique that involves storing frequently accessed data in a temporary storage area, called a cache, in order to speed up future access to that data. In the context of searching, before making a query to a database, the server will first check a cache to see if the search terms and their corresponding results are already stored there. If they are, the server can retrieve the results from the cache. This can help reduce the overall latency of the search process.

One way to implement caching for frequently searched terms is to place a cache in front of the database servers that contain the most commonly searched terms and their suggestions. To ensure that the cache stays up to date, it is important to use a caching strategy like Least Recently Used (LRU), which removes the least recently used items from the cache in favor of more recent items. This is based on the idea that recent search results are more likely to be searched again in the future.

Database Partitioning

There are a few different approaches we can take to partition our data in order to distribute the load across multiple servers. One approach is to **partition the data based on the first letter** of the search term. For example, we could store all words beginning with the letter 'A' on one server, and all words beginning with the letter 'B' on another server. This approach has the advantage of being easy to implement, but it can result in uneven partitioning, with some servers handling significantly more traffic than others.

Another approach is to **use a hash function** to determine which server a particular search term should be stored on. The hash function generates a server number based on the search term, which allows us to store the term on the appropriate server. This approach can result in a more evenly distributed load, but it requires us to send queries to all of the servers in order to generate a complete list of suggestions, which can increase latency.

As a result, we can take the first strategy. We can combine those characters with fewer terms onto one server.

Replication and Load Balancer

To ensure that our autocomplete service can handle a large volume of traffic and remain available even if one of our servers goes down, we need to implement replicas and a load balancer. The replicas are copies of our trie servers, which allows us to distribute the load across multiple servers and improve fault tolerance. If one server goes down, the others can continue to handle traffic.

The load balancer is responsible for managing the distribution of traffic across the servers. It tracks our data partitioning method and redirects incoming traffic to the appropriate server based on the prefix of the search term. This helps to ensure that the load is evenly distributed across the servers and that the service remains available even if one server becomes overwhelmed.

Fault Tolerance

If one of the trie servers goes down, we can use a master-slave arrangement to maintain availability. In this arrangement, we have a primary server (the master) that handles traffic and a secondary server (the slave) that serves as a backup. If the master server goes down, the slave can take over after failover to ensure that the service remains available.

Additionally, if any server needs to be restarted, it can use the last snapshot of the trie to recreate it. This ensures that we don't lose any data and that the service can continue to function without interruption. By implementing these measures, we can ensure that our autocomplete service remains available and can handle a large volume of traffic.

Thanks to Navtosh Kumar for his contribution in creating the first version of this content. Please write in the message below if you want to share more insight. Enjoy learning, Enjoy system design :)