Data Structure for Storing Prefixes

Learn about the efficient data structure that’s used to store the search suggestions.

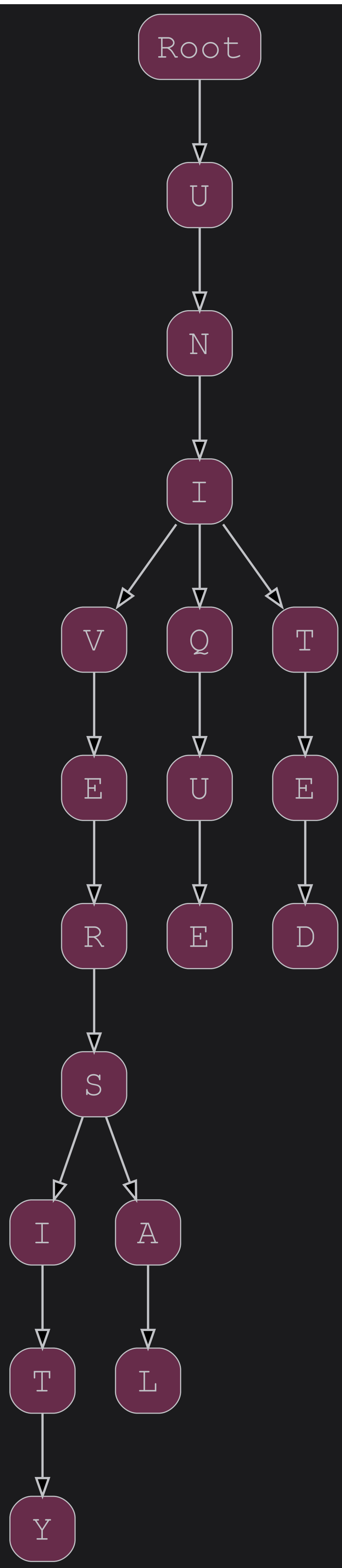
We'll cover the following

The trie data structure

Before we move on to the discussion of the detailed design of the typeahead suggestion system, we must choose an efficient data structure to store the prefixes. Prefixes are the groups of characters a user types. The issue we’re attempting to tackle is that we have many `strings` that we need to store in a way that allows users to search for them using any prefix. Our service suggests the next words that match the provided prefix. Let’s suppose our database contains the phrases `UNITED`, `UNIQUE`, `UNIVERSAL`, and `UNIVERSITY`. Our system should suggest “**UNIVERSAL**” and “**UNIVERSITY**” when the user types “**UNIV**.”

There should be a method that can efficiently store our data and help us conduct fast searches because we have to handle a lot of requests with minim latency. We can’t rely on a database for this because providing suggestions from the database takes longer as compared to reading suggestions from the RAM Therefore, we need to store our index in memory in an efficient data structure. However, for durability and availability, this data is stored in the database.

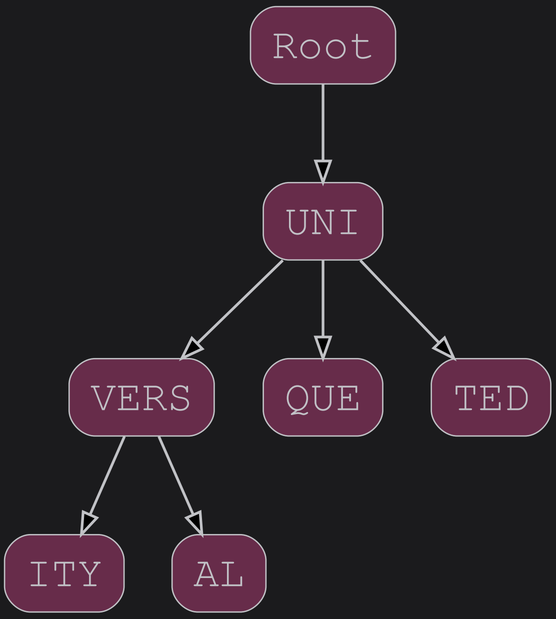
The **trie** (pronounced “try”) is one of the data structures that’s best suited to our needs. A **trie** is a tree-like data structure for storing phrases, with each tree node storing a character in the phrase in order. If we needed to store `UNITED`, `UNIQUE`, `UNIVERSAL`, and `UNIVERSITY` in the trie, it would look like this:



The trie for UNITED, UNIQUE, UNIVERSAL, and UNIVERSITY

If the user types “UNIV,” our service can traverse the trie to go to the node `V` to find all the terms that start with this prefix—for example, `UNIVERSAL`, `UNIVERSITY`, and so on.

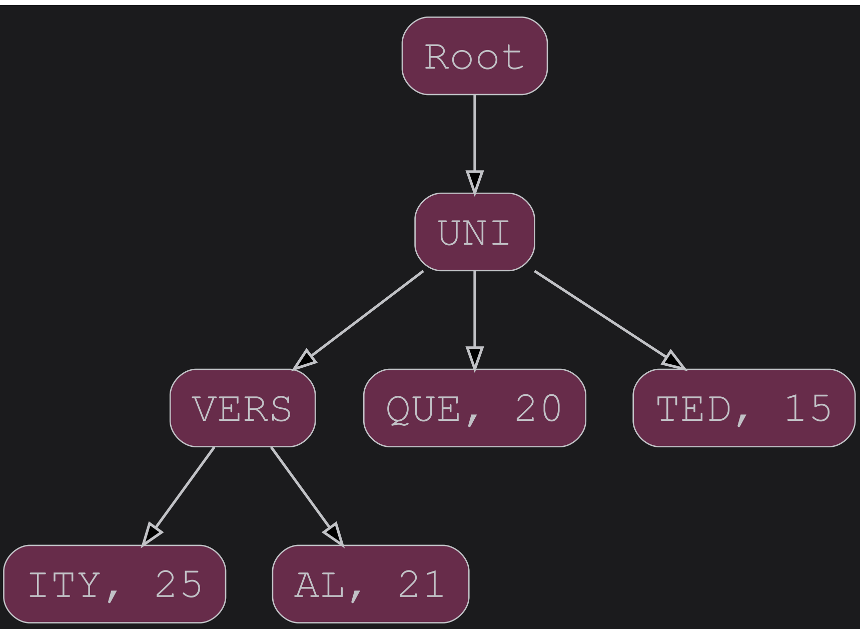
The trie can combine nodes as one where only a single branch exists, which reduces the depth of the tree. This also reduces the traversal time, which in turn increases the efficiency. As an example, a space- and time-efficient model of the above trie is the following:



A reduced Trie for UNITED, UNIQUE, UNIVERSAL, and UNIVERSITY

Track the top searches

Since our system keeps track of the top searches and returns the top suggestion, we store the number of times each term is searched in the trie node. Let’s say that a user searches for `UNITED` 15 times, `UNIQUE` 20 times, `UNIVERSAL` 21 times, and `UNIVERSITY` 25 times. In order to provide the top suggestions to the user, these counts are stored in each node where these terms terminate. The resultant trie looks like this:



A trie showing the search frequency for UNITED, UNIQUE, UNIVERSAL, and UNIVERSITY

If a user types "UNI," the system starts traversing the tree under the root node for `UNI`. After comparing all the terms originating from the root node, the system provides suggestions of all the possible words. Since the frequency of the word `UNIVERSITY` is high, it appears at the top. Similarly, the frequency of the word `UNITED` is relatively low, so it appears last. If the user picks `UNIQUE` from the list of suggestions, the number against `UNIQUE` increases to 21.

Point to Ponder

Question

We reduced the time to traverse the trie by combining nodes with single branches and reducing the number of levels. Is there any other way to minimize the trie traversal time?

Show Answer

Trie partitioning

We aim to design a system like Google that we can use to handle billions of queries every second. One server isn't sufficient to handle such an enormous amount of requests. In addition to this, storing all the prefixes in a single trie isn't a viable option for the system's availability, scalability, and durability. A good solution is to split the trie into multiple tries for a better user experience.

Let's assume that the trie is split into two parts, and each part has a replica for durability purposes. All the prefixes starting from "A" to "M" are stored on Server/01, and the replica is stored on Server/02. Similarly, all the prefixes starting from "N" to "Z" are stored on Server/03, and the replica is stored on Server/04. It should be noted that this simple technique doesn't always balance the load equally because some prefixes have many more words while others have fewer. We use this simple technique to understand partitioning.

We can split the trie into as many parts as we wish to distribute the load on to different servers and achieve the desired performance.

Partitioned Trie

Prefixes	Primary	Secondary
A to M	Server/01	Server/02
N to Z	Server/03	Server/04

Point to Ponder

Question

Where will the mapping between the prefixes and their primary and secondary storage be stored? Who will manage and direct the requests to these servers?

Show Answer

Process a query after partitioning

When a user types a query, it hits the load balancer and is forwarded to one of the application servers. The application server searches the appropriate trie depending on the prefix typed by the user. For example, if a user types something starting from “U,” it either accesses Server/03 or Server/04 since both have the tries stored on them that have prefixes starting with “U.”

Update the trie

Billions of searches every day give us hundreds of thousands of queries per second. Therefore, the process of updating a trie for every query is highly resource intensive and time-consuming and could hamper our read requests. This issue can be resolved by updating the trie offline after a specific interval. To update the trie offline, we log the queries and their frequency in a hash table and aggregate the data at regular intervals. After a specific amount of time, the trie is updated with the aggregated information. After the update of the trie, all the previous entries are deleted from the hash table.

Prefixes and Their Frequencies Updated Periodically

Prefix	Time Interval (One Hour)	Freque
UNIVERSITY	1st hour	[illegible]
UNIVERSITY	2nd hour	[illegible]
UNIVERSITY	3rd hour	100

We can put up a **MapReduce (MR)** job to process all of the logging data regularly, let’s say every 15 minutes. These MR services calculate the frequency of all the searched phrases in the previous 15 minutes and dump the results into a hash table in a database like Cassandra. After that, we may further update the