## **Siri Concept Matcher**

## Objective

* The objective of this document is to design the concept matcher mechanism.
* The concept matcher mechanism matches the user provided input string of words in the list of concepts.
* The concept matcher mechanism finally provides any and all matches of words in the list of concepts, otherwise returns None or empty list.

## Requirements and Assumptions

Following are some of the requirements and assumptions for the solution.

* There will be at most 20 words in the input string.
* The words are separated by one or more whitespace characters.
* Any punctuation in the input should be treated as not significant.
* List of concepts to check against could run into the millions. So, these lists of concepts cannot be stored inside memory and instead will be stored in persistent storage.
* Returns any and all matches from a given list of concepts.
* Solution has to run on mac with readily available utilities or libraries.
* Individual words in given input string can have duplicates.
* Only valid words ("[A-Za-z] +") in input string, need to be matched.
* Each word in input string and each word in concept list is case insensitive. Example: Strings *“ABC”, “Abc”, “ABc”, “aBC”, “aBc”, “abc”* etc., represents the same word (or) concept.
* Order of alphabets does matter in each concept word. Example: “ABC” and “CAB” represents two different concepts.
* Any punctuation in the input should be treated as not significant. Standard punctuation characters are as follows. So, if the word in input string contains any punctuation characters, they will be removed and string will be searched in concept list lookup.
  + , is a comma.
  + ? is a question mark.
  + ! is an exclamation mark.
  + ' is an apostrophe.
  + " is a quotation mark/inverted comma.
  + : is a colon.
  + ; is a semicolon.
* Zero false positives for concept list look up.
* Relative ordering of words in input string is important. Example: “Abc is a good boy” and concept list contains “boy good”, it’s still considered a non-match.
* Concept list supports only en-us. No support for other locales.
* Concept words “A B” and “A B” are same. I.e., the number of spaces between two concept words is significant.

## Definitions

* **Word**: Sequence of characters matched with the given regex ("[A-Za-z] +").
* **Concept**: Sequence of characters matched with the given regex ("[A-Za-z] +").

## Tenants

* Faster look up times for given word search in list of concepts.
* String search in list of concept words is eventually consistent. Any updates to concept list will be available for lookups eventually.

## Scope

### In Scope

* Design for scalable, available and eventual consistent concept list storage structure.
* Concept list partition scheme among multiple storage nodes.
* Concept Matcher Algorithm for matching input string among list of concepts.
* Sample POC implementation for the concept matcher algorithm.

### Out of Scope

* The document doesn’t mention the implementation details to store/update the new entries in concepts list. However, it will mention the usage of concept list storage though.
* The current document doesn’t mention the concept list updater or synchronization between concept list updater and concept list lookup programs. If need be, we can synchronize the reader and updater based upon requirement.

## Algorithm

### Partitioning Scheme for Concepts: Scalability, fault tolerance, Availability:

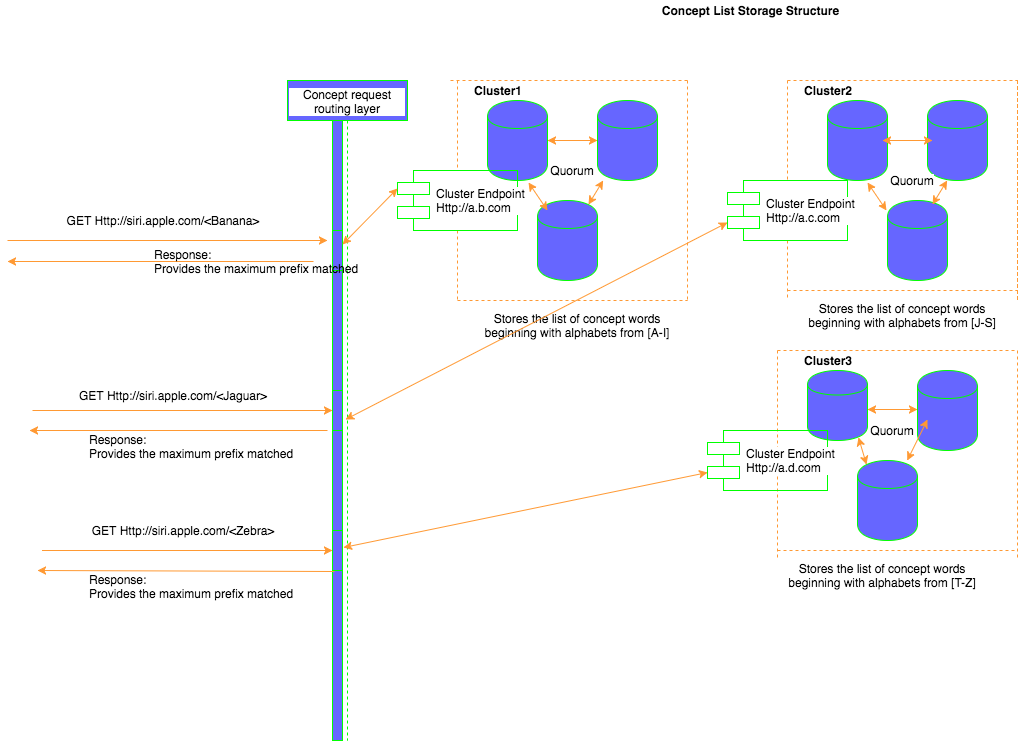
* The list of concepts can be huge and may not fit in to a memory in single node. So, we will ***horizontally partition*** the list of concept words in to multiple concept list partitions, and store each of these partitions into distributed storage nodes.
* Each individual partition of concept list can be stored as either ***hash tables*** ***(or)*** ***tries*** ***(or)*** ***B-tree (or) Bloom filters***. We will store each concept list partition as ***trie*** trees. Each storage node stores concept list words as lower-case entries in individual ***trie*** trees. Concept words will be stored by removing the spaces between them. Bloom filters may ***have false positive***s that negates our requirement.
* Storing Concept list partition on each storage node using ***trie*** structure locally helps in meeting our requirement, where we need to match any and all matches. Bloom filters assumes false positives. If false positives are allowed, we can use ***bloom filter***, a space efficient data structure, otherwise a ***trie*** makes more appropriate for this use case. **Time complexity for search is proportional to *O(key\_length)****,* however the memory requirements of ***trie*** is ***O(ALPHABET\_SIZE \* key\_length \* N).***
* **Scalability, availability, and fault tolerance**: We will maintain distributed, replicated and multiple storage nodes for each concept list partition list. These nodes are grouped into clusters, with each cluster represented by a given endpoint.
  + Multiple storage nodes help in scalability, reliability, fault tolerance and availability. Look up query request for concept list can be routed to an individual storage node based upon the partitioning scheme or consistent hashing.
* Concept word lists are partitioned based upon the starting alphabet of each concept word, in to individual clusters as mentioned below. These are just representative only. More clusters can be added when concept list size grows and each partition size may not fit on a storage node.

|  |  |
| --- | --- |
| Concept Word Initial Alphabet | Hash Table Cluster Endpoint |
| [A-I] | Cluster endpoint (<Http://a.b.com)> |
| [J-S] | Cluster endpoint (<Http://a.c.com)> |
| [T-Z] | Cluster endpoint (<Http://a.d.com)> |

* **Routing Layer:** Routes the incoming request for a given concept word lookup to individual clusters based upon partitioning scheme.
* **Consistency and quorum:** A sloppy quorum will be maintained among storage nodes under each cluster, to derive the consensus. Quorum will be maintained for every write or read and system so is eventual consistency.
* **Replication:** Concept list partitions will be replicated among multiple storage nodes inside of each cluster. Replication provides faster, high throughput lookup and availability.

### Concept List Storage Illustration

* Below diagram illustrates the concept list storage scheme. Below we mentioned of an example illustration where
* Note: The number of clusters can be scaled up, depending upon the size of storage node.



### Storage: List of Concepts

The list of concepts mentioned to be huge, and these may not fit into memory. So, we will store the list of concepts in an external store. Below we provide different mechanisms to store the list of concepts:

* **Partition concept list horizontally into individual storage nodes:** Details of this approach mentioned earlier.
* **Hosted cloud based NoSQL solution say Dynamodb:** We can store the list of concept words inside NoSql store say Dynamodb. Each concept string name is unique, and can be used as a partition key. It provides uniform distribution of concepts across different dynamo partitions. The concept names can be used as both hash key and sort key, which helps to store concepts with same hash key in a list under the same partition of Dynamodb. Dynamodb provides low cost, reliable, highly available and low latency data store which can be accessed through simple API’s. However, the requirement for current solution is mentioned to be running on mac readily, without any external dependencies. So, Dynamodb may not be a viable option.
* **SqlLite** is as an embedded database and is readily available on mac. Each storage node for concept list will use SqlLite to store the concept list and so lookup for the words. It provides the ready-made index **B-tree** as a self-balancing data structure that keeps data sorted and allows searches in [logarithmic time](https://en.wikipedia.org/wiki/Logarithmic_time) O(n).
* **Hash Algorithm for concept word:** The hash value for each concept string can be computed as follows:

**public int** conceptWordHashValue (**final** String concept) {  
 **char**[] charArray = concept.toCharArray();  
 **int** h = 0;  
 **if** (h == 0 && charArray.**length** > 0) {  
 **char** val[] = charArray;  
 **for** (**int** i = 0; i < charArray.**length**; i++) {  
 h = 31 \* h + val[i];  
 }  
 }  
 **return** h;  
}

* + **Record Structure:**

Each record stored under hashtable, contains the following structure (Example reference):

|  |  |
| --- | --- |
| HashId | Concept String |
| 123 | ABC |
| 234 | CAB |
| 789 | DEF, GHI (Two strings match with same HashId) |

* Each record can contain multiple concept words, in case of hash collision. The individual concept words for same HashId will be separated by a delimiter say “,”.
* **File based store:** In this mechanism, we will maintain our own external file store mechanism persisted with the list of concepts. The concept lists will be partitioned into multiple file sets for faster lookups and so to reduce latency. The details of this mechanism, is out of scope for this document.

#### Conclusion:

1. We will partition the cluster list horizontally and store them on individual nodes.
2. For the current assignment, we will not use SqlLite, but instead store concept lists as partitioned ***trie*** data structures on multiple storage nodes. Using trie, we can derive all matches for a given prefix and find any and all matches of input string words against concept list.

### Concept Matcher Algorithm

* Initial worker pool of size 20 (Max words in input string is mentioned to be 20). Parallelizes the concept list look up in case the input string contains 20 different words, and each worker will be used to lookup different word inside concept list.
* Set WordDelimiter = “\\s+”.
* Set InputString = User provides the input string. Example: *“Abc is a := good and kind boy spelled as ABC sometimes.”*
* If InputString is Invalid (empty or null), throw an invalid argument exception.
* Set ListOfWords = InputString.split(WordDelimiter).
* If ListOfWords is empty, throw an invalid argument exception.
* Sanitize the input string i.e., remove punctuations from each word, then discard invalid words, convert all words to lower case. Valid words are defined by regex ("[A-Za-z] +").
* Set ValidWordList = null; // Data structure to hold only valid words. Example, ValidWordList = [“abc”, “is”, “a”, “good”, “and”, “kind”, “boy”, “spelled”, “as”, “sometimes”]. Invalid words [“: =”] will be removed.
* If ValidWordList is invalid (empty or null), throw an invalid argument exception.
* Set MatchedWordsList = null; //Stores the list of words matched with concept list lookup.
* For each word (say currentWord) in ValidWordList [“abc”, “is”, “a”, “good”, “and”, “kind”, “boy”, “spelled”, “as”, “sometimes”], do the following:
  + Set currentWordIndex = Index of currentWord in ValidWordList.
  + Set ValidWordListSize = Size of ValidWordList.
* Route the look up request for each word to relevant concept list cluster based upon initial alphabet. Example: Look up request for “*abc*” is routed to *cluster1*, similarly, “*sometimes*” will be routed to *cluster2* etc.
  + Storage nodes under individual cluster store concept words as ***Trie*** tree.
  + Each storage node does a look up for these words to derive ***all*** the words with given ***prefix*** found in tree.
  + Set PrefixMatchList = List of all words matched in trie structure with given prefix of currentWord.
  + If the obtained prefix list is not null and empty
  + Verify if the currentWord (ignore case) is available in prefix list. // PrefixMatchList contains all matched strings.
  + If yes, add the currentWord to the MatchedWordsList.
  + For each word (tempWord) in the ValidWordList from [currentWordIndex+1, ValidWordListSize], do the following:
    1. Set tempStr = currentWord;  
       Set toAddd = currentWord;  
       tempStr = tempStr + tempWord;  
       toAddd = toAddd + **" "** + tempWord;  
       Verify tempStr.toLowerCase()) in prefixMatches.indexOf(tempStr.toLowerCase())  
       If found, add toad to the MatchedWordList
* If MatchedWordsList is empty or null, return empty list.
* Else: return None/empty list.