Atuocomplete Based on Ternary Search Tree

Niu Wenhao, Lu Yan, Wang Sangtian

June 26, 2016

1 Discription

We realize the autocomplete function for a search engine like *Acemap*. We can predict what users want to search according to the keyword they input. In order to achieve better performance than Redis(a open sourse memory system using by *Acemap*) in time complexity and space complexity, we built a memory system based on $Ternary\ Search\ Tree(TST)$. We choose $Java$ to realize TST, and $Apache\ Tomcat$ 8.0 as our server.

2 Comparison between different data structure

The figure below shows the time complexity and space complexity of different data structures.

	Time Complexity	Space Complexity
Redis	O(L)	O(N * L * L) (should be better since there could be many collisions)
Trie	O(K * L)	O(N * L * R)
Trie with hashmaps	O(K * L)	O(N * L)
TST	O(K * (L + log N))	O(N * L)

Figure 1: Complexity Analyzation

where N represents the number of words. L represents the average length of words. K represents the number of matches we wish to return. R represents the size of alphabet.
From the comparison we can draw a conclusion that TST performs better that

the others. But the height of TST is greatly infected by the order of the letters that we insert, and the height of the tree has a remarkable impact on the performance of TST on time and space. So balancing the TST is wise choice if we intend to have a better performance.

3 Trie Tree

Trie tree, also known as a trie, the word search tree or prefix tree, is a multi-tree structure for fast retrieval. For example, English alphabet trie is a tree 26, a digital trie is a 10-tree.

3.1 The definition of Trie Tree

Trie tree and binary search tree different keys are not stored directly in the node, but is determined by the position of the node in the tree. All descendants of a node has the same prefix (prefix), that is, the node corresponding to the string, and the root node corresponds to an empty string. Under normal circumstances, not all of the nodes have a value corresponding to only part of the internal nodes and leaf nodes corresponding key only related values.
Trie tree can utilize a common prefix string to save storage space, as shown below, the Trie tree with 11 nodes holds eight string tea, ted, ten, to, A, i, in, inn.
We note Trie tree, string tea, ted, and ten of the same prefix (prefix) as "te", if we want to store the string most have the same prefix (prefix), then the Trie tree structure can save a lot of memory space, because the Trie each word is stored by character by character method, it is shared with the same prefix word prefix node.
Of course, if the presence of Trie tree has lots of strings, and these strings are basically no common prefix, then the corresponding Trie tree consuming memory, null pointer Trie drawback is the cost of memory space.The basic nature of the Trie can be summarized as:
(1) The root node does not contain characters, each node except the root contains only one character.
(2) From the root to a node on the path through the hyphenation up for the corresponding string node.
(3) String all child nodes of each node contains is not the same.

3.2 The Achievement of Trie Tree

Trie tree is a shape of a tree data structure in which each node contains an array of pointers, suppose we want to build a 26-letter Trie tree, then each pointer corresponds to a letter of the alphabet. Starting from the root, as long as we turn to find the target in the next letter in the word corresponding to the pointer, you can find a step by step goal. Suppose, we want the string AB,

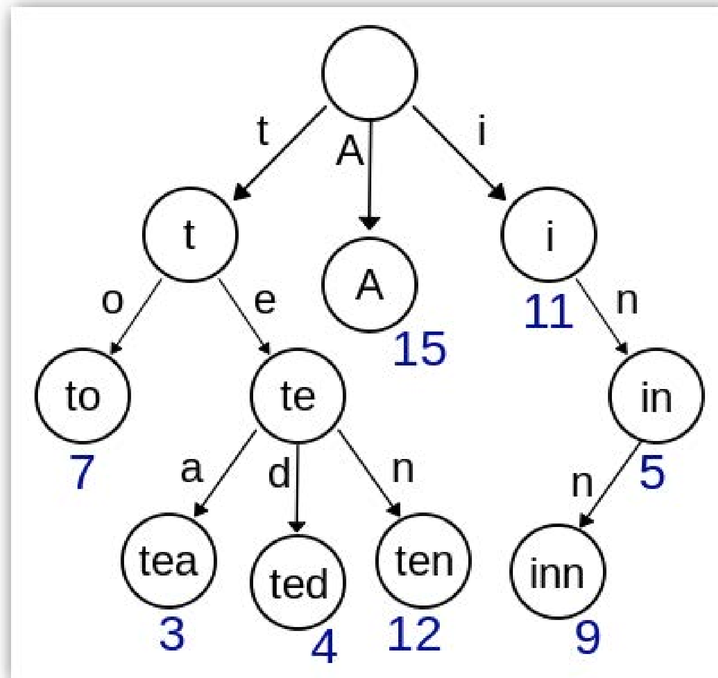

Figure 2: Trie Tree

ABBA, ABCD and BCD into the Trie tree, because the root of the Trie not save any letter, we began to save letters from the direct successor of the root node. As shown below, we save on the second floor Trie tree letters A and B, the third layer holds B and C, where B is marked dark blue word AB has been inserted completed.
We found that as Trie each node has a pointer to an array of length 26, but we know that not every array of pointers to records are kept, empty array of pointers to lead to waste of memory space.
Suppose we want to design a translation software, translation software and ultimately, search word function, and when the user enters a query word, the software prompts similar words, lets the user select a query word, so that users do not need to be able to enter the full word query, and a better user experience.
We will use the Trie structure for storing and retrieving words, in order to achieve intelligent prompts words, here we consider only the 26 English letters matched to achieve, so we will build a 26-tree.
As we finally adopted the trigeminal search tree on T r i e tree implementation is omitted here.

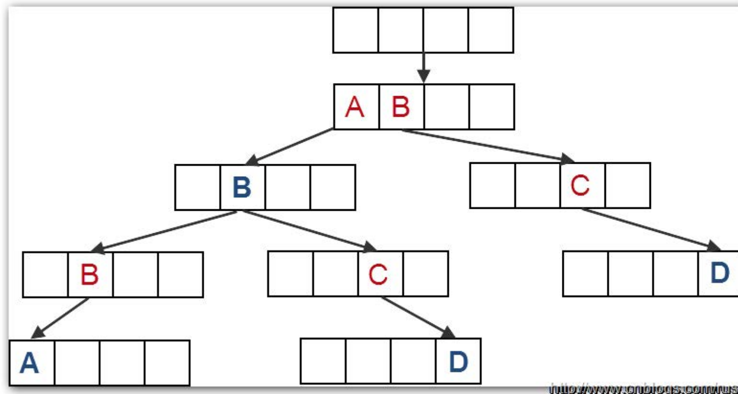

Figure 3: Implement of Trie Tree

4 Ternary Search Tree

Ternary search tree is a special kind of Trie tree data structure, it is a mixture of digital search tree and binary search tree. It has both digital search tree efficiency advantages, but also binary search tree space advantages. Ternary search tree using a clever means to solve the Trie memory problems (pointer array empty). In order to avoid unnecessary pointer memory for each node Trie no longer be represented by an array, but represented as "tree to tree." Trie nodes in each non-null pointer will get part of its own node in the ternary search tree.
Next, we will implement the trigeminal search tree node class, specifically to achieve the following:

Due to the ternary search tree contains three types of arrows. The first in the arrow arrows and Trie is the same, that is, in Figure 2 drawn down arrow dotted line. Travels along the down arrow, it means that "matches" the starting end of the arrow character. If the node is less than the current character in the character, looks to the left along the node, look to the right and vice versa.
Next, we will define Ternary Tree type, and add Insert () and Find () function

Because each node in the search tree ternary only three forks, so we conduct node insertion, just judge inserted character related to the current node (less than, equal to, or greater than) into the corresponding node on OK.

We use the previous example, the string AB, ABBA, ABCD and BCD into the ternary search tree, the tree is inserted into the first string AB, then we insert the string ABCD, since the ABCD and AB have the same prefix AB, Therefore, the node C are stored in CenterChild B, D to C stored in the CenterChild; ABBA when inserted, since the ABBA-AB have the same prefix AB,

```java
    public class Node{
        public Node right;
        public Node left;
        public Node middle;
        public char val = EMPTY;
        public long weight;
        public long maxSubWeight;
        public String word;

        @Override
        public String toString(){
            StringBuilder sb = new StringBuilder();
            sb.append(val + " ");
            if(middle != null)
                sb.append(middle.toString());
            if(left != null)
                sb.append(left.toString());
            if(right != null)
                sb.append(right.toString());
            return sb.toString();
        }
    }

    @Override
    public String toString(){
        return root.toString().trim();
    }

}
```

Figure 4: Node of Ternary Search Tree

B and C fewer characters than the character, so that the B to C stored LeftChild in; when inserted BCD, since the character B is greater than the characters a, so B to C RightChild in storage.
We note that the order of insertion strings would affect the structure of the ternart search tree, in order to obtain the best performance, the string should be in a random order into the ternary tree search tree, in particular, should not be inserted in alphabetical order, otherwise correspond to a single Trie child tree nodes will degenerate into a linked list, find greatly increased costs. Of course, we can also use some of the ways to achieve self-balancing balanced ternary tree.
Since the tree is balanced depends on word order reads, if inserted in sorted order, the way the resulting tree is the most uneven. Word read into the order

```
private void insert(String s, long w, Node n, String word){
    //get first character
    char first = s.charAt(0);
    //if node val is empty, put first char there
    if(n.val == EMPTY){
        n.val = first;
    }
    //if maxSubWeight < weight, make max = weight
    if(n.maxSubWeight < w){
        n.maxSubWeight = w;
    }
    if(first == n.val){
        if(s.length() > 1){
            //if node val matches first character and s.length > 1,
            //insert tail of s in middle
            if(n.middle == null)
                n.middle = new Node();
            insert(s.substring(1), w, n.middle, word);
        } else {
            //if s is length==1, this is the last character
            //if val = first  and weight is > 0, then this word has been inserted
            //already, throw exception
            //otherwise, set val = char and weight = weight
            n.weight = w;
            n.word = word;
        }
    } else if(first < n.val){
        //if node val is greater than first character,
        //get left node
        //if left is null, make it a new node
        //insert whole string into left
        if(n.left == null)
            n.left = new Node();
        insert(s, w, n.left, word);
    } else {
        //otherwise do the whole process for right
        if(n.right == null)
            n.right = new Node();
        insert(s, w, n.right, word);
    }
```

Figure 5: Find() Function

for the creation of a balanced search tree ternary very important, so we choose a middle value after sorting the data set, and use it as the starting node through continuous binary insertion intermediate values, we can create a balance Triple tree.

5 Implement TST in search engine

5.1 Different kinds of data

We have five kinds of data include affiliations, authors, conference, field, journal. In order to import these data, we build five TSTs to save these data and search in these five TSTs every time.

5.2 Same names

There are many authors have same name. To solve these problem, we add authors ID behind their name When build TST. For example, there are many authors named "tao wang". When building TST, we insert "tao wang: 80F5E6B7"?"tao wang:7DAE408A"?"tao wang:7F9D979B". When users input "tao wa", all "tao

wang" will be returned and distinguished by ID. The result is shown below.

"authors":["AuthorID":"80F5E6B7", "AuthorName":"tao wang", "PaperCounts":"tao an:7ED7DAE8" , "AuthorID":"7DAE408A", "AuthorName":"tao wang", "PaperCounts":"2344" , "AuthorID":"7F9D979B", "AuthorName":"tao wang", "PaperCounts":"1227" , "AuthorID":"81B85205", "AuthorName":"tao wang", "PaperCounts":"817" , "AuthorID":"8549B91E", "AuthorName":"tao wang", "PaperCounts":"540" , "AuthorID":"852D6A16", "AuthorName":"tao wang", "PaperCounts":"416" , "AuthorID":"80CB0BA7", "AuthorName":"tao wang", "PaperCounts":"338" , "AuthorID":"854CF78C", "AuthorName":"tao wang", "PaperCounts":"300"]

5.3 Wrong inputs by users

Users may type in wrong prefix, and we can tolerate most of error. First kind of error: user may want to search "xinbing wang", but they input "xinbnig" as prefix, "i" and "n" is exchanged. This is really a prevalent error. To solve this kind of error, we exchange the every adjacent characters from the second character and search again. Finally sort all the result by weight. When user input "xinbnig", the result is shown below.

"authors":["AuthorID":"7E0DFF97", "AuthorName":"xinbing wang", "PaperCounts":"311" , "AuthorID":"7D928FFE", "AuthorName":"xinbing zhao", "PaperCounts":"162" , "AuthorID":"80873A46", "AuthorName":"xinbing yu", "PaperCounts":"117"]

Second kinds of error: users may insert a character for two times. For example, user may type in "xinbinng". To solve this problem, we delete each character from the second character and search again. When user type in "xinbinng", the result is shown below. When searching "brian whit" the result is shown below.

"authors":["AuthorID":"7E0DFF97", "AuthorName":"xinbing wang", "PaperCounts":"311" , "AuthorID":"7D928FFE", "AuthorName":"xinbing zhao", "PaperCounts":"162" , "AuthorID":"80873A46", "AuthorName":"xinbing yu", "PaperCounts":"117"]

5.4 Middle name

Some users are not familiar with the second name of authors, and they may only know the author's first name and last name. So we should take this problem into consider. Users may only type in first name and last name and ignore middle name. For example, author "brian e whitacre" has a middle name "e". And user may only type in "brian whit" when search this author. When we find that there are only one space in users input, we will insert 26 letter "a" to "z" in space and search again.

"authors":["AuthorID":"7EE6750B", "AuthorName":"brian w whitcomb", "PaperCounts":"104" , "AuthorID":"78151475", "AuthorName":"brian hitsman", "PaperCounts":"56" , "AuthorID":"00016A3E", "AuthorName":"brian e whitacre", "PaperCounts":"51" , "AuthorID":"7D40AD69", "AuthorName":"brian r white", "PaperCounts":"51"]

6 Comparison on performance

We test our memory system against *Redis* on the server, using 100,000 words as our database. The result shows , compared with *Redis*, our memory system cut down half of the memory cost. For *Redis*, it will take a long time (nearly 30 sec) to return a result if the keyword is the first time to search. Whereas, our memory is not bothered with this problem and can return the result almost immediately even the number of the words is servral million.