基于三叉搜索树的网页 AUTOCOMPLETE 的实现

无线通信原理项目报告

2016-5-15

目录

1. 简述

我们实现了搜索框自动匹配功能，当用户开始输入想要搜索的关键词时，我们会根据当前输入的字符，我们会预测用户想要搜索的完整的关键字，并显示在下拉框中。为了减小时间复杂度和空间复杂度，我们采用了三叉搜索树(TST)来实现此功能。相比于其他方式(如 redis，Trie 等)，TST 占用内存明显更小，搜索速度上也有一定优势。在具体实现方法上，我们采用 Java 编写 TST 主程序，并使用 Tomcat 实现前端与后端交互。

2. 各种实现方法复杂度的比较

我们在选择几种不同的数据方法在缓存中存储数据的时候，考虑了他们的时间复杂度。具体见下表。

	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)

N = # number of words　　L = # average length of words

K = # number of matches we wish to return　　R = # size of alphabet

通过对比可以发现，TST 在时间复杂度上的优势。但是单词字母的插入顺序会很大程度上影响树的高度，而树的高度又与它的性能息息相关。

时间复杂度：

设树的高度为 h，查找字符串的时间复杂度：o(logh)；插入字符串的时间复杂度：o(logh)

缺点

匹配效率依赖于树的高度，可以在建 TST 之前，可采取相应措施，使得 tst 为平衡树，提高匹配的时间效率。具体方法后文会介绍。

3. Trie 树

Trie 树，又称字典树，单词查找树或者前缀树，是一种用于快速检索的多叉树结构，如英文字母的字典树是一个 26 叉树，数字的字典树是一个 10 叉树。

3.1 Trie 树的定义

Trie 树与二叉搜索树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀（prefix），也就是这个节点对应的字符串，而根节点对应空字符串。一般情况下，不是所有的节点都有对应的值，只有叶子节点和部分内部节点所对应的键才有相关的值。

Trie 树可以利用字符串的公共前缀来节约存储空间，如下图所示，该 Trie 树用 11 个节点保存了 8 个字符串 tea，ted，ten，to，A，i，in，inn。

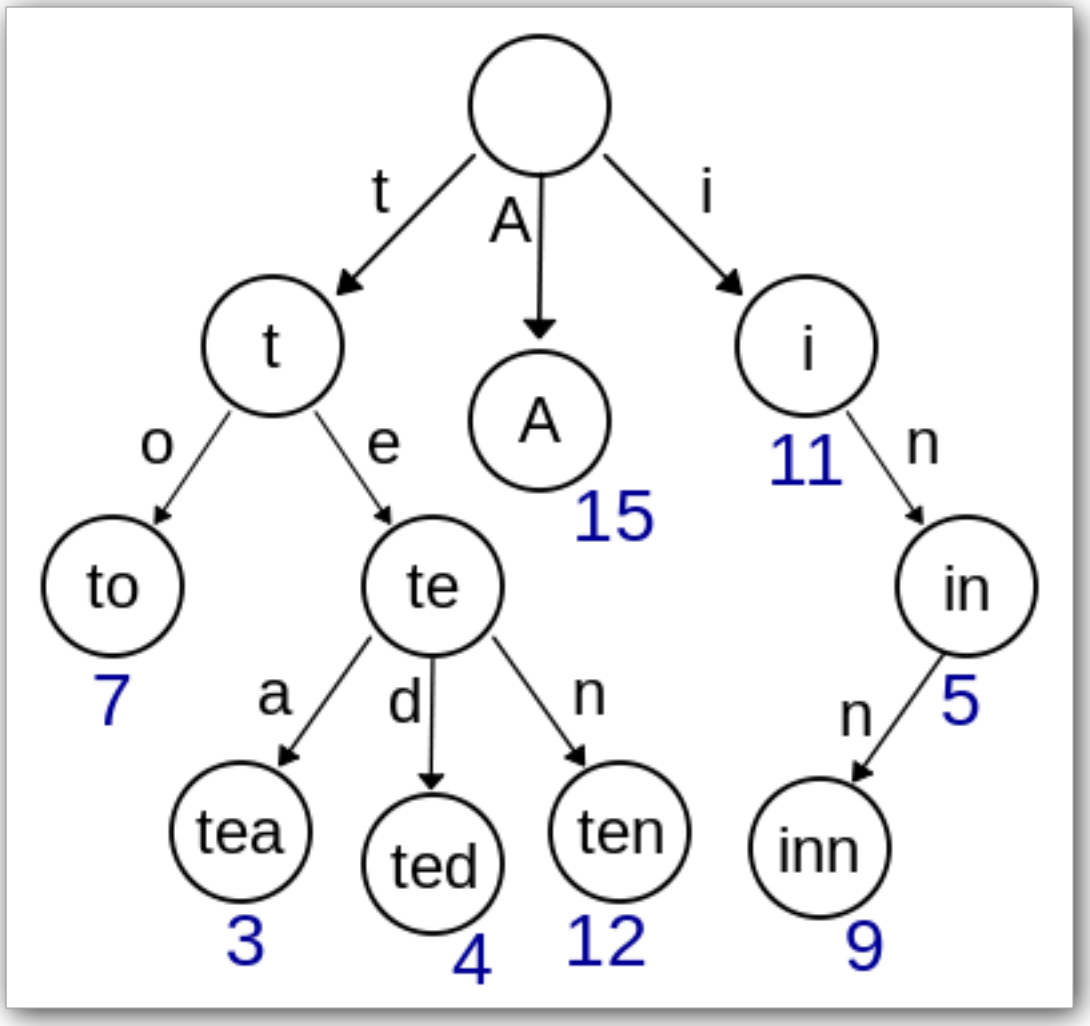


图 1Trie 树

我们注意到 Trie 树中，字符串 tea，ted 和 ten 的相同的前缀（prefix）为“te”，如果我们要存储的字符串大部分都具有相同的前缀（prefix），那么该 Trie 树结构可以节省大量内存空间，因为 Trie 树中每个单词都是通过 character by character 方法进行存储，所以具有相同前缀单词是共享前缀节点的。

当然，如果 Trie 树中存在大量字符串，并且这些字符串基本上没有公共前缀，那么相应的 Trie 树将非常消耗内存空间，Trie 的缺点是空指针耗费内存空间。

Trie 树的基本性质可以归纳为：

（1）根节点不包含字符，除根节点外的每个节点只包含一个字符。

（2）从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。

（3）每个节点的所有子节点包含的字符串不相同。

3.2 Trie 树的实现

Trie树是一种形似树的数据结构，它的每个节点都包含一个指针数组，假设，我们要构建一个26个字母的Trie树，那么每一个指针对应着字母表里的一个字母。从根节点开始，我们只要依次找到目标单词里下一个字母对应的指针，就可以一步步查找目标了。假设，我们要把字符串AB，ABBA，ABCD和BCD插入到Trie树中，由于Trie树的根节点不保存任何字母，

我们从根节点的直接后继开始保存字母。如下图所示，我们在Trie树的第二层中保存了字母A和B，第三层中保存了B和C，其中B被标记为深蓝色表示单词AB已经插入完成。

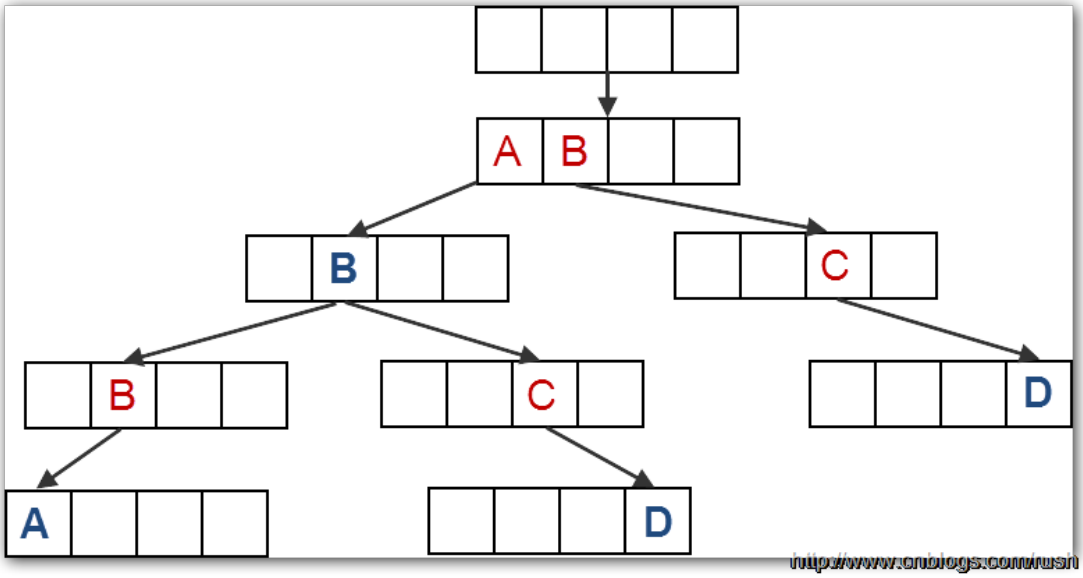


图2 Trie树的实现

我们发现由于Trie的每个节点都有一个长度为26指针数组，但我们知道并不是每个指针数组都保存记录，空的指针数组导致内存空间的浪费。

假设，我们要设计一个翻译软件，翻译软件少不了查词功能，而且当用户输入要查询的词汇时，软件会提示相似单词，让用户选择要查询的词汇，这样用户就无需输入完整词汇就能进行查询，而且用户体验更好。

我们将使用 Trie 树结构存储和检索单词，从而实现词汇的智能提示功能，这里我们只考虑 26 英文字母匹配的实现，所以我们将构建一棵 26 叉树。

由于我们最终采用的是三叉搜索树，关于 Trie 树的具体实现这里不再赘述。

4. 三叉搜索树

三叉搜索树是一种特殊的 Trie 树的数据结构，它是数字搜索树和二叉搜索树的混合体。它既有数字搜索树效率优点，又有二叉搜索树空间优点。三叉搜索树使用了一种聪明的手段去解决 Trie 的内存问题（空的指针数组）。为了避免多余的指针占用内存，每个 Trie 节点不再用数组来表示，而是表示成“树中有树”。Trie 节点里每个非空指针都会在三叉搜索树里得到属于它自己的节点。

接下来，我们将实现三叉搜索树的节点类，具体实现如下：

```
    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();
    }

}
```

图三 三叉搜索书的结点类

由于三叉搜索树包含三种类型的箭头。第一种箭头和 Trie 里的箭头是一样的，也就是图 2 里画成虚线的向下的箭头。沿着向下箭头行进，就意味着“匹配上”了箭头起始端的字符。如果当前字符少于节点中的字符，会沿着节点向左查找，反之向右查找

接下来，我们将定义 Ternary Tree 类型，并且添加 Insert()和 Find()函数

```
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);
    }
```

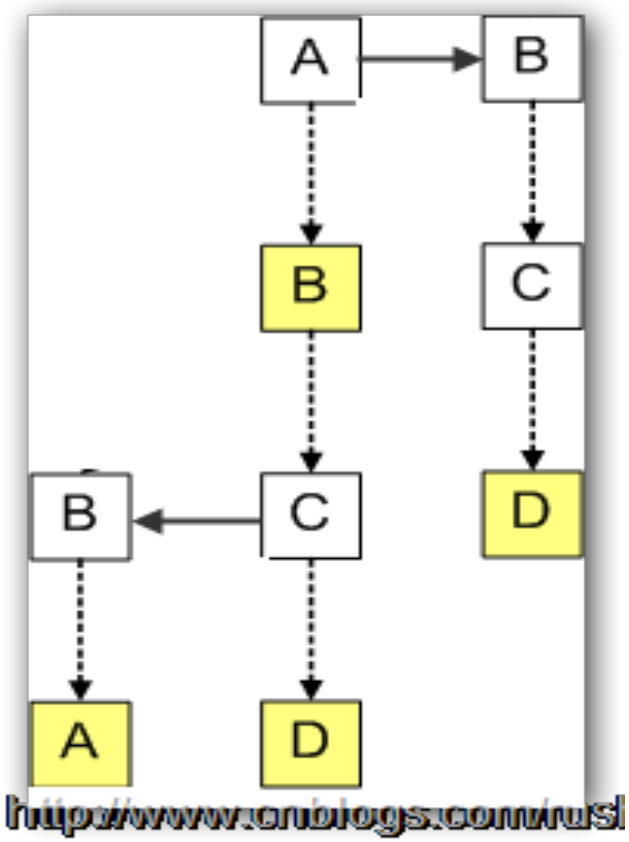
图四 三叉搜索树的 insert 函数

```
private Node findNode(String s, Node n){
    if(s.length() < 1)
        return null;

    //get first character
    char first = s.charAt(0);

    if (n==null)
        return null;

    if(first == n.val){
        if(s.length() > 1){
            //if node val matches first character and s.length > 1,
            //keep looking down middle
            return findNode(s.substring(1), n.middle);
        } else {
            //if s is length==1, this is the last character
            //return the weight
            return n;
        }
    } else if(first < n.val){
        //if node val is greater than first character,
        //get left node
        //if left is null, string is not in trie, return 0
        //otherwise, look in left for whole string
        if(n.left == null)
            return null;
        return findNode(s, n.left);
    } else {
        //otherwise do the whole process for right
        if(n.right == null)
            return null;
        return findNode(s, n.right);
    }
}
```

图五 三叉搜索树的 find 函数

由于三叉搜索树每个节点只有三个叉，所以我们在进行节点插入操作时，只需判断插入的字符与当前节点的关系（少于，等于或大于）插入到相应的节点就 OK 了。

我们使用之前的例子，把字符串AB，ABBA，ABCD和BCD插入到三叉搜索树中，首先往树中插入了字符串AB，接着我们插入字符串ABCD，由于ABCD与AB有相同的前缀AB，所以C节点都是存储到B的CenterChild中，D存储到C的CenterChild中；当插入ABBA时，由于ABBA与AB有相同的前缀AB，而B字符少于字符C，所以B存储到C的LeftChild中；当插入BCD时，由于字符B大于字符A，所以B存储到C的RightChild中。



图六 三叉搜索树

我们注意到插入字符串的顺序会影响三叉搜索树的结构，为了取得最佳性能，字符串应该以随机的顺序插入到三叉树搜索树中，尤其不应该按字母顺序插入，否则对应于单个Trie节点的子树会退化成链表，极大地增加查找成本。当然我们还可以采用一些方法来实现自平衡的三叉树。

由于树是否平衡取决于单词的读入顺序，如果按排序后的顺序插入，则该方式生成的树是最不平衡的。单词的读入顺序对于创建平衡的三叉搜索树很重要，所以我们通过选择一个排序后数据集合的中间值，并把它作为开始节点，通过不断折半插入中间值，我们就可以创建一棵平衡的三叉树。

5. 具体解决的问题

5.1 不同类型数据的处理

导入的数据库包括有 affiliations、authors、conference、field、journal 共 5 类数据，因此我们建立了 5 个的 TST，分别插入这 5 类数据。在搜索时，分别在 5 个 TST 内搜索。其中 authors 库比较大，大约有 73 万个作者，其他库共计约 3 万条数据，整体上数据量在 75 万左右。

5.2 重名问题

数据库中作者存在很多同名。为了解决重名问题，在建树时，我们在作者姓名后面加上作者 ID，使用 ID 区分同名作者。比如，数据库中存在多个作者姓名为“tao wang”，在插入建树时，我们插入“tao wang: 80F5E6B7”、“tao wang：7DAE408A”、“tao wang：7F9D979B”。当用户输入“tao wa”时，将查找出所有姓名为“tao wang”的作者，并

以 ID 加以区分，查找结果如下：


```
"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 用户输入可能存在各种错误。

第一类错误：用户希望搜索“xinbing wang”，但是在输入时输入为“xinbnig”，也就是 n 和 i 输入反了，这是很常见的错误，我们需要能够在这种错误情况下依然返回用户想要的结果。这里我们采用了穷举的方法，从第二个字符开始，将所有的前后前后两两交换并进行搜索，最后将所有结果按照权值排序。当输入“xinbnig”时，结果（json 形式，下同）如下：


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


第二类错误：用户重复输入了某个字符，比如用户输入为“xinbinng”。同样采用穷举的方式，从第二个字符开始，每次删除当前字符，并搜索，依次向后。当输入“xinbinng”时，结果如下：


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

5.4 **对于 middle name 的处理**。

一些用户对欧美人的 middle name 不太熟悉，往往只会记住 first name 和 last name，所以在搜索过程中，需要考虑到用户可能不会输入 middle name。以作者“brian e whitacre”为例，用户可能只会输入“brian whit”，而忽略了 middle name。我们解决方法是在空格处插入不同字母进行搜索。当用户的输入中有且只有一个空格时，我们会在这个空格处轮流插入所有 26 个字母进行搜索。当输入“brian whit”是，结果如下：


```
"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. 性能评估比较

acemap 网站之前使用的 redis。一下分别从理论和实际测试比较一下 redis 和三叉搜索树(TST)的时间和空间复杂度。

理论	时间复杂度	空间复杂度
Redis	O(L)	O(N * L * L)
TST	O(K * (L + log N))	O(N * L)

N = # number of words

K = # number of matches we wish to return

L = # average length of words

实际	时间上的表现	空间上的表现
Redis	在已经缓存过的搜索词搜索上可以达到实时，但第一次搜的词会有几秒的反应时间。	服务器上占的内存较多
TST	在 10 万数据的测试中基本达到实时出现	每 10 万数据占用 100M 内存，相比于之前使用 redis(10 万数据占用 210M 内存)，内存占用量减小一半。

结论：经我们测试，对于现在数据的量级，无论在时间还是空间上的花费，TST 的表现均优于 Redis.和理论的上的分析一致。