

# 一种高效的 XMLQuery 基本模式匹配算法

蒋 科, 郑有才

(西安电子科技大学 计算机学院, 陕西 西安 710071)

**摘 要:**有效的支持结构连接是实现数据库系统 XML 文件查询的关键。结构连接是用来查找所有满足基本的结构关系的元素对,即指定 XML 树型结构文件元素对的关系(父亲-孩子和祖先-子孙的关系)。文中在分析常见的 XMLQuery 模式匹配算法(Stack-Tree 连接算法)的基础上,提出一种改进的 Stack-Tree 连接算法将 Stack-Tree-Desc 算法和 Stack-Tree-Anc 算法统一;并且采用动态分配存储空间方法,比 Stack-Tree-Anc 大大节省了存储空间。最后给出了改进的 Stack-Tree 连接算法分析和试验结果。

**关键词:**结构连接;模式匹配;Stack-Tree 连接算法

中图分类号:TP301.6

文献标识码:A

文章编号:1673-629X(2007)07-0087-04

## A Primitive for Efficient XMLQuery Pattern Matching Algorithm

JIANG Ke, ZHENG You-cai

(School of Computer Science, Xidian University, Xi'an 710071, China)

**Abstract:**Efficient support for structural joins is the key to efficient implementations of XML queries on database system. Structural joins are used to find all pairs of elements satisfying in the query, namely, parent-child and ancestor-descendant relationships. An improved algorithm is proposed to unify two algorithms of Stack-Tree-Desc and Stack-Tree-Anc together; also to use dynamic method to greatly save the storing space. In the end, the analysis of this algorithm's performance and results of experiment are given.

**Key words:**structural joins; pattern matching; Stack-Tree join algorithm

### 0 引 言

当前 XML 已经成为万维网的数据表示和数据交换的标准,对于 XML 数据的管理和查询成为当前研究的热点。在文献[1]提出 XML 具有半结构化的特点,能够很好地表示万维网的异构数据。然而,从另一个方面来说,由于 XML 的不规则性,如何高效地查询大量的 XML 数据成了一个新的挑战。实际上任何一个 XML 文件是一种有序的树型结构,树节点对应于 XML 文件中的元素,一个树节点的相对位置取决于它上下元素的位置。而结构连接是实现数据库系统 XML 文件查询的关键。一种复杂的树型模式可以分解为基本的二元结构关系的集合,比如节点对间父亲-孩子和祖先-子孙的关系。查询模式匹配通过:(1)根据 XML 文件匹配每一个二元结构基本关系;(2)集合返回这些基本匹配关系。笔者分析了 Stack-Tree-Desc 算法和 Stack-Tree-Anc 算法,并且提出一种改进的 Stack-Tree 连接算法,实现了 Stack-Tree-

Desc 算法和 Stack-Tree-Anc 算法的统一,并且比起 Stack-Tree-Anc 算法大大节省了存储空间,提高了查询效率。

### 1 节点元素数据结构描述

在文献[2~4]中提出对 XML 文件中的元素采用位置编码方式,是目前树节点最有效的编码方式。XML 文件中的每一个节点采用五元组表示法(docID, StartPos, endPos, levelNum, treeNodeName):docID 表示 XML 文件编号;startPos 表示该元素在 XML 文件的起始位置;endPos 表示该元素在 XML 文件的结束位置;levelNum 表示该元素节点在 XML 文件按树型结构中的深度;treeNodeName 表示该元素节点名字。采用这样编码方式可以容易确定任意两个树型节点对的基本关系:

(1) 祖先-子孙关系:用五元组表示法假设节点  $n_1(D_1, S_1, E_1, L_1, N_1)$ ,  $n_2(D_2, S_2, E_2, L_2, N_2)$ , 如果  $D_1 = D_2$  和  $S_1 < S_2$  而且  $E_1 > E_2$  则  $n_1$  是  $n_2$  祖先。

(2) 父亲-孩子关系:用五元组表示法假设节点  $n_1(D_1, S_1, E_1, L_1, N_1)$ ,  $n_2(D_2, S_2, E_2, L_2, N_2)$ , 如果

收稿日期:2006-09-04

作者简介:蒋 科(1980-),男,广东增城人,硕士研究生,研究方向为软件工程;郑有才,副教授,研究方向为软件工程。



$D_1 = D_2$  和  $S_1 < S_2$  和  $E_1 > E_2$  而且  $L_2 = L_1 + 1$  则  $n_1$  是  $n_2$  父亲。

这样可以很容易地确定同一个 XML 文件中任意两个元素的基本关系,为 XML Query 查询结构连接算法实现提供了良好的数据结构。

## 2 Stack-Tree 连接算法的输入输出

在文献[2,3]提出 Stack-Tree 连接算法输入对象是 AList  $[a_1, a_2, a_3, \dots]$  和 DList  $[d_1, d_2, d_3, \dots]$ , 分别表示 XML 树中谓词匹配祖先-子孙关系连接  $(a_i, d_i)$  节点的集合。 $a_i, d_i$  都是节点的编码元素,而 AList, DList 中的元素分别按  $(docID, startPos)$  从小到大顺序排列。这里算法分析都是假设 AList, DList 都是已经得到的有序的输入队列。

Stack-Tree-Anc 连接算法输出 outputList  $= [(a_i, d_j), \dots]$ 。 $a_i \in AList, d_j \in DList, a_i.startPos < a_{i+1}.startPos (i = 1, 2, 3, \dots)$ ; Stack-Tree-Desc 连接算法输出 outputList  $= [(d_j, a_i), \dots]$ 。 $a_i \in AList, d_j \in DList, d_j.startPos < d_{j+1}.startPos (j = 1, 2, 3, \dots)$ , 这里为了方便算法的描述,假设所有节点的元素的 docID 都相等,而且 AList, DList 做为输入对象也是已经得到的有序的输出队列。

## 3 Stack-Tree 连接算法描述

在文献[3]中提出 Stack-Tree 连接算法的主要思想:当发现 AList 中的当前节点是当前栈顶节点的子孙时,堆栈 AList 中此节点。堆栈始终是祖先节点的序列,在栈中的每一个节点是它栈底元素的子孙。这时,如果 DList 中当前节点是栈顶元素的子孙,它就是栈中所有堆栈元素的子孙节点。而如果 AList 中下一个元素不是栈顶元素的子孙则再没有元素是当前栈顶元素的子孙,也可以保证 DList 中当前节点不是栈外其他的节点的子孙。

Stack-Tree-Desc 算法是指 outputList 的输出序列按照 DList 中元素的顺序输出所有的基本关系序列  $(d_j, a_i) (i = 1, 2, 3, \dots)$ 。而所谓 Stack-Tree-Anc 算法的 outputList 的输出序列按照 AList 中元素的顺序输出所有的基本关系序列  $(a_i, d_j) (j = 1, 2, 3, \dots)$ 。

### 3.1 Stack-Tree-Anc 算法

根据 Stack-Tree 连接算法的主要思想,实现 Stack-Tree-Desc 算法,outputList 输出不需要附加的数据结构。而在文献[3]中提出 Stack-Tree-Anc 算法要求先输出栈底元素基本匹配模式,依次输出上层元素。而栈数据结构的特点是 FILO,为了实现 Stack

-Tree-Anc 算法,为堆栈中的每一个元素附加两个队列<sup>[3]</sup>:第一个叫做 self-list,用来存储和此元素存在基本匹配关系的 DList 中的元素;第二个叫做 inherit-list,用来记录此元素节点在 AList 队列中的子孙节点元素序列。有了 self-list 队列,当发现 DList 中的一个新节点是当前栈顶元素的子孙节点,就把它加入 self-list 队列,DList 指向下一个元素,接下来 DList 现在当前元素不是此时栈顶元素的子孙节点,那么可以确定 DList 没有其他元素是当前栈顶元素的子孙,那么 pop() 栈顶元素。在出栈之前,判断如果此元素不是栈底元素,那么将此元素的 inherit-list 队列 append() 附加到 self-list 队列后边。出栈后,将出栈元素的 self-list 队列 append() 当前栈顶元素的 inherit-list 队列中。按上述步骤不断操作出栈元素,直到栈底元素出栈后,将 inherit-list 队列 append() 附加到输出队列 outputList。这里每个堆栈元素不断使用自己的 self-list 队列和 inherit-list 队列的操作,从时间和空间上耗费很大。

### 3.2 改进的 Stack-Tree 连接算法

改进算法保持了 Stack-Tree 连接算法主要思想,引入一个整体堆栈、向量和队列数据结构实现了两种算法的统一,不用像 Stack-Tree-Anc 算法为每个堆栈元素附加两个队列,消耗大量存储空间且对出栈元素操作的时间和空间耗费都小于 Stack-Tree-Anc 算法。而且改进算法的实现采用 C++ STL(应用标准模板库)为数据操作采用动态方法产生、回收存储空间,大大节省了存储空间。改进算法描述如下:

```
vector<struct treeNode> v;
stack<struct treeNode> treeNodeStack;
list<struct a_treeNodesRelationship> reList;
a = AList->first; d = DList->first;
While(! AList.empty() || ! DList.empty() || ! treeNodeStack.empty()) {
    If(d.is a descendant of a) {
        treeNodeStack.push(a);
        a = a->nextNode;
    }
    else {
        if(! treeNodeStack.empty()) {
            while(d != NULL) {
                if(d is a descendant of treeNodeStack.top()) {
                    v.push_back(d); d = d->nextNode;
                }
                else { //这里将两种输出方式放在一起表示,实际程序只需用其中之一
                    d_outputList(treeNodeStack, v, reList);
                    a_outputList(treeNodeStack, v, reList);
                }
            }
        }
        else { if(! reList.empty()) {
            while(! reList.empty()) {
```



```

reList.front =>outputList;
reList.pop_front();
for(int i=0;i<v.size();i++)
v.pop_back();
else {move to next element from AList or
DList;}}}}

```

以上改进算法实现了 Stack - Tree 连接算法的统一。如果按 DList 元素顺序输出则调用函数 d\_outputList 函数;如果按 AList 元素顺序输出则调用 a\_outputList 函数。函数描述如下:

```

a_outputList (stack< struct treeNode> & treeNodeStack,vector< struct treeNode> & v, list< struct
treeNodesRelationship> & reList){
    treeNodeStack.pop()=>tempTreeNode;
    for(int i=v.size()-1;i>=0;i--)

```

```

        将 tempTreeNode 和 v[i]=>reList.front();
d_outputList(stack< struct treeNode> & treeNodeStack,vector<
struct treeNode> & v, list< struct treeNodesRelationship> &
reList){
    依次将 v[v.size()-1]和 treeNodeStack 从栈顶到栈底元素=
    >reList.front();
    treeNodeStack.pop()=>tempTreeNode;
    while(v[v.size()-1] is not descendant of treeNodeStack.top
    ()){
        v.pop_back();}}

```

这里假设  $a, d$  指向 AList,DList 中的元素,讨论算法执行情况:循环判断  $d$  是不是  $a$  指向元素的子孙,是则  $a$  堆栈, $a$  指向 AList 下一个元素;若  $d$  不是,但  $d$  是栈顶元素的子孙,则将  $d$  插入到向量  $v$ , $d$  指向下一个元素,再循环判断  $d$  是不是栈顶元素的子孙,是则  $d$  插入  $v$ ;如果不是,可以确定 DList 没有其他元素是栈顶元素的子孙,则将匹配基本模式序列插入 reList,有如下情况:

(1)按 AList 中元素顺序输出,即调用函数 a\_outputList。首先 top()栈顶元素,然后依次将其与向量元素从尾部到头部形成匹配序列插入 reList 队列。如图 1 所示。

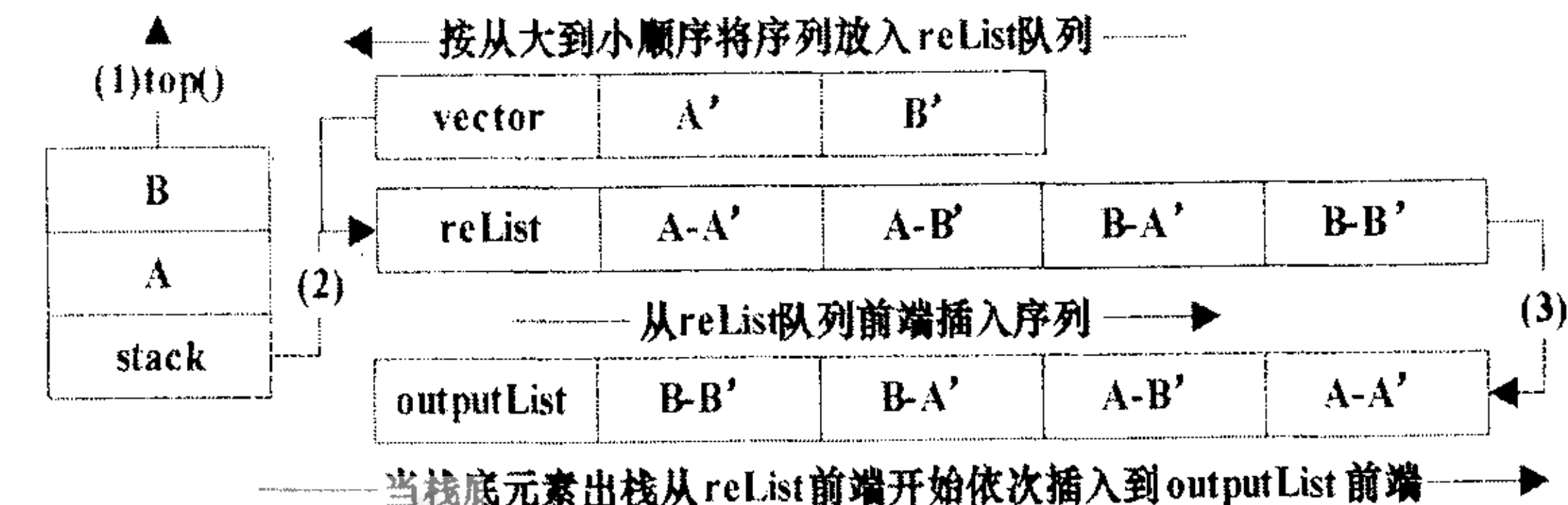


图 1 a\_outputList 匹配序列输出步骤

(2)按 DList 中元素顺序输出,即调用函数 d\_outputList。首先依次将  $v[v.size() - 1]$  元素和 treeNodeStack

odeStack 栈顶到栈底元素形成匹配序列插入 reList 队列,然后出栈栈顶元素,最后循环判断  $v[v.size() - 1]$  元素不是当前栈顶元素的子孙,则 delete  $v[v.size() - 1]$  元素。如图 2 所示。

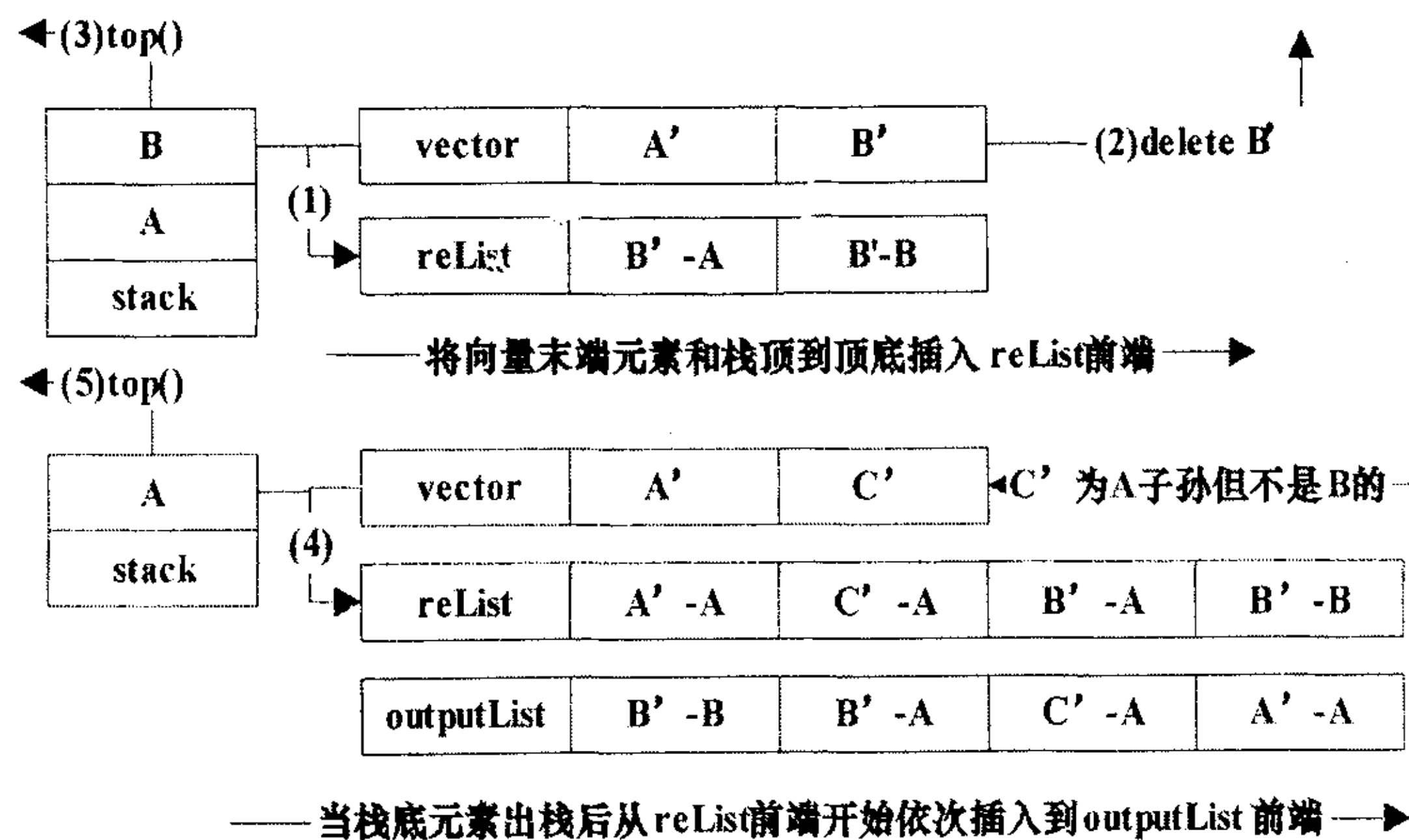


图 2 d\_outputList 匹配序列输出步骤

这里图 1,2 描述的步骤是最常规的步骤,由于篇幅有限有些特殊情况在这里不再详细介绍。

## 4 算法性能和试验结果

这里分析讨论改进算法的最坏算法性能的情况,如图 3 所示,向量  $v$  和 reList 队列最多需要  $2n$  个存储空间,时间复杂度为: $2 \times (2n + (2n - 2) + \dots + 2) = 2n(n + 1)$ ,即  $o(n^2)$ 。这里的时间复杂度主要消耗在输出匹配序列上,匹配过程的时间复杂度为  $o(n)$ 。为了测试该算法的性能,实验室环境为 Pentium4 2.93GHz,Red Hat Enterprise Linux 4 操作系统。抽取 XML 文件内容 DTD 文件、试验数据和试验结果如下:

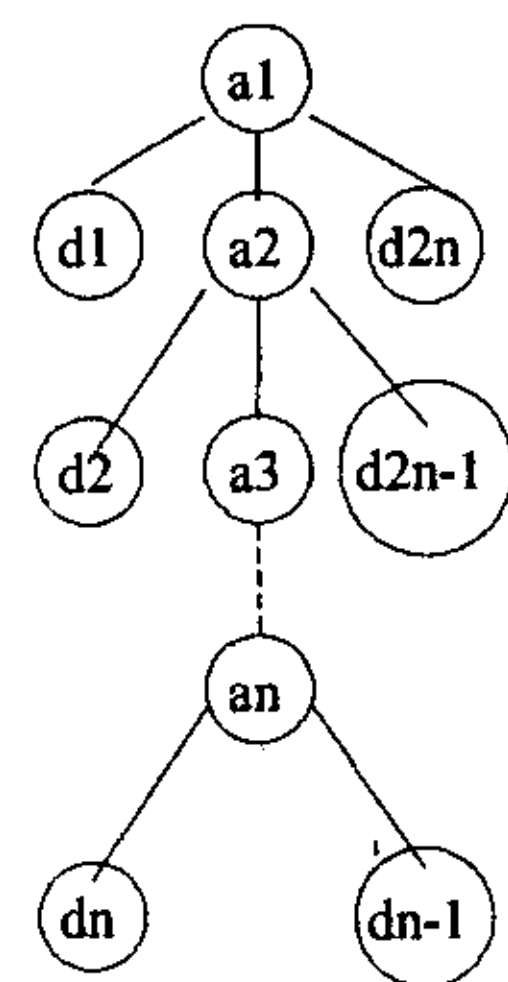


图 3 改进算法的最坏情况

```

<! ELEMENT manager (name, (manager | de-
partment | employee) + )>
<! ATTLIST manager id CDATA #FIXED "1"
>
<! ELEMENT department (name, employee + ,
department * )>
<! ATTLIST department id CDATA #FIXED "2">
<! ELEMENT employee (name + )>

```



```
<! ATTLIST employee id CDATA #FIXED "3">
<! ELEMENT name (#PCDATA)>
<! ATTLIST name id CDATA #FIXED "4">
```

试验结果表明(见表 1),在进行简单查询时使用 Stack-Tree-Desc 算法效率最好;进行复杂查询的时候,改进的 Stack-Tree-Anc 算法效率明显提高,且在任何查询情况下改进 Stack-Tree-Anc 算法都略好于未改进的。

表 1 试验数据和试验结果

| Node       | Count | Query | XQuery Path Expression | Results |
|------------|-------|-------|------------------------|---------|
| manager    | 10    | QS1   | employee/name          | 2368    |
| department | 25    | QS2   | employee/name          | 3450    |
| employee   | 1500  | QC1   | manager/employee/name  | 1678    |
| name       | 5000  | QC2   | manager/employee/name  | 4643    |

| 查询表达式                   | QS1   | QS2   | QC1   | QC2   |
|-------------------------|-------|-------|-------|-------|
| Tree-Stack-Desc 响应时间    | 267ms | 304ms | 416ms | 462ms |
| Tree-Stack-Anc 响应时间     | 312ms | 327ms | 523ms | 502ms |
| 改进 Tree-Stack-Desc 响应时间 | 285ms | 317ms | 446ms | 480ms |
| 改进 Tree-Stack-Anc 响应时间  | 284ms | 309ms | 467ms | 454ms |

## 5 结束语

讨论了 Stack-Tree 连接的相关算法,并提供一种改进的 Stack-Tree 算法,引进新的数据结构将 Stack-Tree-Anc 算法和 Stack-Tree-Desc 算法统一,且使用 C++ STL,动态产生、回收存储空间,能够大大

(上接第 86 页)

时执行,这一设计思想体现了系统的交互性。在即时执行过程中,单个检测方法或单个清洗策略的执行效果可以立即呈现给用户,由用户决定是否需要修改检测方法和调整清洗策略,这样做可以提高准确度。批量执行可以提高效率,但比较难把握准确度,即时执行和批量执行要根据实际情况取得平衡。它们的交互式定义和执行流程如图 2 所示。

## 3 实验结果与结论

本系统运行在 P4 微机上,实验数据来源于某市人口信息系统,540 多万条人口数据。检测和清理重复记录 2347 条,用时 13 分钟,而传统手段耗时 1 个多小时,这说明效率大幅度提高。清洗系统查到错误姓名达到 8971 条,通过多方核对,确实存在错误是 8723 条,准确率达到  $8723/8971 = 97.2\%$ ,并且从公民身份号码重号清洗等也得到相似结果。从实验结果可以得知,此模型在执行效率上相当高,效果相当好,可操作性强。

节省存储空间。但是由于 Stack-Tree 连接算法不能忽略操作一些不存在匹配序列的 AList 和 DList 的元素,要进一步改进或引进新的数据结构能够动态地忽略这些多余的元素,文献[5]提出引进 B+-树和 R-树的数据结构,进一步提高 XMLQuery 查询中结构连接的基本模式匹配的性能。

### 参考文献:

- [1] Li O, Moon B. Indexing and Querying XML Data for Regular Path Expressions[C]// Apers P M G, Atzeni P. In: Proc. of VLDB 27. Roma, Italy: [s. n.], 2001: 217-228.
- [2] Zhang C, Naughton J, Dewitt D, et al. On Supporting Containment Queries in Relational Database Management Systems [C]// Aref W G. In: Proc. of SIGMOD. Santa Barbara, California: [s. n.], 2001: 425-436.
- [3] Al-Khalifa S, Jagadish H V, Koudas N, et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching [C]// Johnsen B. In: Proc. of ICDE. Oslo, Norway: [s. n.], 2002: 141-152.
- [4] Chien S Y, Tsotras C. Storing and Querying Multiversion XML Document using Durable Node Numbers[C]// In: Proc. of WISE. California, USA: National Science Foundation, 2001: 323-336.
- [5] Chien S Y, Vagena Z, Zhang D, et al. Efficient Structural Joins on Indexed XML Document[C]// In: Proc. of VLDB 28. Hong Kong, China: Morgan Kaufmann, 2002: 263-274.

### 参考文献:

- [1] 陈文伟. 数据仓库与数据挖掘教程[M]. 北京: 清华大学出版社, 2006: 6-94.
- [2] Galhardas H, Florescu D, Shasha D, et al. Declarative data cleaning: language, model and algorithms[C]// In: Apers P, Atzeni P, Ceri S, et al. Proceedings of the 27th International Conference on Very Large Data Bases. Roma: Morgan Kaufmann, 2001: 371-380.
- [3] Kantardzic M. 数据挖掘——概念、模型、方法和算法[M]. 闪四清, 陈茵, 程雁, 等译. 北京: 清华大学出版社, 2003: 144-154.
- [4] Hernandez M A, Stolfo S J. The merge/purge problem for large databases[C]// In: Carey M J, Schneider D A. Proceeding of the ACM SIGMOD International Conference on Management of Data. San Jose: ACM Press, 1995: 127-138.
- [5] Lee M L, Ling T W, Low W L. IntelliClean: a knowledge-based intelligent data cleaner[C]// In: Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Boston: ACM Press, 2000: 290-294.