第 35 卷 第 4 期 计 算 机 工 程 2009 年 2 月
Vol.35 No.4 Computer Engineering February 2009
·软件技术与数据库· 文章编号：1000—3428(2009)04—0035—03 文献标识码：A 中图分类号：TP311.12

XML 流上的关键字查询算法

李 波，杨卫东
(复旦大学计算机与信息技术系，上海 200433)

摘 要：针对当前 XML 流过滤研究中存在的问题，使用关键字查询方法作为解决方案。提出最右包含边界的概念，结合一个虚拟栈实现用于在 XML 数据流上进行关键字查询的 XVirtualStack 算法。理论分析和实验结果证明，该算法具有高效性。
关键词：关键字查询；XML 流过滤；最右包含边界

Algorithm for Keyword Search over XML Stream

LI Bo, YANG Wei-dong
(Department of Computer and Information Technology, Fudan University, Shanghai 200433)

【Abstract】 This paper points out several problems of current researches on XML stream filtering, and uses keyword search method as a solution. Based on the conception of Rightmost Contain Border(RCB), it implements an algorithm called XVirtualStack via using a virtual stack for keyword search over XMLstream. The theoretical analysis and experimental results validate that this algorithm is efficient.
【Key words】 keyword search; XML stream filtering; Rightmost Contain Border(RCB)

1 概述

XML 逐渐成为互联网上数据存储和交换的标准，很多与 XML 数据流相关的应用大量出现，如股票交易信息、实时新闻的订阅和发布等。在上述应用中，用户要求以查询的方式存放在服务器中。当 XML 数据流到达服务器时，服务器中的过滤引擎根据用户注册的查询将符合用户要求的数据返回给用户。

现有研究主要针对基于 XPath/XQuery 的查询，该方式存在 2 个问题：(1)每个用户都需要掌握一门复杂的查询语言，即对用户是不友好的，且对初级用户来说是不现实的。(2)每个用户需要知道流的结构才能写出相应查询，增加了用户负担，且在一些情况下，由于用户无法得到流的结构，因此无法写出查询。例如，一个文档是结构良好的但不是有效的，即无 DTD 可遵循。

使用关键字查询可以解决上述问题，关键字查询对用户是友好的。

2 XML 流上关键字查询的相关工作

当前在 XML 流上进行关键字查询的相关工作主要包括以下 2 个方面：(1)在 XML 流上进行 XPath/XQuery 查询[1]；(2)在 XML 文档上进行关键字查询[2-4]。文献[2]主要讨论在 XML 树上快速找到包含关键字的 LCA(Lowest Common Ancestor)节点集合，并提到了如何对结果进行等级划分的方法。文献[3-4]主要解决如何快速找到 SLCA(Smallest LCA)节点集合。

文献[5]提出在 XML 流上进行关键字查询，并给出一个基于栈的 lookup 算法，将最小相关连通子树作为结果返回。本文提出最右包含边界(Rightmost Contain Border, RCB)概念，并在此基础上使用一个虚拟栈实现一个高效算法，即 XVirtualStack，它将 SLCA 节点集合作为返回结果。

一个 XML 文档如下：

```
<a>
    <b>w1 k1 </b>
    <c>
        <d>k2 w2 </d>
        <e>
            <f>k3 k1 </f>
        </e>
    </c>
</a>
```

上述 XML 文档的树结构如图 1 所示。树的每个内部节点都有一个标记与之对应，每个叶子节点由一串词组成。查询 $Q=\{k1,k2,\cdots,kn\}$是词的集合。将出现在 Q 中的词称为关键字。对于一个查询 Q，返回 Q 在该文档上的 SLCA 节点集合。SLCA 的定义见文献[3,5]。例如，对于上述文档，如果 $Q=\{k3,w2\}$，就返回集合$\{c\}$；如果 $Q=\{k1\}$，则返回$\{b,f\}$。

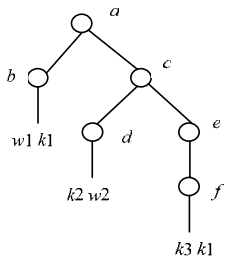

图 1 XML 文档的树结构

3 最右包含边界

定义 设集合 $Q=\{k1,k2,\cdots,kn\}$，且对任意的 $i\neq j$，有 $ki\neq kj$；$T=<t0,t1,\cdots,tm\text{-}1>$为一个有序序列，且对任意 $i\in[0,m\text{-}1]$，有 $ti\in Q$。T 在 Q 上的最右包含边界 $RCB(T,Q)$被定义为一个整数 p，p 满足以下 3 个条件：

(1)$p\in[0,m\text{-}1]$。

作者简介：李 波(1983－)，男，硕士研究生，主研方向：XML 数据库及流处理；杨卫东，副教授
收稿日期：2008-07-10 **E-mail**：052021152@fudan.edu.cn

(2)对于任意 $k \in Q$，存在一个 $i \in [p,m\text{-}1]$，且 $ti=k$。

(3)对于任意 $q \in (p,m\text{-}1]$，存在一个 $k \in Q$，且对任意 $i \in [q, m\text{-}1]$，有 $ti \neq k$。

从最右包含边界的定义可以看出，RCB(T,Q)标明的是 T 中包含 Q 所有元素的最右部分。例如，若 $T=<a,c,g,a,a,c,a,d,g,d>$, $Q=\{a,c,g,d\}$，则 $RCB(T,Q)=4$。T 中的元素下标从 0 开始。

4 XVirtualStack 算法

4.1 初始算法

对流进来的 XML 数据一般使用 SAX 方式进行解析，相当于对 XML 树进行前序遍历。本文在初始算法中使用 2 个栈：(1)运行时栈 RTstack，保存对 XML 树进行前序遍历过程中的当前路径；(2)用于缓存关键字的栈 KWstack。RTstack 中的每个元素都是一个二元组(tag,pos)，其中，tag 表示 XML 树中节点的标记；pos 对应该节点进入 RTstack 时，KWstack 的栈顶位置。图 2 给出了节点 v 进入 RTstack 和准备从 RTstack 退出时，2 个栈的状态。当 v 准备出栈时，KWstack 中从 v.pos 到 KWstack.top 这段栈中保存的关键字就是出现在以 v 为根的子树中的所有关键字。此时可以查看该段栈的内容，以确定查询中的所有关键字是否都出现在其中。如果是，且 v 的任何子孙都不是 slca 节点，则 v 是该查询的一个 slca 节点。因为一个 slca 节点的祖先不可能是 slca 节点，所以应该对 v 的所有祖先节点做标记。由于 v 的所有祖先都在栈中，因此本文通过将其 pos 值改为“×”实现对它们的标记。

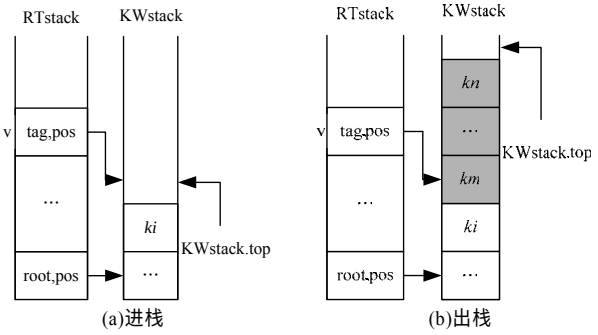

图 2 节点 v 进栈和出栈时 2 个栈的状态

由于初始算法在时间和空间上效率较低，因此需要用一个多余的栈 KWstack 来缓存遇到的关键字，当一个节点出栈时，还需要扫描 KWstack 的一部分以确定是否所有关键字都出现了。

4.2 改进算法

将 KWstack 中缓存的关键字看成是一个有序序列 T，T 的第 1 个元素为 KWstack 的最底元素。对查询 $Q=\{k1,k2,\cdots,kn\}$，计算 T 在 Q 上的最右包含边界 $RCB(T,Q)$。如果以 v 为根节点的子树包含 Q 的所有关键字，则 v 刚进栈时，KWstack 的栈顶 v.pos 一定要小于或等于 v 将要出栈时 T 的最右包含边界。反之，如果以 v 为根节点的子树不包含 Q 的所有关键字，那么 v.pos 一定要大于 v 将要出栈时 T 的最右包含边界。

根据上述结论，可以设置一个向量 ***NPV***，其大小为$|Q|$，即 Q 中关键字的数目。***NPV*** 的每一项都是一个整数，与 Q 中的每个关键字对应，并记录该关键字在 KWstack 中的最新位置，其初始值为-1。指针 p_RCB 指向 ***NPV*** 中的一个元素，该元素保存了当前 KWstack 的最右包含边界。因为 ***NPV*** 中的最小元素记录了当前 KWstack 的最右包含边界，所以 p_RCB 一定指向 ***NPV*** 中最小的一个元素。初始时，p_RCB 指向任意一个元素。当一个关键字进入 KWstack 栈后，用它在 KWstack 中的位置更新它在 ***NPV*** 中对应的项。如果 p_RCB 指向的元素被更新，则该 p_RCB 不再指向 ***NPV*** 的最小值，此时应扫描 ***NPV***，以便让 p_RCB 指向新的最小值。

因为有 ***NPV***，所以无须使用 KWstack，只要假设它存在，因此，称 KWstack 为虚拟栈。在 ***NPV*** 中记录每个关键字在 KWstack 中的最新位置即可。当一个节点 v 打算出栈时，将 v.pos 与虚拟栈 KWstack 的当前最右包含边界值(由 p_RCB 指向)对比。如果 v.pos 小于或等于该值，则以 v 为根的子树包含 Q 中所有关键字。反之，则以 v 为根的子树不包含 Q 中所有关键字。

改进算法的伪代码如下：

```
输入 XML stream, Q={k1,k2,…,kn}
输出 Q 的所有 slca 节点
startDocument()
  初始化 RTstack;
  top=0,p_RCB=rand(0,|Q|-1);
  NPV[0…|Q|-1]=-1;
startElement(String tag)
  RTstack.push(tag,top);
endElement()
  弹出 RTstack 的栈顶并存于 sn 中;
  如果 sn.pos 为“×”，则返回;
  如果 sn.pos 小于等于 NPV[p_RCB]，则
        将 sn.tag 作为 slca 节点输出;
        将 RTstack 中所有元素的 pos 域标为“×”;
character(buffer B)
对 B 中的每一个词 w
      如果 w 对应于 Q 中的第 i 个关键字
            NPV[i]=top;
            top=top+1;
            如果 i 等于 p_RCB
                  扫描 NPV，让 p_RCB 指向最小值
endDocument()
清空 RTstack;
```

检查一个节点出栈时是否被标记为“×”，如果是，则该节点是一个 slca 节点的祖先，因此不是 slca 节点。伪代码第 11 行完成上述检查。当一个节点被确认为 slca 节点后，伪代码第 13 行-第 14 行将其所有祖先标记为“×”。在 character() 处理函数中，对于缓存中的每个词，都要查看它对应哪个关键字，并更新它在 ***NPV*** 中的值。若 ***NPV*** 中保存原先最右包含边界的项被更新，则需要重新找到 RCB，如伪代码第 20 行第 21 行所示。

图 3 描述了当查询 $Q=\{k1,k2,k3\}$时，算法在第 2 节所述文档上的处理过程。初始化时，RTstack 为空；***NPV*** 的各项为-1；p_RCB 为 0~2 之间的一个随机整数；top 为 0。具体说明如下：在步骤(1)中，当 a 和 b 进栈时，因为没有遇到关键字，所以其他变量都保持不变。当关键字 $k1$ 被识别后，***NPV***[0]被更新为当前 top 的值，然后 top 被加 1，指向虚拟栈的新栈顶，如步骤(2)所示。在步骤(4)中，当 d 准备出栈时，d.pos 等于 1，而此时 RCB 为 NPV 中的最小值-1，d.pos 大于 RCB，因此，d 不是一个 slca 节点，此时 d 直接出栈即可。当步骤(6)中的 f 和 e 准备出栈时，他们和 d 都属于类似的情况。在步骤(7)中，当 c 准备出栈时，c.pos 等于 1，而此时 RCB 为 1，即 c.pos 小于等于 RCB，因此，c 是一个 slca 节点。c 被作为一个结果输出，然后将 RTstack 中所有项的 pos 域标记为“×”。在步骤(8)中，当 a 准备出栈时，发现其 pos 域被

标记为“×”，说明 a 是一个 slca 节点的祖先，a 直接出栈。此时 RTstack 栈为空，算法执行结束。得到了查询 Q 在该文档上的 slca 节点集，即$\{c\}$。

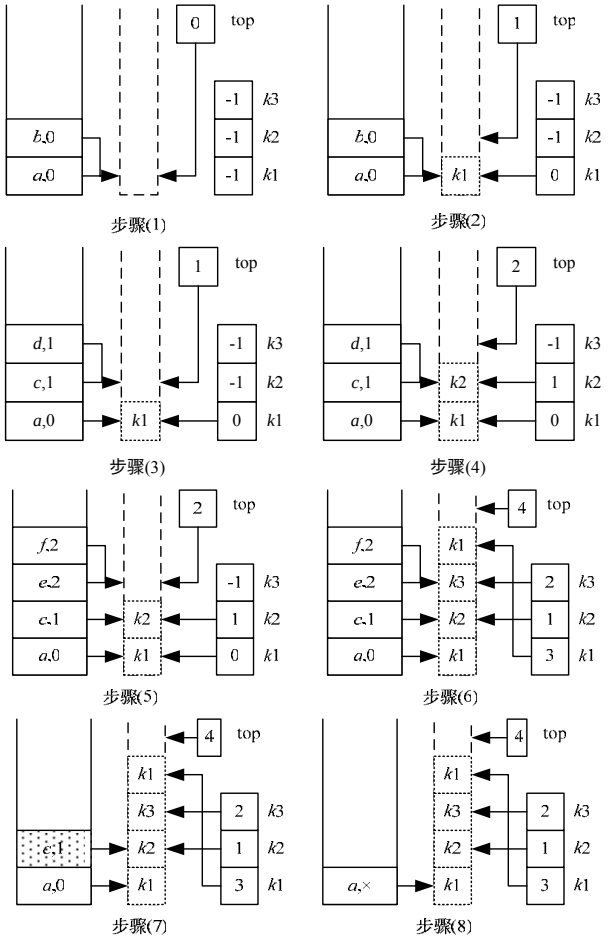

图 3　算法运行示例

5　性能分析与实验结果

假设查询为 Q，其关键字个数为 k；文档 T 有 N 个内部节点，所有叶子节点共含有 W 个词。可以将查询 Q 中的每个关键字放入一个 hash 表。hash 表中的关键字域为查询中的关键字，值域为该关键字在 Q 中的索引。对于流中的每个词，只要 $O(1)$时间就可以知道它是否是 Q 中的关键字以及是哪个关键字。对于文档中的每个节点，其进栈和出栈次数均为 1，且最多有一次被标为“×”，因此，以上操作总的时间代价为 $O(N)$。假设查询中的每个关键字在 T 中出现的概率相等，则改进算法伪代码的第 20 行-第 21 行只有 $1/k$ 的概率被执行。因为第 21 行的代价为 $O(k)$，所以第 20 行-第 21 行的平均代价为 $O(1)$，因此，character()操作的总时间代价为 $O(W)$。综上所述，XVirtualStack 算法的时间代价为 $O(N+W)$，即只要对流进行一次扫描，就能得到查询 Q 的所有 slca 节点。对于文献[5]中的 lookup 算法，连同节点压栈的是一个大小为 $O(k)$ 的标志数组，数组中的每个位在必要时都要被更新，因此，lookup 算法的时间复杂度为 $O(kN+W)$。当 $W=O(N)$时，XVirtualStack 的性能优于 lookup。

从空间复杂度上看，由于 lookup 算法每次压栈的是一个大小为 $O(k)$的标志数组，因此其空间复杂度为 $O(kN)$。而 XVirtualStack 每次压栈的是固定大小的空间，且除了栈空间外，仅用到大小为 $O(k)$的 ***NPV*** 向量和 hash 表，因此，XVirtualStack 的空间复杂度为 $O(k+N)$。

由上述分析可以看出，在时间和空间上，XVirtualStack 都有具有较好性能。

选择 XMark 作为实验数据集，并用 xmlgen 生成大小分别为 1 MB, 2 MB, 3 MB, 4 MB, 5 MB 的 5 个文件。算法的实现语言为 Java，CPU 为 AMD 双核 4000+，1 GB RAM，操作系统为 Windows XP。实验结果如图 4 和图 5 所示。由图 4 可以看出，当查询的关键字个数确定时，lookup 算法和 XVirtualStack 算法的时间复杂度成线性增长，但 lookup 的增长速度较快。由图 5 可以看出，XVirtualStack 的时间复杂度与关键字个数无关，而 lookup 算法的时间复杂度与关键字个数呈正比。实验结果证实上述理论分析。

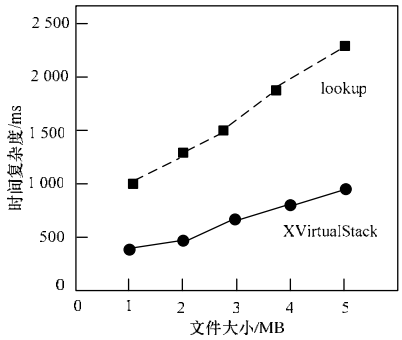

图 4　关键字数为 4 时 2 种算法的时间复杂度

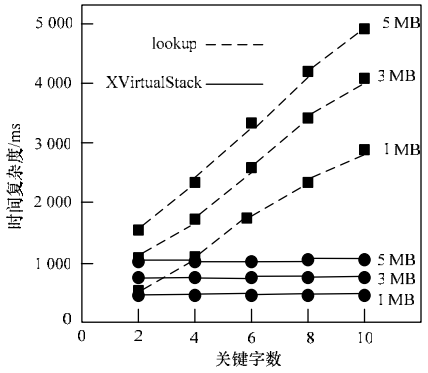

图 5　不同关键字数时 2 种算法的时间复杂度

6　结束语

本文总结了当前 XML 流研究单纯使用 XPath/XQuery 作为查询语言带来的问题以及使用关键字查询的优点。提出 XVirtualStack 算法并与 lookup 算法进行实验对比，结果表明本文算法效率较高。

参考文献

[1] Diao Yanlei, Altinel M, Franlin M, et al. Path Sharing and Predicate Evaluation for High-performance XML Filtering[J]. ACM Trans. on Database System, 2003, 28(4): 467-516.

[2] Guo Lin, Shao Feng, Botev C. XRANK: Ranked Keyword Search over XML Documents[C]//Proceedings of the 22nd ACM SIGMOD Conference. San Diego, California, USA: ACM Press, 2003: 16-27.

[3] Xu Yu, Papakonstantinou Y. Efficient Keyword Search for Smallest LCAs in XML Databases[C]//Proc. of the 24th ACM SIGMOD Conference. Baltimore, Maryland, USA: ACM Press, 2005: 527-538.

[4] Li Yunyao, Yu Cong, Jagadish H V. Schema-free XQuery[C]// Proceedings of VLDB. Toronto, Canada: [s. n.], 2004: 72-83.

[5] 王小峰, 张　新, 谢　敏, 等. XML 数据流上的关键字查询[J]. 计算机研究与发展, 2006, 43(增刊): 464-470.