

XML Query Technology with Positional Predicates over Streams

Chenyang Wang¹, Husheng Liao², Hang Su¹

1. College of Computer Sciences, Beijing University of Technology, Beijing 100124, China

2. College of Software Engineering, Beijing University of Technology, Beijing 100124, China

#Email: eagle_james@163.com; liaohs@bjut.edu.cn; suhang@bjut.edu.cn

Abstract

Twig pattern matching is a core operation in XPath query processing. In the field of XML streaming data processing, holistic twig pattern matching algorithms have been studied widely nowadays. However, little attention is paid on positional predicates defined on navigation axis in XPath expressions. Due to the lack of context information, these algorithms cannot filter out the final solutions satisfying the positional predicates from the results of tree pattern querying. To fill this technology vacancy, in this paper, we propose a new holistic twig pattern matching algorithm PosLQ for XML stream data to implement the XPath queries with the positional predicates. Experimental results show that our algorithm can efficiently tackle these types of XPath queries, which is practical in the field of XML data stream.

Keywords: XML; XPath; Stream; Tree Pattern Query; Positional Predicate

支持位置谓词的 XML 流数据查询技术*

王晨阳¹, 廖湖声², 苏航¹

1. 北京工业大学, 计算机学院, 北京 100124

2. 北京工业大学, 软件学院, 北京 100124

摘要: 树模式查询是 XPath 查询语言的核心操作, 特别是采用整体匹配策略的树模式查询算法也已得到了广泛的研究。但是在 XML 流式数据处理领域中, 对于含有位置谓词的 XPath 查询计划的处理, 现有的整体匹配算法均未予以考虑。由于必要上下文信息的缺失, 现有算法无法从树模式查询的返回解中过筛选出满足位置谓词约束的最终解。为了能够使树模式查询算法支持位置谓词的处理, 本文提出了一种新的针对 XML 流数据的整体匹配算法 PosLQ。实验结果表明, 该算法可高效地处理上述类型的 XPath 查询请求, 增强了树模式查询技术在 XML 流数据领域的实用性。

关键词: XML; XPath; 流数据; 树模式查询; 位置谓词

引言

XML 作为一种半结构化的数据描述语言, 已成为互联网上广泛应用的数据表示方式和传输格式的工业界标准。如何对 XML 数据进行高效地查询也成为数据库领域的研究热点。随着 XML 被应用于 Web 数据交换等各个领域, “XML 流式数据^[1]”受到了研究者们普遍关注, 给 XML 查询处理提出了新的需求和挑战。流式数据通常具有如下四个基本特点^[2]: 数据在线到达, 处理端对数据到达的顺序无法控制, 数据量的大小可能没有边界, 处理完某个数据元素便将它转发或丢弃。基于上述特点可知, 现有的大部分面向传统 XML 数据库的查询机制将不再适用于流式 XML 数据。因此, 研究专门针对 XML 流数据的查询机制也就变得尤为重要。

* 基金资助: 受北京市自然科学基金项目 (NO.4122011)、国家自然科学基金青年基金项目 (NO.61202074) 支持资助。

针对 XML 数据的查询具有一个共同的核心操作, 即从 XML 数据源中抽取查询所感兴趣的片段(一个或多个节点)——通常人们用“树模式”(Twig)表示查询感兴趣的节点以及他们之间的关系(祖先后代、双亲子女等)^[3]。这一从数据源中抽取所需节点片段的过程就是树模式查询的处理过程。树模式查询处理性能的优劣很大程度上决定了整个 XML 查询处理效率的高低。

在 XPath 等 W3C 规定的 XML 查询语言规范中, 都会定义一个用于在当前被处理的项序列中指定某项位置的函数 position()。而 XPath 中的位置谓词可被看作是位置函数的一种特殊表现形式, 用于筛选当前被处理的项序列中满足位置约束的项, 是针对 XML 数据的查询中一种很常见的功能。

对于面向流式 XML 数据的树模式查询, 目前已存在多种算法。Yi Chen 等人提出的 TwigM 算法^[4]、Gang Gou 等人提出的 LQ 系列算法^[5]、Wook-Shin Han 等人提出的 StreamTX 算法^[6]都各具特点, 并且执行效率可以与面向非流式数据的树模式查询算法相媲美。但是, 这些算法均未考虑 XPath 查询中出现位置谓词的情形, 故无法处理含有位置谓词的查询。

对于面向非流式 XML 数据的树模式查询, Fajin Li 等人提出了 TwigPos 算法^[7], 通过一种新型的嵌套栈结构支持了位置谓词的筛选, 不仅扩展了树模式查询算法的支持范围, 同时也保证了树模式查询的执行效率。但是, TwigPos 算法针对的是非流式 XML 数据, 对于流式 XML 数据, 该算法并不适用。

针对上述问题, 本文研究了在 XML 流数据树模式查询中支持位置谓词处理的方法, 主要贡献如下:

1、提出了一种面向 XML 流数据的整体树模式匹配算法 PosLQ, 支持含有谓词特别是位置谓词的 XPath 查询, 填补了目前 XML 流数据领域无法从树模式整体匹配结果中筛选出符合位置谓词的结果这一技术空缺。

2、基于不同的 XML 基准数据集, 完成了几组相关树模式匹配算法的性能对比实验, 并说明了 PosLQ 算法在扩展了 XML 流数据树模式查询支持范围的同时, 也保证了较高的树模式查询执行效率。

本文组织如下: 第 2 节阐述本文的工作动机; 第 3 节详细介绍了 PosLQ 算法的设计思路, 并给出了具体的 PosLQ 算法流程; 第 4 节介绍本文的实验部分; 第 5、6 节分别介绍相关工作和总结。

1 动机

目前, 针对流式 XML 数据的树模式匹配算法均不能处理含有位置谓词的 XPath 查询。根据 W3C 给出的 XPath 语义规范, 在处理 XPath 查询中的位置谓词时, 需要同时借助上下文中的两部分信息: 当前节点项(context node)和当前节点项所在的项序列(context sequence)。由于在原有的 XML 流数据树模式查询的返回结果中并不具备上述两种上下文信息, 因此也就无法对这些查询结果进行位置谓词满足性的判定。

例如, 对图 1 (a)中所示的 XML 文档进行图 1 (b)中所示的 XPath 查询“//A/B[./C][2]”。根据语义, 该查询意图在给定的 XML 文档中找出第 2 个同时具有 A 双亲和 C 后裔的 B 节点。在查询过程中, 若当前节点为 b2、b3 或 b15, 则当前节点所在的序列为{b2, b3, b15}。对于 b2 和 b15 节点, 由于其不存在 C 后裔, 故均不是所求结果。而对于 b3 节点, 虽然其满足所有的结构约束, 并且是该序列中的第 2 个节点, 但由于该序列中的第 1 个节点 b2 不满足结构约束, 因此 b3 实际上是该序列中满足条件的第 1 个节点。故由于位置谓词的约束, b3 节点亦不是所求。类似地, 若当前节点为 b7 或 b9, 则当前节点所在的序列为{b7, b9}。因 b7 和 b9 均满足结构约束, 且 b9 同时也满足位置谓词约束, 故 b9 是所求。综上, 对图 1 (a) 中所示的 XML 文档进行图 1 (b) 中所示的查询, 应该只得到一个结果 b9。但是, 如果先不考虑该查询计划中的位置谓词, 直接运用现有的针对 XML 流数据的树模式匹配算法进行计算, 那么将会得到 b3、b7、b9 三个节点。继而, 在之后谓词处理中, 由于无法获得这三个节点所在的上下文项序列, 因此无从判断 b3、b7、b9 三个节点是否满足查询中给出的位置谓词约束。所以, 对于含有位置谓词的 XPath 查询, 现有的 XML 流数据树模式匹配算法均不能适用。

综上所述, 若要在树模式查询算法中支持对位置谓词的筛选处理, 就必须对当前节点项和当前节点项所在的项序列两个上下文信息加以保留和维护。因此, 本文意图通过对现有的 XML 流数据树模式匹配算法

进行改造，在算法进行的过程中维护有用的上下文信息，并在适当时机作用于位置谓词的判定，从而达到在树模式匹配算法中支持位置谓词处理的最终目的。

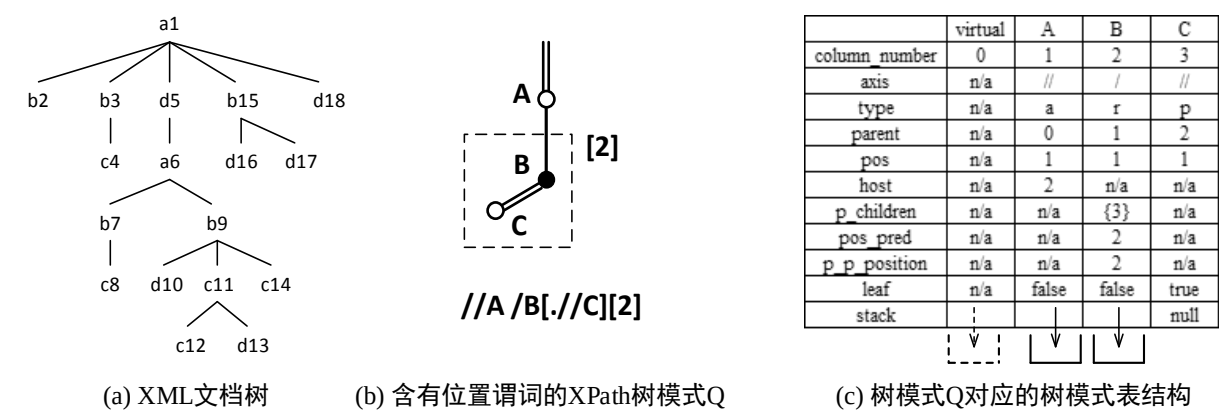


图 1 含有位置谓词的树模式查询举例

2 支持位置谓词的流数据树模式匹配技术

本文基于 LQ 和 TwigPos 两种针对不同领域的高性能树模式匹配算法，将 LQ 算法的“可支持流式数据树模式查询”特点和 TwigPos 算法的“可支持含有位置谓词的树模式查询”特点相结合，提出一种新的针对 XML 流数据的支持位置谓词处理的树模式匹配算法 PosLQ。本节将对 PosLQ 算法进行详细介绍。

2.1 树模式表示方案

在支持位置谓词的 XML 流数据树模式匹配算法 PosLQ 中，对树模式的描述方法在部分延用 LQ 算法树模式表示方案的同时，引入了两个新的整数字段 pos_pred 和 p_p_position，以此来增强原有“表”结构的对于位置谓词的描述能力。PosLQ 算法中表结构包含的各字段如表 1 所示：

表 1 PosLQ 算法中描述树模式的表结构

字段名	字段含义	字段名	字段含义
1 axis	当前树模式节点所在的轴类型 (/, //)	6 p_children	当前树模式节点的所有谓词孩子节点的标号集合
2 type	当前树模式节点的类型 (轴节点、返回节点、谓词节点)	7 leaf	当前树模式节点是否是叶节点的标记
3 parent	当前树模式节点的父节点标号	8 LQStack	LQ核心栈结构
4 pos	当前树模式节点在兄弟间的位置	9 pos_pred	当前节点所需满足的位置谓词
5 host	当前树模式节点所在的PC子片段中最靠近返回节点的节点标号	10 p_p_position	位置谓词在当前节点的各个谓词中出现的位置

表 1 中的前 8 个字段延用了 LQ 算法中使用的“表”结构，每个树模式节点对应一列（需设置一个“虚根”节点），分别以数字进行标识，保证一一对应。表 1 中的第 9 和第 10 个字段是 PosLQ 算法新引入的字段，用以描述查询中的位置谓词信息。其中，pos_pred 字段用以表示当前节点所需满足的位置谓词，p_p_position 字段用以表示该位置谓词在当前节点的各个谓词中出现的位置。

由 W3C 给出的 XPath 规范可知，XPath 表达式中的位置谓词可能出现在若干个存在谓词之间。然而，作为一个查询步节点的位置谓词，其作用范围应该仅限于那些满足在其之前出现的结构约束条件的节点，即那些满足位置谓词之前的所有谓词约束的节点。由此可知，为了完整地无二义性地表达查询中出现的位置谓词的语义，需要 pos_pred 和 p_p_position 两个字段同时进行描述。

如图 1 (b)中所示的查询 Q，用于描述查询 Q 的表结构如图 1 (c)所示。由于 B 节点的各个谓词中含有一个位置谓词 “[2]”，因此 B 节点对应的 pos_pred 字段为 2，即 get PosPred (B) = 2；由于 B 节点所需满足的位置谓词出现在了 B 节点的所有谓词的第 2 个位置，所以 B 节点对应的 p_p_position 字段为 2，即 get

PPPosition (B) = 2。对于 A 节点和 C 节点，由于没有需要满足的位置谓词，因此 A 和 C 两个节点对应的 pos_pred 和 p_p_position 两个字段均为 n/a。树模式中各节点的其他信息均与原始的 LQ 算法保持一致。

2.2 核心栈结构

在 TwigPos 算法中，运用了三层嵌套栈结构来存储当前节点项所在项序列的上下文信息。三层嵌套栈的结构如图 2 (a) 所示。SiblingStack（以下简称为 SS）栈本身会被当做 InnerStack（以下简称为 IS）栈中的元素，嵌套在 IS 栈中，IS 栈本身也会被当做 NestedStack（以下简称为 NS）栈中的元素，嵌套在 NS 栈中，以此方式构成三层嵌套栈结构。对于每个 SS 栈中的元素，均会设置 name、id、parent_id、position 四个字段，分别用以记录每个元素的节点名、节点 ID、父节点 ID、当前节点在同名兄弟节点间所处的位置这四个信息。对于每个 SS 栈，除了需以栈结构来存储各元素外，还有 sibling Count 和 failure Count 两个字段与其对应。这两个字段均是采用 Hash Map 的形式，分别记录 SS 栈中与每个查询节点对应的 XML 同名兄弟节点的总个数，以及不满足位置谓词之前所有谓词的节点的个数之和。

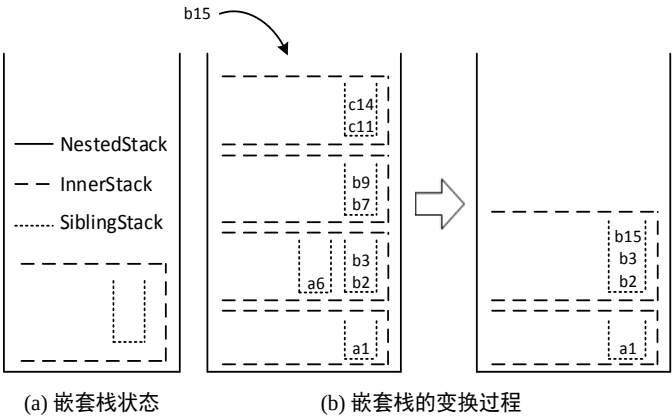


图 2 NS 嵌套栈结构

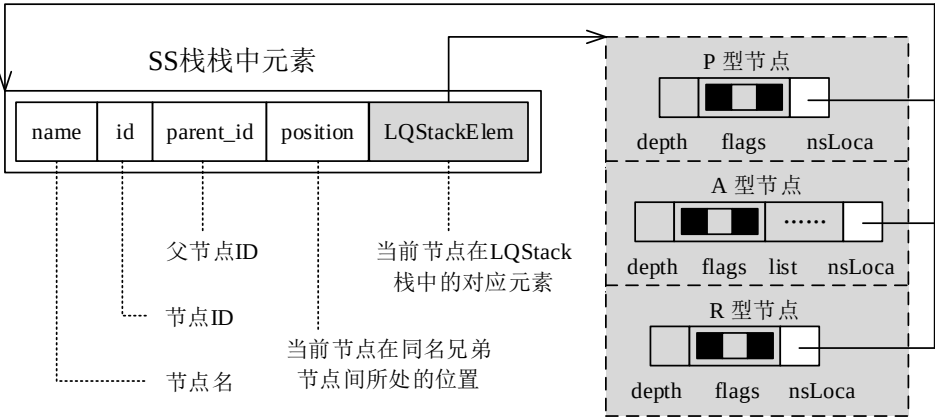


图 3 SS 栈的栈中元素结构

为了方便描述，此处假设每个栈的栈底下标为 0，栈顶下标为 top。NS[i][j][k]表示 NS 中第 i 个 IS 中第 j 个 SS 中的第 k 个 XML 元素。NS 嵌套栈的数据组织具有如下三个性质：

- (1) 每个SS栈中存储的元素互为兄弟；
- (2) 在任何一个IS栈NS[i]中，所有SS栈NS[i][j]中的XML节点的父节点是NS[i][j+1]中所有XML节点的祖先 (0≤j≤top-1)。
- (3) 若NS[top][top][top]为当前入栈节点，则NS[i][top][top] (0≤i≤top) 组成的XML节点集合为当前入栈的XML节点到根节点的路径。

为了能够保持上述三条 NS 嵌套栈的性质，在读入某个 XML 标签时，首先要“清理”当前 NS 嵌套栈

中的存储内容，只要 $NS[top][top][top]$ 的父节点不是待入栈节点的祖先，就弹出 $NS[top][top]$ 栈。其间若 $NS[top]$ 栈为空，则将此 IS 栈一并弹出。经历过上述过程之后，才能对当前节点进行入栈操作。

在将 XML 元素压入 NS 嵌套栈时分四种情况：

- 若 NS 嵌套栈为空，则将当前节点压入新建的 SS 栈中，并将此 SS 栈压入新建的 IS 栈，最后将此 IS 压入 NS 栈；
- 若当前节点为嵌套栈栈顶 $NS[top][top][top]$ 节点的后裔，则与第一种情况类似，创建新一层的 IS 栈，其中包含了只有一个当前节点的 SS 栈，并压入 NS 栈；
- 若当前节点与嵌套栈中 $NS[top][top][top]$ 节点互为兄弟，则将该节点压入 $NS[top][top]$ 的 SS 栈中；
- 若是其他情况，则向 $NS[top]$ 中压入一个只包含当前节点的 SS 。

根据上述四种情况，对当前节点进行不同的入栈操作。对于图 1 (a) 中所示的 XML 文档，图 2 (b) 给出了以标签流形式读入 b_{15} 前后 NS 嵌套栈中内容的变化。由于 a_6 、 b_7 、 b_9 、 c_{11} 、 c_{14} 的父节点均不是 b_{15} 的祖先，因此上述节点全部出栈。由于 b_{15} 与 b_2 、 b_3 互为兄弟，根据前文所述， b_{15} 被压入 b_2 和 b_3 所在的 SS 栈中。

在 PosLQ 算法进行的过程中，当前被处理的 XML 节点的信息会被划分成两部分进行存储，如图 3 所示。节点名、节点 ID、父节点 ID 和当前节点处在同名兄弟节点之间的位置等信息存储于 SS 栈中，在此基础上，PosLQ 算法对每个 SS 栈的栈中元素引入 $LQStackElem$ 字段。该字段可被看作是 LQ 算法中的核心栈结构 $LQStack$ 中元素的引用，存储当前节点在整个文档中所处的深度、当前节点的谓词匹配情况等信息。 $LQStack$ 中元素的结构根据查询节点的类型不同分为 ‘a’、‘r’、‘p’ 三种，分别代表“轴节点”、“返回节点”和“谓词节点”。三种不同的节点中所包含的 $depth$ 、 $flags$ 、 $list$ 和 $nsloca$ 四个字段分别用于记录当前嵌套栈中的元素在 XML 数据源中的深度，记录该节点各存在谓词的匹配情况，暂存以该节点为根的子树中的候选解，以及指向 SS 栈中对应的 XML 元素。通过设置 $nsLoca$ 字段，可将两部分独立的信息相互关联起来，使其一一对应，从而达到快速地定位并获取当前节点上下文信息的目的，为当前节点位置谓词满足性的判断提供重要依据。

表 1 PosLQ 算法整体处理流程

算法1: PosLQ	
<i>startElement(n)</i>	<i>endElement(n)</i>
1. $DEPTH = DEPTH + 1;$	1. $cn = mapping(n);$
2. $cn = mapping(n);$	2. if $cn \neq FAIL$
3. if $cn \neq FAIL$	3. $PosLQ::endBlock(cn);$
4. $t = new SSElement(n);$	4. $DEPTH = DEPTH - 1;$
5. $PosLQ::startBlock(cn , t);$	

2.3 PosLQ 算法流程设计

本节基于前文介绍的新的树模式表示方法，和经过扩展的三层嵌套栈结构，提出一种新的针对 XML 流式数据的树模式查询算法 PosLQ，从而达到在流数据树模式匹配算法中支持位置谓词处理的最终目的。

2.3.1 PosLQ 算法整体处理流程

PosLQ 算法整体的处理流程如表 2 所示。算法大体的驱动方式是：从 XML 数据流中获取 XML 节点数据，针对 XML 节点的开始标签和结束标签，分别触发事件处理函数 $startElement(n)$ 和 $endElement(n)$ ，进而分别触发 PosLQ 算法核心处理模块 $startBlock(cn, t)$ 和 $endBlock(cn)$ ，由这两个事件处理函数做出相应的处理动作。其中， n 表示当前读入的待处理节点； cn 表示当前读入节点对应的树模式节点的 $column_number$ ；

t 表示以当前读入节点 n 的各字段为参数，新建出的 SS 栈中元素。

2.3.2 PosLQ 算法开始标签处理模块 startBlock

PosLQ 算法的 startBlock 处理模块是读入 XML 开始标签时的核心处理部分。表 3 中给出了完整的 startBlock 算法流程。

PosLQ 算法的 startBlock 处理模块的处理流程大体分为两个步骤：NS 嵌套栈相关操作（算法 2 第 1-5 行）和 LQ 核心栈相关操作（算法 2 第 6-24 行）。startBlock 处理模块在一开始，便按照原前文介绍的入栈原则，将当前节点以 3.2 节叙述的方式暂存入三层嵌套栈 NS 中。进行过 NS 嵌套栈相关操作之后，再对读入的元素进行 LQ 核心栈相关的操作。处理的大体流程基本与 LQ 算法相类似，同样是根据节点类型和轴类型的不同进行分类讨论，对不同的情况进行不同的操作。所不同的是，在算法最后将元素压入对应的 LQ 核心栈时，PosLQ 算法需要对 NS 嵌套栈中元素的 LQStackElem 字段和 LQ 核心栈中元素的 nsLoca 字段进行设置（算法 2 第 22、24 行）。通过这样的操作，将当前读入元素所对应的 NS 嵌套栈中的元素和 LQ 核心栈中的元素两部分相互关联起来，使其一一对应，便于位置谓词信息的获取。

startBlock 过程中“压入”和“弹出”NS 嵌套栈的两个操作 NSPush 和 NSPop 也已在表 3 中给出。

表 3 PosLQ 算法开始标签处理模块

算法 2: PosLQ::startBlock(cn , t)	
<pre> 1. ss = NSPop(NS , t); 2. while ss != Null do 3. while !empty(ss) do pop(ss); 4. ss = NSPop(NS , t); 5. NSPush(NS , t) 6. cp = parent(cn); 7. if LQStack[cp] is empty return; 8. p = top(LQStack[cp]); 9. if type[cn] ≠ 'p' p.flags[pos[cn]] == 0 10. if axis[cn] == '/' p.depth + 1 == DEPTH 11. if leaf[cn] == true 12. if type[cn] == 'p' p.flags[pos[cn]] = 1; 13. else if type[cn] == 'r' 14. s = newLQStackElem(cn , DEPTH); 15. s.nsLoca = t; t.LQStackElem = s; 16. if t.position == getPosPred(cn) 17. b = newBufferNode(id) 18. if cp==0 flushNode(b); 19. else appendNode(p.list , b); 20. else 21. s = newLQStackElem(cn , DEPTH , t.id); 22. s.nsLoca = t; 23. push (LQStack[cn] , s); 24. t.LQStackElem = s; </pre>	<pre> Procedure NSPush(NS , t) 1. if empty(NS) 2. t.position = 1; 3. push(NS , new IS(new SS(t))); 4. else 5. y = top(top(top(NS))); 6. if y is an ancestor of t 7. t.position = 1; 8. push(NS , new IS(new SS(t))); 9. else if y is a sibling of t 10. t.position = 1 + GetSiblingCount (top(top(NS))); 11. push(top(top(NS)) , t); 12. else 13. t.position = 1; 14. push(top(NS) , new SS(t)); Procedure NSPop(NS , t) 1. if parent of top(top(top(NS))) is not an ancestor of t 2. ss = pop(top(NS)); 3. if empty(top(NS)) pop(NS); 4. return ss; 5. else return Null; </pre>

2.3.3 PosLQ 算法结束标签处理模块 endBlock

PosLQ 算法的 endBlock 处理模块是读入 XML 结束标签时的核心处理部分。表 4 中给出了完整的 endBlock 算法流程。

处理的大体流程也是基本与 LQ 算法相类似，根据节点类型和轴类型的不同进行分类讨论，对不同的情况进行不同的操作。然而，与 LQ 算法所不同的，也是 PosLQ 算法所最具特色的，便是 evaluate 函数。这一

函数在整个 engBlock 模块中起到了至关重要的作用，其功能是判断当前读入的节点是否满足树模式中要求的所有谓词约束。当然，这其中也包括位置谓词的检查和筛选。

endBlock 过程中需要用到的 evaluate 操作也已在表 4 中完整地给出。根据 TwigPos 算法的原理可知，位置谓词满足性的判断需要两部分信息：其一是嵌套栈 NS 中每个元素的位置信息 position；其二是 SiblingStack 中的失败计数器 FailureCount。因此，在 evaluate 判断谓词满足性时，除了需检查当前节点的 flags 数组是否均被置为 1，即检查当前节点的存在谓词满足情况，还需检查当前节点是否满足查询计划中要求的位置谓词约束，也就是检查当前节点的 position 字段是否等于树模式中给出的位置谓词 getPosPred (cn) 与失败计数器 GetSiblingFail (SS, cn) 之和（算法 3 中 evaluate 过程第 1 行）。只有同时满足上述两项检查的节点才会被判定为满足条件的节点。若当前节点不能满足上述两个条件，则还需要进行进一步检查。如果在当前节点的 flags 数组中，位置谓词之前的谓词标记位并未都被置为 1（算法 3 中 evaluate 过程第 4 行），那么则需要对当前节点对应的 FailureCount 进行加 1 操作，即将当前节点判定为“不满足位置谓词前置谓词的节点”。

2.3.4 PosLQ 算法分析

分析上述算法可知，PosLQ 算法可分为三层嵌套栈 NS 相关处理和 LQ 核心栈相关处理两个部分，因此，对于 PosLQ 算法时间复杂度的分析也可从这两方面入手。首先，对于三层嵌套栈 NS 相关处理部分，时间复杂度由两部分构成。第一部分是嵌套栈的出栈和入栈操作，即 NSPush 和 NSPop 两个过程，其时间复杂度为常量级别，即： $\sum_{i=1}^{n+1} O(1)$ 。第二部分是算法 2 第 2-4 行。最里层的循环，每次会将一个栈中的元素进行出栈操作。由此，可得出该部分的时间复杂度为： $\sum_{i=1}^{n+1} \sum_{j=0}^{M(i)} \sum_{k=1}^{N(i,j)} O(1)$ ，其中 M(i) 和 N(i, j) 分别是与循环次数相关的函数。故三层嵌套栈 NS 相关处理部分的时间复杂度可维持在 O(n) 级别，其中 n 为 XML 文档的节点数。

对于 PosLQ 算法的 LQ 核心栈相关处理部分，由于算法是以 LQ 算法为蓝本而设计，且并未引入循环、迭代等高时间复杂度的操作，因此，其时间复杂度依然是 O(n × Q)。其中 n 为 XML 文档节点数，Q 为树模式节点数。

表 4 PosLQ 算法结束标签处理模块

算法 3: LQ::endBlock(cn)			
1.	if leaf[cn] == true LQStack[cn] is empty return;	18.	
2.	s = top(LQStack[cn]);	19.	if cp == 0 flushNode(b);
3.	if s.depth == DEPTH	20.	else appendNode(p.list , b);
4.	pop(LQStack[cn]);	21.	else if type[cn] == 'a' && getPosPred[cn] == n/a
5.	if LQStack[cn] is not empty	22.	if !LQStack[ch=host[cn]]. empty()
6.	q = top(LQStack[cn]);	23.	h = top(LQStack[ch]);
7.	q.dFlags = q.dFlags s.dFlags;	24.	appendList(h.list , s.list);
8.	if evaluate(s , cn) == true	25.	else destroyList(s.list);
9.	cp = parent[cn]; p = top(LQStack[cp]);		destroyList(s);
10.	if type[cn] = 'p'	Procedure evaluate(s , cn)	
11.	p.flags[pos[cn]] = 1;	1.	if all bits in s.flags are 1 && s.nsLoca.position ==
12.	if axis[cn] == '/' clearPredStack(cn);		getPosPred(cn) + GetSiblingFail(SS, cn)
13.	else if type[cn] == 'a'	2.	return true;
14.	if cp == 0 flushList(s.list);	3.	else
15.	else appendList(p.list , s.list);	4.	if not all bits in s.flags[0..getPPPPosition(cn)+1] are 1
16.	else if type[cn] == 'r'	5.	AddSiblingFail(top(top(NS) , s);
17.	b = newBufferNode(s.id)	6.	return false;

综上所述，综合分析 PosLQ 算法的三层嵌套栈 NS 相关处理和 LQ 核心栈相关处理两个部分，可得出最

终的时间复杂度： $T(n) = n \times Q + \sum_{i=1}^{n+1} O(1) + \sum_{i=1}^{n+1} \sum_{j=0}^{M(i)} \sum_{k=1}^{N(i,j)} O(1)$ 。进而可以得出最终的结论：PosLQ 算法的时间复杂度级别与 LQ 算法相同，依然可以维持在 $O(n \times Q)$ 数量级。

3 实验

本节介绍针对 PosLQ 算法进行的各类对比实验，简要分析实验结果，旨在阐述该算法的正确性与高效性。

3.1 实验环境

实验环境为一台配有 Intel Core i7-2600 3.40GHz CPU，12G 内存的 Windows 7 PC，Java 运行环境 (JRE) 版本 1.6。实验数据采用的数据集分为两组：第一组是 3 个 XML 基准数据集，分别是两个真实数据集 DBLP 和 TreeBank，以及一个根据参数人工生成的数据集 XMark（如表 5 所示）；第二组是 6 个根据参数人工生成的不同大小的 XMark 数据集（如表 6 所示）。

待测试的 XPath 查询用例分为三组，如表 7 所示。第一组用例 (Q1.1-Q1.9) 只包含 AD 轴、PC 轴和存在谓词；第二组用例 (Q2.1-Q2.9) 用来与第三组用例 (Q3.1-Q3.9) 作对比；第三组用例中包含 AD 轴、PC 轴、存在谓词和位置谓词，并且不对位置谓词出现的个数加以限制，即一个 XPath 查询计划中允许含有多个位置谓词（如 Q3.3、Q3.6）。

表 5 三个 XML 基准数据集

数据集标号	数据集	大小 (MB)	节点数 (百万)
①	DBLP	127	3.3
②	XMark	115	1.7
③	TreeBank	82	2.4

表 6 不同大小的 XMark 基准测试集

XMark参数	大小 (MB)	节点数 (百万)
0.1	11	0.16
1	115	1.7
5	581	8.4
10	1164	16.7
15	1749	25.1
20	2333	33.4

表 7 XPath 测试案例

数据集	XPath查询计划	数据集	XPath查询计划
Q1.1 ①	//dblp/article[./author][./title][./url]	Q2.6 ②	//item[./name][./text/keyword]/incategory
Q1.2 ①	//dblp/article[./author][./title]/year	Q2.7 ③	//S/VP/PP[./NP/VBN]/IN
Q1.3 ①	//dblp/article[./author][./title][./url][./ee]/year	Q2.8 ③	//S/VP/PP[./NP/VBN][./IN]
Q1.4 ②	//open_auction[./initial]/bidder/personref	Q2.9 ③	//NP/VP/_NONE_
Q1.5 ②	//item[./location][./mailbox/mail//emph]/description/keyword	Q3.1 ①	//dblp/inproceedings[3][./title]/author
Q1.6 ②	//category[./name][./description/listitem[./keyword]/parlist[./emph]]	Q3.2 ①	//dblp/article[2][./author][./title][./url]/year
Q1.7 ③	//S/VP[./PP/IN]/NP	Q3.3 ①	//dblp/inproceedings[./author[4]][2]/crossref
Q1.8 ③	//S/VP/PP[./IN]/NP[./CD]/VBN	Q3.4 ②	//people/person[88][./profile/education]
Q1.9 ③	//S/VP/PP[./NN][./NP[./CD]/VBN]/IN	Q3.5 ②	//people/person[./profile][36]/interest
Q2.1 ①	//dblp/inproceedings[./title]/author	Q3.6 ②	//item[./name][2][./text/keyword[6]]/incategory
Q2.2 ①	//dblp/article[./author][./title][./url]/year	Q3.7 ③	//S/VP/PP[2][./NP/VBN]/IN
Q2.3 ①	//dblp/inproceedings[./author]/crossref	Q3.8 ③	//S/VP/PP[./NP/VBN][2][./IN]
Q2.4 ②	//people/person[./profile/education]	Q3.9 ③	//NP/VP/_NONE_[2]
Q2.5 ②	//people/person[./profile]/interest		

3.2 测试结果及分析

为了检验 PosLQ 算法的正确性,我们分别在上述 XML 数据集上进行了测试,并将 PosLQ 算法得到的查询结果与 LQ 算法、TwigPos 算法及目前现有的 XPath 查询引擎得到的查询结果进行对比,确保相互一致。

在检验 PosLQ 算法的高效性时,我们以 XML 数据流的查询执行时间作为衡量依据。PosLQ 算法的有效执行时间 $T = t_{total} - t_{in} - t_{out}$, 其中, t_{total} 为 XML 数据流处理的总时间, t_{in} 为从磁盘上读入 XML 文档并对其进行解析的时间, t_{out} 为从内存向磁盘输出查询结果的时间。

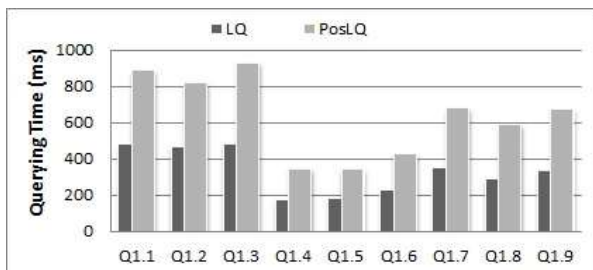


图 4 LQ 算法与 PosLQ 算法性能对比

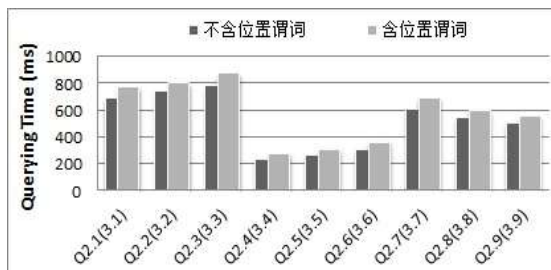


图 5 PosLQ 算法处理位置谓词的性能对比

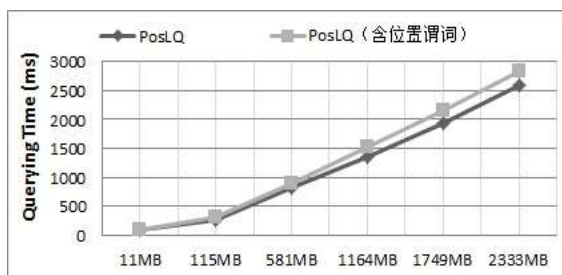


图 6 不同大小 XMark 数据集上的执行时间变化趋势

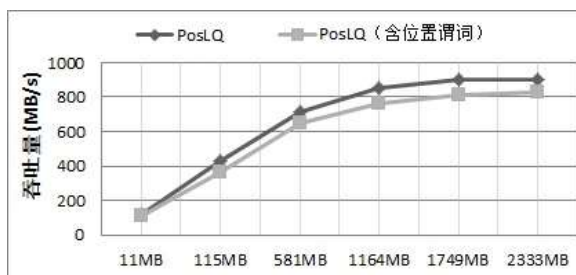


图 7 PosLQ 算法吞吐量变化趋势

图 4 反映了针对 Q1.1-Q1.9 分别执行 LQ 算法和 PosLQ 算法的执行时间对比。上述查询的共同特点是查询计划均中不含位置谓词,且较之其他的测试案例,存在谓词较复杂,包含并列、嵌套及相互组合等各种情况。可以看出,PosLQ 算法依然可以高效地处理谓词的各种并列和嵌套情况,但由于 PosLQ 算法引入了三层嵌套栈机制,因此与 LQ 算法相比,在处理不含位置谓词的查询计划时,会有一定的额外处理开销。但尽管如此,PosLQ 算法的执行效率仍与 LQ 算法维持在同一数量级。考虑到 PosLQ 算法在功能上扩展了对位置谓词的支持,因此上述额外的开销是可以被接受的。

图 5 反映了针对 Q2.1-Q2.9 和 Q3.1-Q3.9 执行 PosLQ 算法的执行时间对比。其中,查询计划 Q3.1-Q3.9 均是在 Q2.1-Q2.9 的基础上添加位置谓词而得到的。可以看出,PosLQ 算法在处理上述两组查询计划时,所花费的时间开销基本相当,可以维持在同一数量级。

图 6 反映了针对表 6 所示的不同大小的 XMark 数据集,PosLQ 算法在执行 Q2.5 和 Q3.5 时的执行时间变化趋势。可以看出,随着数据集的增大,PosLQ 算法的执行时间基本呈线性增长趋势。

针对图 6 得出的实验数据,我们对 PosLQ 算法进行吞吐量的计算,即求得查询处理的数据量(MB)与查询时间(s)的比值。求得的结果如图 7 所示。可以看出,随着数据量的不断增大,PosLQ 算法的吞吐量逐渐趋于稳定,对于不含位置谓词的查询计划,PosLQ 算法的吞吐量可以达到 902MB/s,而对于含有位置谓词的查询计划,PosLQ 算法的吞吐量亦可以达到 825MB/s。

4 相关工作

在基于堆栈的 XML 流数据树模式查询技术领域,目前,针对 XML 流数据的树模式查询算法主要分为三类:基于自动机 (automaton-based)、基于数组 (array-based)、基于栈 (stack-based)。在一般情形下,基于栈的 XML 流数据树模式匹配算法性能更优,在时间和空间复杂度上都有着明显的优势^[8]。

2006 年, Yi Chen 等人提出了 TwigM 算法。算法中使用了一个小巧的栈式数据结构组织树模式匹配过程中所需的信息, 而不是显式地将它们存储起来。该方法首次将流数据树模式查询的计算复杂度由原先的指数级别降至多项式级别, 对各种数据集的查询也表现出良好的性能。2007 年, Gang Gou 等人提出了 LQ 系列算法。这一算法可被看作是对 TwigM 算法的改进和补充。算法最突出的贡献就是首次将流数据树模式查询的计算复杂度控制在 $O(|D| |Q|)$ 之内, 同时具有较为良好的空间复杂性, 其中 $|D|$ 表示 XML 文档节点数, $|Q|$ 表示树模式节点数。这样的时间和空间复杂度要比之前介绍的 TwigM 算法更加优秀。2008 年, Wook-Shin Han 等人提出了 StreamTX 算法。由于该算法是从 TwigStack 算法^[9]演变而来的, 因此 StreamTX 算法的一大特点就是可支持查询中含有多个返回节点。实验结果表明, 虽然 StreamTX 的时空复杂度与基于数组的流数据树模式匹配算法相比有较为明显的提高, 但还是没有达到 LQ 算法的数量级。主要原因是: StreamTX 算法在进行过程中, 把每一个树模式节点都认为是返回节点, 记录了每一个节点的信息, 这些信息对于非返回节点而言是冗余的。此外, 在 StreamTX 算法的最后需要枚举过程, 这也是 LQ 算法所不需要的。

在支持含有位置谓词的树模式查询技术领域, 2005 年, Z Vagena 等人基于二元结构连接的树模式匹配算法, 提出了一种判断位置谓词满足性的方法^[10], 用于处理含有位置谓词的查询计划。但是, 该算法存在大量冗余的中间结果, 且不适用于流式 XML 数据。2013 年, Fajin Li 等人提出了 TwigPos 算法。算法最突出的贡献就是运用特殊的“三层嵌套栈”结构, 提出了一种新的整体树模式匹配算法 TwigPos, 填补了无法从树模式整体匹配结果中筛选符合位置谓词的查询结果这一技术空缺。但是, 该算法的应用领域仅局限于非流式 XML 数据, 对流式 XML 数据, 该算法并不能很好地支持。

5 结论

针对现有 XML 流数据树模式查询算法均不支持含位置谓词的 XPath 查询等问题, 本文基于 TwigPos 算法和 LQ 算法的工作基础, 提出并实现了一种面向 XML 流数据的整体树模式匹配算法 PosLQ, 支持含有位置谓词的 XPath 查询, 填补了目前 XML 流数据领域无法从树模式整体匹配结果中筛选出符合位置谓词的结果这一技术空缺。实验表明, PosLQ 算法在扩展了 XML 流数据树模式查询支持范围的同时, 也保证了较高的树模式查询执行效率。

REFERENCES

- [1] Weidong Y, Baile S. A Survey of XML Stream Management[J]. Journal of Computer Research and Development, 2009, 10: 018
- [2] Deng Z, Liao H, Gao H. Twig pattern matching running on XML streams[M]//Web Technologies and Applications. Springer Berlin Heidelberg, 2012: 35-42
- [3] Meng Xiaofeng. XML Data Management Concepts and Techniques[M]. Beijing: Tsinghua University Press, 2009
- [4] Chen Y, Davidson S B, Zheng Y. An efficient XPath query processor for XML streams[C]//Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on. IEEE, 2006: 79-79
- [5] Gou G, Chirkova R. Efficient algorithms for evaluating XPath over streams[C]//Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007: 269-280
- [6] Han W S, Jiang H, Ho H, et al. StreamTX: extracting tuples from streaming XML data[J]. Proceedings of the VLDB Endowment, 2008, 1(1): 289-300
- [7] Fajin Li, Husheng Liao, Hongyu Gao. Twig Pattern Matching with Positional Predicates in XML queries [J]. Journal of Computer Research and Development, 2013, 50(z1)
- [8] Wu X, Theodoratos D. A survey on XML streaming evaluation techniques[J]. The VLDB Journal—the International Journal on Very Large Data Bases, 2013, 22(2): 177-202
- [9] Bruno N, Koudas N, Srivastava D. Holistic twig joins: optimal XML pattern matching[C]//Proceedings of the 2002 ACM SIGMOD international conference on Management of data. ACM, 2002: 310-321

- [10] Vagena Z, Koudas N, Srivastava D, et al. Efficient handling of positional predicates within XML query processing[M]//Database and XML Technologies. Springer Berlin Heidelberg, 2005: 68-83

【作者简介】



¹ 王晨阳（1987-），男，汉族，硕士研究生，主要研究领域为 XML 数据库技术。Email: eagle_james@163.com



² 廖湖声（1954-），男，汉族，北京工业大学教授、博士生导师，主要研究领域为软件自动化方法，编译技术、XML 数据处理、程序分析与程序变换。
Email: liaohs@bjut.edu.cn



³ 苏航（1978-），男，汉族，博士，讲师，主要研究领域为编译技术、XML 数据处理、程序分析与程序变换。
Email: suhang@bjut.edu.cn