XML 数据流上的有序 XPath 查询处理

谢敏[1] 王小锋[1] 张新[1] 孟小峰[1] 周军锋[1,2]

[1](中国人民大学信息学院，北京 100872)

[2](燕山大学计算机系，河北 秦皇岛 066004)

(xiemin@ruc.edu.cn)

摘要 XML 数据流上的查询处理是最近研究工作的一个热点，如何高效地处理 XML 数据流上的 XPath 查询是其中的核心问题。之前的相关工作主要考虑了无序 XPath 查询处理的情况，而在股票信息监控，新闻信息订阅等很多的 XML 数据流应用中常常需要对有序 XPath 查询进行有效的支持。对于有序 XPath 查询的处理，之前的方法需要将查询进行分解，然后通过连接将分解后的子查询得到的中间结果合并。针对有序 XPath 查询自身的特点，本文提出了在查询树上引入顺序和位置标记，记录查询结点之间的顺序关系，并在此基础上提出了一种创新的 XML 数据流上的 XPath 查询处理算法 OrderedXP。相比之前的工作，OrderedXP 能够大量地减少缓存的中间结果数目，而且不需要分解原来的查询，避免了额外的连接操作。详细的实验数据验证了 OrderedXP 能够显著地提高有序 XPath 查询在 XML 数据流上的执行效率。

关键字 XML 数据流；XPath；查询处理

中图法分类号 TP391

Ordered XPath Query Processing on XML Stream

Xie Min[1], Wang Xiaofeng[1], Zhang Xin[1], Meng Xiaofeng[1], and Zhou Junfeng[1,2]

[1](Information School, Renmin University, Beijing 100872)

[2](College of Computer Science, Yanshan University, HeBei Qinghuangdao 066004)

Abstract Recently, query processing on the XML stream is a hot topic in research community, in which how to efficiently handle XPath query on the XML stream is a core problem. On dealing with this problem, the previous work mainly concerns how to efficiently handle unordered XPath query, but in the application of XML stream like stock information monitoring, news feed monitoring and etc., we often need to handle ordered XPath query. On concerning these requests, the previous methods often break the XPath into some query Fragments and execute them separately, at the last stage these algorithms refer to an explicitly join to get the final results. On observing the characteristic of the ordered XPath query, we bring order specification and position specification into the query tree which may record the order relationship between the query nodes, and then we propose a novel XPath query processing method OrderedXP for XML stream, which can to the maximum extent reduce the number of intermediate results we cache in the memory compared to the previous work with the additional benefits of no decomposition and final join step. Extensive experiments show that our OrderedXP algorithm can handle all the ordered XPath query efficiently.

Keywords XML Stream; XPath; Query Processing

1. 引 言

可扩展标记语言(XML)[1]由于其易用性和强大的复杂数据描述能力，已经逐渐成为因特网上数据交换的标准。很多流行的数据库引擎已经开始通过增加对于这种新数据类型以及相关操作的支持来满足新的需求。XML 数据流上的应用是 XML 研究的一个很重要的方面，在新的基于 Web 的应用场景下，有研究已经提出，有效地处理 XML 数据流上的 *XPath* [2]，*XQuery*[3]查询将会成为下一代信息系统的特点[4，5]。

收稿日期：2006-05-25
基金项目：国家自然科学基金项目（60573091，60273018）

XPath[2]是 W3C 推荐的 XML 上的查询语言,用于抽取满足条件的 XML 文档片段。之前在 XML 数据流上有很多关于 *XPath* 查询处理的工作[4，5，6，7]，这些工作主要集中在讨论 XML 数据流上处理 XPath 中结点间的父子(*Parent-Child*)关系，祖先后代(*Ancestor-Descendant*)关系,以及 *XPath* 中的谓词，分支结点。这些之前工作处理的 *XPath* 查询都不涉及数据上的顺序关系。我们把这些不涉及 XML 数据顺序关系的 *XPath* 查询称作***无序 XPath 查询***(*Unordered XPath Query*)。

但是在 XML 数据流的实际应用,比如股票信息监控和新闻信息过滤等,有很多的查询会对数据的出现顺序有一定的要求[8]。如图 1 所示，查询 Q_1 需要查找满足先后顺序要求的两个交易的情况,查询 Q_3 需要查找位置满足某一要求的特定新闻，而查询 Q_2 和 Q_4 则需要查找同时满足先后顺序和位置要求的特定交易信息和新闻信息。这些查询无论有先后顺序的要求还是有位置的要求，一个共同的特点就是对 XML 数据流上查询涉及到的结点的出现顺序有一定的限制。

Q_1	查找在IBM和联想交易发生之后的微软的交易
Q_2	查找IBM在与联想交易后发生的第1笔交易
Q_3	查找关于NBA火箭队的头10条新闻
Q_4	查找北京申奥成功新闻发布后出现的头10条新闻

图 1 XML 数据流上的例子查询

我们把这些查询中涉及结点上的先后顺序或者位置的要求称作查询的***有序要求***(*Order Request*)。这些有序要求直接转化成 *XPath* 中的 *Following* 关系，*Following-Sibling* 关系和 *Position* 谓词。我们把这种含有 *Following* (*Preceding*)关系，*Following-Sibling*(*Preceding-Sibling*)关系或者 *Position* 谓词的 *XPath* 查询称为有顺序要求的 *XPath* 查询，在文章里为了描述方便，简称为***有序 XPath 查询***(*Ordered XPath Query*)。下面的讨论中，我们将假设查询中不含有 *Preceding* (*Preceding-Sibling*)轴，可以通过之前的工作[4]将其转换成 *Following*(*Following-Sibling*)轴进行处理。

之前的工作在处理 XML 数据流上的 *XPath* 查询的时候，当查询中出现 *Following* 或者是 *Following-Sibling* 轴时，往往需要从 *Following*，或者是 *Following-Sibling* 轴处将查询分解成为多个查询，然后将其各自的结果通过一个额外的连接操作来得到最后的查询结果[4]。这种分解的方法会造成两个方面的问题：首先，因为将查询分解成了几个部分，会造成缓存很多无用的中间结果，比如当查询是 *//A/B/Following-Sibling::C*,而 XML 数据如图 2(*a*)所示，我们将查询分解成两个查询，*A/B* 和 *A/C* 将会造成缓存 *n* 个无用的中间结果 *b* 和 *c*，而当查询是 *//A/Following::B*，XML 数据如图 2(*b*)所示，我们将查询分解成为两个查询 *//A* 和 *//B* 将会造成缓存无用的中间结果 *b*；其次，在得到分解后的子查询的中间结果后，我们需要一个额外的连接操作才能得到最后的结果。而对于 *Position* 谓词，之前所有的方法都没有考虑 XML 数据流上处理含有这种谓词的 *XPath* 查询。

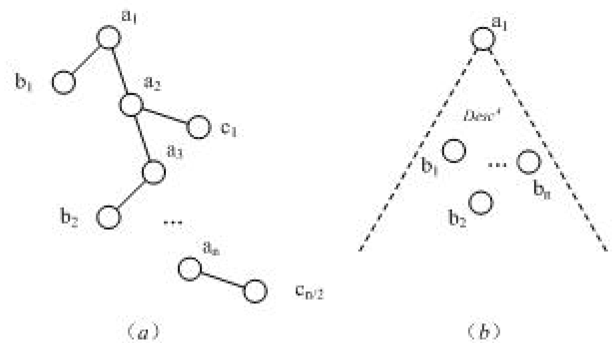

图 2 两个 XML 文档的例子

针对上面提到的 XML 数据流上有序 *XPath* 查询的需求以及之前工作中存在的问题，我们创新性地提出了一种在 XML 数据流上处理有序 *XPath* 查询的方法。本文的贡献如下：

- 分析了 XML 数据流上的三种类型有序关系 *Following*，*Following-Sibling*，*Position* 各自的特点,在查询树上引入了 O-Spec(顺序标记)和 P-Spec(位置标记)，通过这些标记的组合可以完整地描述 *XPath* 查询的有序要求涉及的结点之间关系。
- 结合顺序标记 O-Spec 和位置标记 P-Spec，本文给出了一种创新算法 OrderedXP，相对于之前的工作可以有效地减少缓存的中间结果的数目，并且避免了额外的连接操作。
- 详细的实验数据验证了本文提出的 OrderedXP 查询算法能非常有效地处理 XML 数据流上的有序 *XPath* 查询。

本文后面部分的组织如下：第 2 部分描述有序 *XPath* 查询,及各种有序关系在流处理情况下的特点；第 3 部分阐述 OrderedXP 算法；第 4 部分讨论实验情况；第 5 部分介绍相关工作；最后一部分是我们的工作总结和展望。

2. XML 数据流上有序 XPath 查询及其分析

2.1 XML 数据流模型和有序 XPath 查询

XML 是具有有序特性的树结构，结点包括元素

(*Element*)结点和文本结点。XML 上结点的先序遍历顺序反映了该结点在 XML 文件中的位置，对应到结点的文档顺序(*Document Order*)。

XML 数据流是由一系列连续的 SAX 事件流组成。而这些 SAX 事件流由 SAX 分析器分析 XML 文件流的时候产生，对每个 XML 上的元素结点，开始和结束标签都会对应到一个 SAX 事件。

本文处理的有序 *XPath* 查询是 W3C 推荐的 *XPath* 查询[2]的一个子集，如图 3 所示。如引言中所述，我们不考虑 *Following*，*Ancestor* 等逆向轴，在查询中涉及到这些轴的时候，我们可以用现有的工作[4]首先将其进行转换，再用本文的方法进行处理。

```
PathExpr ::=  '/' Path | '//' Path
Path     ::= Path Step Path | Path '[' Pred ']' | label
Step     ::=  '/' | '//' | 'following' | 'following-sibling'
Pred     ::= Path | PosPred
PosPred  ::=  'Position()' Oprel PosDef
Oprel    ::=  '<' | '<=' | '>' | '>=' | '='
PosDef   ::= Dec-Number | 'Last()' - Dec-Number
```

图 3 本文处理的 XPath 查询

在详细分析各种有序关系的特点之前，我们在这里首先给出在 XML 数据流上的查询处理一些基本假设和定义。

定义 1：*对于一个数据结点 T 来说，假设 T 对应查询结点Q_T，那么如果对于Q_T在查询树上的所有后代结点Q_T^C，都能在数据中找到一个匹配结点T^C，而且 T 以及所有的这些匹配结点T^C满足Q_T为根的查询子树上的所有的边要求(假设Q_T对应的子树为SQ_T，$\forall x, y \in \{T \cup T^C\}, x \neq y$，如果$x$，$y$对应的查询结点$Q_T$，$Q_T'$在$SQ_T$上有边关系 Edge-Req，那么$x$，$y$也满足 Edge-Req)，我们就说数据结点 T 有一个**子查询匹配**。*

引理 1：*XML 数据流上无序 XPath 查询处理过程中，遇到一个数据结点 T 的结束标签</T>时，能够判断：这个结点 T 要么有一个子查询匹配，要么没有。*

这个引理的证明可以由之前工作[4]中得到。但是当查询中涉及 *Following-Sibling* 或者是 *Following* 关系以及 *Position* 谓词时，很显然，我们在遇到</*T*>时，不能确定 *T* 是否有子查询匹配。

有序 *XPath* 查询在遇到结点结束标签时不能确定结点的子查询匹配，这种情况是之前的工作所不能处理的。在这里，我们提出 XML 数据流上处理这种有序 *XPath* 查询时，一种好的策略应该具有的特征如定义 2 所述。

定义 2： *我们说 XML 数据流上一个有序 XPath 查询处理是**最优的**(Optimal)，当且仅当因为查询中的有序关系结点 T 在遇到结束标签被缓存之后：1.如果结点 T 是有用的，能够在遇到第一个子查询匹配时对结点 T 做出判断；2.如果结点 T 没有匹配，也可以在遇到第一个能够判断 T 没有子查询匹配的标签流过时对 T 做出判断。*

我们称最优处理第一方面的条件为**最早肯定条件**，第二方面条件为**最早否定条件**，如果一个查询处理策略同时满足最早肯定条件和最早否定条件，那么这个查询处理策略是**最优的**。

如果一个有序 *XPath* 查询处理是最优的，那么算法必然能够以最少的缓存消耗完成 XML 流上的查询。下面两个小节我们将分别分析不同的有序要求在处理时要达到最优目标所需要的条件。

下文中为了描述方便，*Following-Sibling* 简称为 F-S， *Ancestor-Descendant* 简称为 A-D，而 P-C 代指 *Parent-Child*。对于查询中的 *A/Axis::B*，称 *A* 为轴关系的上下文(*Context*)结点，*B* 为轴关系的目标(*Target*)结点。

2.2 Following-Sibling,Following 分析

F-S 在处理时会遇到两种情况。第一种情况，F-S 关系的上下文结点跟其父查询结点是父子边关系，如图 4(*a*)所示。决定 F-S 关系中上下文结点 *B* 是否有子查询匹配就需要看在 *B* 结束之后结束的 *C* 中是否有 *B* 的右兄弟。而在 *B* 结束标签之后结束的 *C* 结点从图 4(*b*)中可以看到有 4 种情况。对应最优处理策略的两个条件，对于 *B* 而言：如果 *B* 是有子查询匹配的，那么在遇到第一个 *C* 兄弟结束标签时，我们就能对 *B* 做出判断；而如果 *B* 没有右兄弟，我们能够最早决定 *B* 结点没有子查询匹配的事件是 *A* 结点的结束事件。

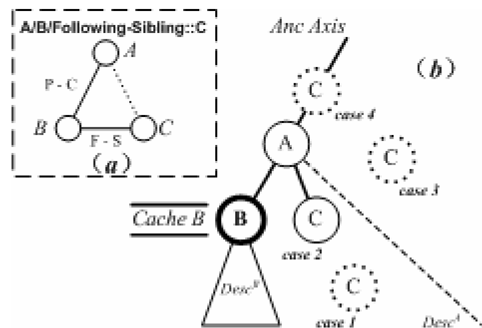

图 4 F-S 关系对数据结点的影响(1)

另一种情况，当 F-S 上下文结点 *B* 和其查询父结点是 A-D 关系的时候，找到一个最优的处理策略会比较困难。如图 5 (a)所示的 *XPath* 查询，我们可以看到在 *B* 结点结束之后结束的 *C* 的情况增加到 6 种。如果我们还是用上面的策略来处理的话，我们就不能保证查询处理最优要求的最早否定条件。原因是我们应该在数据中遇到 *B* 的父亲 *D* 的结束标签时就可以判断 *B* 没有子查询匹配，在 *A* 结束时判断就会将决定 *B* 的

时机延后。但是因为我们在数据流过之前并不知道 *B* 的父亲结点是 *D*，所以需要在查询时动态做记录，才能保证处理策略的最优。在本文里，我们暂时用之前的方法来处理这种上下文结点和其父查询结点是 A-D 关系的 F-S 的情况，在 *A* 结束时清除 *B* 结点的缓存。

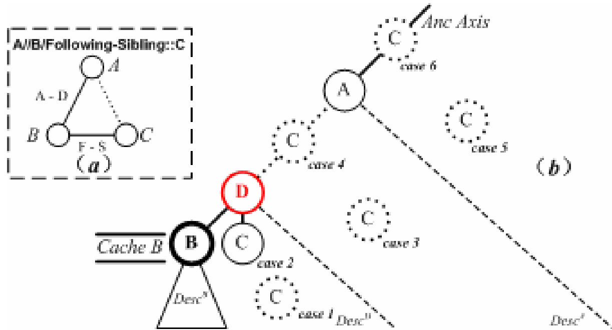

图 5 F-S 关系对数据结点的影响(2)

Following 的情况与 *Following-Sibling* 有所不同。对于一个查询 *A/Following::B* 来说，从图 6(*a*)可以看出，在 *A* 结束之后结束的 *B* 一共有两种情况：*A* 的祖先；*A* 的 *Following*。就查询处理策略最优的第一个条件而言，如果 *A* 有子查询匹配的话，我们需要缓存 *A* 直到 *A* 的有子查询匹配的 *Following* 结点出现；而对于第二个条件，如果 *A* 是没有子查询匹配的，那么我们需要将 *A* 缓存直到 XML 文档流结束，之前的任意时间我们释放 *A* 的缓存都会造成丢失解。

在查询中出现 *Following* 轴时还会导致**级联缓存**的问题。考虑查询 *C/A/Following::B*，如图 6(b)所示，我们需要对 *Following* 轴的做特殊的级联缓存来处理这种情况。

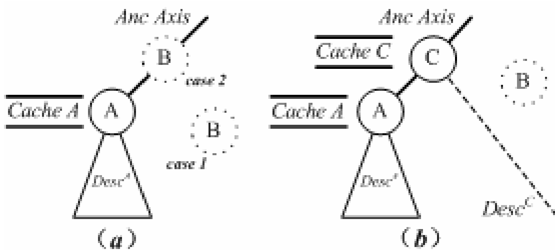

图 6 Following 关系对数据结点的影响

2.3 Position 谓词分析

Position 谓词(位置谓词)可以分为两种情况，不含有 *Last()* 的简单位置谓词和含有 *Last()* 的复杂位置谓词。对于两种谓词我们都可以通过在缓存的数据结点上增加一个计数器和缓冲队列进行处理，较为简单，在这里我们将不详细描述这部分的工作。

2.4 查询结点的 O-Spec 和 P-Spec

根据上面对于两种有序关系的分析，我们在查询树上为有序关系相关的查询结点引入 O-Spec(顺序标记)和 P-Spec(位置标记)，通过这些标记的组合我们可以将一个有序 XPath 查询的所有的有序要求反映到查询树的结点，这样可以根据 2.2，2.3 节的描述通过在处理结点开始或结束事件时，作相应的操作来支持有序 *XPath* 查询。

O-Spec(*Order-Specification* 顺序标记)用于标记先后顺序关系要求涉及的结点及其相互的关系。O-Spec 由一个三元组{*Type*， *Relation*， *Target*}构成。根据先后顺序关系类型的不同(*Following*，F-S)，我们分为两种情况来考虑如何给结点设置合适的 O-Spec 标签：

1. 对查询中的一个先后顺序关系 F-S，为 F-S 的上下文结点 $Q_{context}$ 设置 O-Spec {FS，ls，-}，为 F-S 的目标结点 Q_{target} 设置 O-Spec {FS，rs，-}。如果 F-S 的上下文结点和其父查询结点 Q_{parent} 之间是 P-C 或者 A-D 关系，那么我们在结点 Q_{parent} 上设置 O-Spec {FS，p，$Q_{context}$}
2. 对查询中的一个先后顺序关系 *Following*，为 *Following* 关系的上下文结点 $Q_{context}$ 设置 O-Spec {F，l，-}，为 *Following* 关系的目标结点 Q_{target} 设置 O-Spec {F，r，-}。级联缓存的情况由 *Relation* 为 *c* 来表示。

根据结点上的 O-Spec 标签，我们可以依据 2.2 节所述，对查询中的先后顺序关系进行相应的处理。

P-Spec(*Position-Specification* 位置标记)用于标记位置关系涉及结点之间的相互关系。P-Spec 由一个四元组{*Type*， *Target*， *Op*， *Num*}构成。根据结点上的 P-Spec 标签，我们可以依据 2.3 节所述，结合计数器和缓存队列，对查询中的 Position 进行相应的处理。

通过将查询中的有序关系转化成查询树上结点的 O-Spec 标记和 P-Spec 标记，我们可以将原来的有序查询问题简化为在遇到数据结点的开始或结束标签时根据结点的标记来做适当的缓存和检查操作。从下节的 OrderedXP 算法可以看出，通过这种方式，我们可以在一遍遍历 XML 数据流的情况下完成有序 *XPath* 查询。

3. 一种有序 XPath 查询算法 OrderedXP

3.1 相关的数据结构

在详细阐述有序 *XPath* 查询处理算法 OrderedXP 之前，下面先介绍算法涉及的数据结构。

我们的基础数据结构同之前的工作[4]类似，对给出的 *XPath* 查询，首先将其转化成一个查询树，根据有序 *XPath* 查询的需要，我们扩展了两部分的数据结构：首先，对于查询结点对应的数据结构，我们增

加了相应的 O-Spec 和 P-Spec 标签，如果查询结点 *A* 的 O-Spec 有{*F*, *1*, -}，{*FS*, *1s*, -}，我们给其查询结点 DS 增加一个缓存队列 CQ_A；然后，对于每个缓存的数据结点，如查询结点 *A* 有 P-Spec{*Complex*, *target*, *op*, *num*}，给数据流流过时给 *A* 对应的每个数据结点 *a* 的数据结点 DS 增加一个 *target* 的队列 Q_{target}。

下面是一个查询的数据结构的例子。对于查询 *A//B[C//Following-Sibling:: D]/E[position() = Last()]*而言，其对应的查询树结构如图 7 所示。

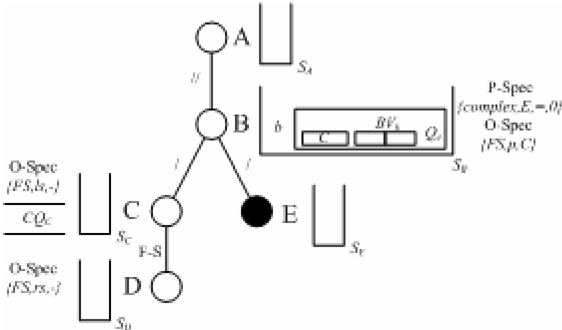

图 7 XPath 查询对应的查询树

3.2 OrderedXP 算法

在描述完 OrderedXP 算法涉及的数据结构之后，下面将会详细描述算法的流程。

StartElement 比较简单，同之前的工作[4]类似，根据结点的标签到查询树上找到数据结点 *n* 对应的查询结点 Q_n 和其查询的父结点 Q_{parent} (2-3 行)。注意这里如果父结点不存在，表示对应查询的根结点，直接将数据结点压栈并返回(4-6 行)。然后判断当前的结点 *n* 是否在查询父结点的栈 $Stack_p$ 或者缓存(如果边关系是 F-S 或者 *Following*)中有对应的满足边关系的数据结点(9-10 行)，如果有的话，将 *n* 放入当前查询结点 Q_n 的栈中。

```
Algorithm 1: StartElement(n, QT)
   Input: n is the label of the element
   Input: QT is the Query-Tree structure
1  begin
2     Q_n = QT.findQueryNode(n)
3     Q_parent = QT.getParentNode(Q_n)
4     if Q_parent is NULL then
5        Q_n.getStack().push(n)
6        RETURN
7     Edge = Q_n.getEdge()
8     Stack_p = Q_parent.getStack()
9     if Stack_p.existValidParent(n, Edge) then
10       Q_n.getStack().push(n)
11 end
```

算法 1

EndElement 由下面的算法 2 可以看出分为 6 个步骤：1.检查查询结点的 P-Spec，如果当前结点是某个 F-S 关系上下文结点的父结点，检查该 F-S 关系上下文结点的缓存，删除所有缓存的结点中该 F-S 关系不满足的结点；2.如果当前结点有 P-Spec 标签，检查当前结点的关于孩子(/)或后代(//)子查询结点的 *Counter* 和缓存队列，如果有子查询结点的 Position 谓词不满足条件，当前结点必然没有子查询匹配，做适当清理后返回；3.如果有子查询结点不满足，且该子查询结点不造成当前结点级联缓存，则肯定当前结点不含有子查询匹配，做适当的清理然后返回；4.如果结点的 O-Spec 标签含有{*F*, *1*, -}或者是{*FS*, *1s*, -}，将结点放入缓存 *CQ* 中并返回；5.如果结点的 O-Spec 有标签{*F*, *c*, *target*}，则查看是否已经有合适的 *target* 结点流过，如果没有，将结点放入缓存并返回；6.表示当前结点有一个子查询匹配，通知父查询结点检查缓存结点的相应信息。

```
Algorithm 2: EndElement(n, QT)
   Input: n is the label of the element
   Input: QT is the Query-Tree structure
1  begin
2     Q_n = QT.findQueryNode(n)
3     item = Q_n.getStack.pop()
4     /*STEP 1*/
5     if SOME Q_n.P-Spec LIKE {FS,p,*} then
6        ClearFSCache(QT, item)
7     /*STEP 2*/
8     if Q_n.P-Spec NOT NULL then
9        if ¬ StructurePosPredicateSatisfy(Q_n, item) then
10          Clear and Return
11    /*STEP 3*/
12    BV_stru = item.getStructureBV()
13    if ¬ AllSatisfy(BV_stru) then
14       Clear and Return
15    /*STEP 4*/
16    if SOME Q_n.O-Spec LIKE {F,l,−} OR {FS,ls,−} then
17       Cache item into Q_n.CQ and Return
18    /*STEP 5*/
19    if SOME Q_n.O-Spec LIKE {F,c,target} then
20       if ¬ has match target item then
21          Cache item into Q_n.CQ and Return
22    /*STEP 6 SUCCESSFUL MATCH*/
23    Q_parent = QT.getParentNode(Q_n)
24    NotifyParentQueryNode(Q_parent,item)
25 end
```

算法 2

通知父查询结点的函数 *NotifyParentQuery Node* 是系统一个比较重要的操作，主要功能是在当前查询结点找到一个子查询匹配之后通知父查询结点，修改相应的状态向量 *BV*，缓存结点的状态等等。

从第 2 节的讨论可以看出，我们的方法可以在一遍扫描数据的情况下，完整地支持有序 *XPath* 查询。

4. 实验

为了验证本文提出的算法的性能和相对于之前

工作效率的提升，实验我们将分为两部分：一部分是 OrderedXP 算法和之前工作(TwigM[4])的比较(这里我们在原 TwigM 的算法基础上实现了 TwigMO 算法用于有序要求的处理，根据原文献[4]的描述，我们采用了先查询分解最后连接过滤的方式)；另一部分是算法在数据规模增长的情况下性能的分析。

实验的环境配置：CPU 为 P4 2.0G，内存 768M，80G ATA 硬盘，操作系统是 Windows XP SP2，SAX 分析器使用了 Apache Xerces C++ Parser (Version 2.7.0)[9]。

这里我们使用了通用的 XMark 数据集作为我们的实验数据集。在数据集上用到的查询 Q_1 是只含有顺序关系的查询，Q_2 是只含有位置谓词的查询，Q_3 是既含有顺序关系又含有位置谓词的查询。

Q_1	//person/name[/fs::phone]
Q_2	//item/incategory[position()=last()-3]
Q_3	//item/location[/fs::incategory[2]]

表 1

表 2 是查询 Q_1，Q_2，Q_3 在 XMark3 上的查询实验数据。可以看到，OrderedXP 相对于之前的方法(TwigMO)，大大减少了中间缓存结点的数目，这是因为我们在数据流经过的同时检查缓存数据结点的有效性，将所有缓存的无用中间结点清除，避免了最后的连接操作。同时也可以看到 OrderedXP 相对于之前的方法也有较好的时间性能表现。

XMark3	算法	缓存结点数目	结果数目	时间 ms
Q_1	OrderedXP	2	6368	41958
	TwigMO	19118	6368	43771
Q_2	OrderedXP	9	4545	41110
	TwigMO	40925	4545	41400
Q_3	OrderedXP	3	9235	41520
	TwigMO	52800	9235	43100

表 2

图 8 是 OrderedXP 算法在 XMark1(5M)到 XMark5 (200M)上执行同时含有有序要求和位置谓词查询 Q_3 的查询时间(ms)分析。从图 8 可以看出，OrderedXP 算法的耗时随 XML 数据流大小增长线型增加，容易得到我们提出的 OrderedXP 算法有较好的扩展性。

5. 总结和未来工作

本文提出了在 XML 数据流上做有序 *XPath* 查询的问题，详细地分析了各种有序关系在数据流情况下处理可能的最优策略。根据这些分析将有序 *XPath* 查询中的有序要求转化成了查询树上结点上的 O-Spec 和 P-Spec 标签，结合这些标签，我们提出了 OrderedXP 算法，保证在一遍扫描 XML 数据流的基础上就给出查询所有的解。在今后的工作中我们将考虑怎样结合模式信息来对 XML 数据流上的 XPath 查询做各种优化。

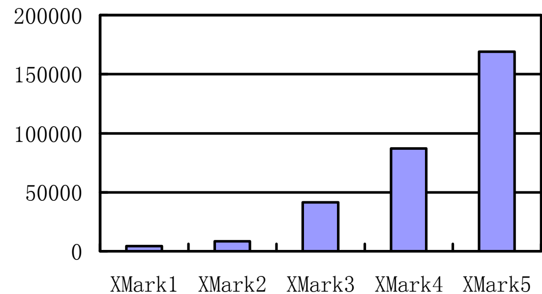

图 8

参考文献

[1] http://www.w3.org/TR/REC-xml/, 2006-4-10

[2] http://www.w3.org/TR/xpath, 2006-4-10

[3] http://www.w3.org/TR/xquery, 2006-4-10

[4] Y. Chen, S.b. Davidson, Y. Zheng. An Efficient XPath Query Processor for XML Streams. In: Proceedings of ICDE. Atlanta:IEEE Press, 2006. 79-79

[5] V. Josifovski, M. Fontou, A. Barta. Querying XML streams. In VLDB Journal, 2005, 14(2):197-210

[6] C. Barton, P. Charles, M. Fontoura, V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. In: Proceedings of ICDE. Bangalore:IEEE Press, 2003. 455-466

[7] A.K. Gupta, D. Suciu. Stream Processing of XPath Queries with Predicates. In: Proceedings of SIGMOD. San Diego:ACM Press, 2003. 419-430

[8] A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White. Towards Expressive Publish/Subscribe Systems. In: number 3896 in Lecture Notes in Computer Science. Munich: Springer-Verlag, 2006. 627-644

[9] http://xerces.apache.org, 2006-4-10

谢敏，男，1983 年生，硕士研究生，研究方向：XML 数据库。

王小锋，女，1980 年生，硕士研究生，研究领域：XML 数据库。

张新，男，1983 年生，硕士研究生，研究领域：XML 数据库。

孟小峰，男，1964 年生，教授（博导），研究领域：Web 数据管理，XML 数据库，移动数据管理。

周军锋，男，1977 年生，博士研究生，研究方向：XML 数据库。