XML 流上的 XQuery 前缀共享查询*)

孙东海[1] 张 昱[1,2] 吴晓勇[1]
(中国科学技术大学计算机科学与技术系 合肥 230027)[1]
(中国科学院计算机科学重点实验室 北京 100080)[2]

摘 要 如何在 XML 流上高效地执行大量复杂 XQuery 查询是当今研究的热点之一。在数据选择分发等应用中,还希望在解析流的同时尽早地输出查询结果。为此,本文将 XQuery 查询的路径导航和结果构造两个阶段分别运行于服务器、客户机两端。导航阶段针对 XQuery 查询定义了扩展的基本 XSIEQ 机 E-XSIEQ(Extended XML Stream Query with Immediate Evaluation),它是一种被索引化、基于栈的自动机。在 E-XSIEQ 机上设计应用了 TreeBuf(Tree Buffer)算法,它是一种树型提升缓冲的查询算法,算法使用了前缀共享计算的技术,能高效处理 XQuery 查询,而且能优化 XPath 查询。实验证明了 TreeBuf 算法的高效性。

关键词 XQuery 查询,XPath,XML 流

A Sharing Prefix XQuery Query Engine over XML Stream

SUN Dong-Hai ZHANG Yu WU Xiao-Yong
(Department of Computer Science & Technology, University of Science & Technology of China, Hefei 230027)
(Key Lab. of Conputer Seience, Chinese Academy of Science, Beiging 100080)[2]

Abstract Much research has been done in evaluating massive complicated XQuery set over an XML stream efficiently. In some applications especially data selecting and distributing, it is further required to output the results while parsing XML stream for higher system efficiency. An XQuery query process is divided into two stages: path navigation and result construction and this two stages separately run at two sides of system that has the Client/Server model. In the former stage, an extended XSIEQ machine E-XSIEQ(Extended XML Stream Query with Immediate Evaluation) is defined, which is a kind of indexed automata based on stack. Moreover, TreeBuf(Tree Buffer) algorithm based on promoting tree buffer is put forward. It uses XQuery's sharing prefix characteristic and can both process XQuery query efficiently and optimize XPath query. Experimental results show that TreeBuf algorithm's performance is considerable.

Keywords XQuery query,XPath,XML stream

1 引言

随着 XML 的迅速发展和广泛应用,XML 数据的查询语言 XQuery 逐步成为最流行的查询语言之一,目前已是 W3C 的候选建议[9]。以 XQuery 为查询条件的 XML 流查询研究尚处于发展阶段,一些基于 XPath 的 XML 流查询引擎能支持简单的 XQuery 查询,如文[1,7]。还有一些专门的 XQuery 流查询项目[5,6],但支持力度有限,不能满足日益增长的 XQuery 查询需求。本文旨在提供一种能够支持复杂 XQuery 的高效流查询引擎。

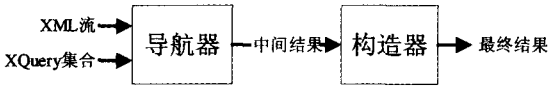


图 1 XQuery 查询系统简图

一次典型的 XQuery 查询过程可分为路径导航和结果构造两个阶段,分别运行于导航器和构造器(图 1)。导航器提取 XQuery 集合中的 XPath 式作为条件,查询 XML 流,得到的结果作为中间结果;在流终止后,构造器将中间结果按要求进行过滤、排序和连接,并将最终结果输出。

XQuery 查询的结果构造在得到所有的中间结果后通过过滤和排序获得,所以不能及时地输出结果。如果查询服务器同时承担导航和结果构造,则将导致:一方面必须在服务器缓存中间结果直到结果构造,系统缓存开销很大;另一方面在收集到全部中间结果后,执行结果构造会导致同时计算和输出的量比较大,造成系统和网络负荷过重。

为此可采用客户/服务器模式,服务器负责导航,客户端负责本地的结果构造(如图 2),从而使两端计算负载平衡,同时服务器可以尽早输出中间结果。

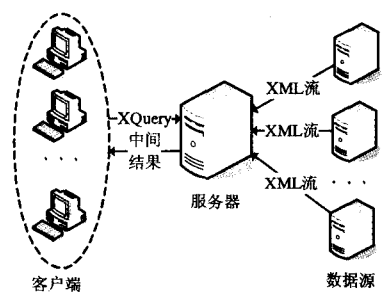


图 2 系统应用场景

*)Supported by the National Nature Science Foundation of China under Grant No. 60673126,(国家自然科学基金);the Foundation of Laboratory of Computer Science,the Chinese Academy of Science under Grant No. SYSKF0502,(中国科学院计算机科学重点实验室开放课题基金)。**孙东海** 硕士研究生,主要研究领域为 XML 数据处理;**张 昱** 博士,副教授,主要研究领域为 XML 数据处理、程序设计语言理论和实现技术;**吴晓勇** 硕士研究生,主要研究领域为 XML 数据处理。

服务器剥离构造器后，性能取决于导航器，所以需要一个高效的查询算法。文[1～4]的算法主要用于 XPath 查询，没有针对 XQuery 查询的优化，文[5～7]没有给出复杂 XQuery 查询的解决方案。本文关注的重点是导航器的性能，针对 XQuery 的特点，提出了前缀共享计算的 XQuery 查询算法：当变量之间的导航路径有相同的前缀时，实现前缀中谓词以及前缀所关联的缓冲区计算的共享。

本文的贡献如下：

1）提出一种前缀共享计算的查询算法 TreeBuf。TreeBuf 在支持 XQuery 查询的同时，还能实现 XPath 查询的谓词共享计算，提高 XPath 查询的性能。

2）利用 TreeBuf 算法实现高效的 XQuery 流查询原型系统。

3）通过大量的实验，给出影响 XQuery 查询性能的因素，同时证明 TreeBuf 算法对 XPath 查询性能的提升。

2 预备知识

2.1 XQuery

XQuery 语言的核心是 FLWOR 表达式和内嵌 XPath 式。XQuery 查询中路径导航的工作由内嵌 XPath 式完成。FLWOR 表达式由 for、let、where、order、return 共五个子句组成：for、let 子句中定义了若干变量，每个变量关联一个 XPath 式，实现导航功能；where 子句实现选择过滤功能；order 子句实现排序功能；return 子句实现结果构造。FLWOR 表达式允许嵌套。每个 XQuery 式必有一个顶层 FLWOR 表达式和任意多个嵌套 FLWOR 表达式。

定义 2.1 对任意作用在单文档上的 FLWOR 表达式，有且仅有一个根变量，对应文档的根结点。出现在 for、let 子句中的变量绑定称为变量定义，for 子句中定义的变量称为 for 变量，let 子句中定义的变量称为 let 变量，在 where、order、return 子句中出现变量称为对变量的引用，被引用的变量称为结果变量。每个变量的关联 XPath 式分为两部分：前缀变量和路径。for 变量关联的 XPath 式称为 forXPath 式，let 变量关联的 XPath 式称为 letXPath 式，结果变量关联的 XPath 式称为结果 XPath 式。

定义 2.2 对一个 XPath 式 P，P 的主 XPath 式是指去除 P 中所有谓词后剩余的部分。主 XPath 式匹配的 XML 结点称为候选结果。

例 1 本文中所有示例用到的 XQuery 表达式 xq 描述如下：

```
for $a in doc(abcd.xml)//a, $b in $a/b
let $c := $a/c, $d := $b/d
order by $b
return <result>
          {$a, $b, $c, $d}
       </result>
```

xq 中 a、b 是 for 变量；c、d 是 let 变量；a、b、c、d 是结果变量。a^1 在 for 子句中定义，在 return 子句中被引用。b 的关联 XPath 式是 $a/b，其中 $a 是前缀变量，/b 是路径。

本文所讨论的 XQuery 查询系统支持具有以下特性的 XQuery[9] 子集：单文档上的多 XQuery 并发查询；FLWOR 表达式中 let、where、return 子句的多层嵌套；带 and、or 连接词的 where 子句；多关键字排序的 order 子句和直接元素构造器的 return 子句。XPath 特性的支持力度同文[8]。

1 文中使用的 XQuery 变量在没有歧义时都忽略了变量符号 $。

3 XSIEQ 系统

XSIEQ[8] 是本实验室的在研项目，旨在研发高效的 XML 流查询引擎。目前 XSIEQ 已支持复杂的 XPath 查询和本文介绍的 XQuery 查询，分别记为 XSIEQ-XP 和 XSIEQ-XQ。

3.1 XSIEQ-XP

XSIEQ-XP 系统的框架是基本 XSIEQ 机。它将各查询 XPath 式中的主 XPath 式和谓词中的 XPath 式提取出来，利用前缀共享的方法增量式地构造为一个 NFA；然后对 NFA 中的以下特殊状态进行类型标记并添加索引。

定义 3.1 对于基本 XSIEQ 机中的任意 NFA 状态 s，如果它匹配主 XPath 式，则称为结果状态，并在该状态上保存其对应的各主 XPath 式列表 LR(s)。如果 s 匹配谓词中的 XPath 式，则称为叶子状态，并在该状态上保存它关联的各谓词列表 LP(s)。如果 s 是某 XPath 式对应的主路径上具有的、分支到谓词中 XPath 式的状态，则称为分支状态，第一个分支到谓词的状态又称最左分支状态，在分支状态保存从该状态分支出去的谓词列表 LB(s)。

在基本 XSIEQ 机上可以实施不同的查询算法，如 PBuf[8]。PBuf 算法的特点体现在候选结果的缓冲机制上。对于一个 XPath 式 p，若它含有 m 个带谓词的位置步，自左至右依次标记为 ls_0，…，ls_{m-1}，则 p 的缓冲区含有 m 层，第 i ($0\leqslant i\leqslant m-1$)层保存的候选结果满足 ls_i 到 ls_{m-1} 中的所有谓词。当 XSIEQ 机运行回溯到 ls_{m-1} 对应的分支状态时将候选结果添加到缓冲区的第 $m-1$ 层；回溯到 ls_i ($0\leqslant i<m-1$)对应的分支状态时进行缓冲区的层次提升；在到达第 0 层缓冲区后得到确认。

3.2 XSIEQ-XQ

FLWOR 表达式中 for 变量不仅具有导航功能，同时控制着查询结果的结构化输出，因此 forXPath 式和结果 XPath 式的功能不同，对应在自动机运行时的操作也不同。于是 XSIEQ-XQ 使用对基本 XSIEQ 机进行了扩展的框架，记为 E-XSIEQ 机，它修改了基本 XSIEQ 机的状态类型定义及索引。

定义 3.2 对于 E-XSIEQ 机中的任意 NFA 状态 s，如果 s 是匹配结果 XPath 式的主 XPath 式，则称为结果状态。如果 s 是匹配 forXPath 式的主 XPath 式，则称为 for 变量绑定状态，在 for 变量绑定状态上保存其对应的主 XPath 式列表 LF(s)。

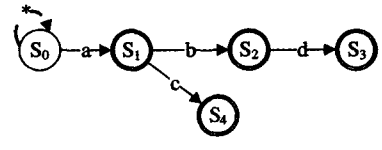


图 3 xq 提取的 XPath 式构造的 E-XSIEQ 机

图 3 是为 XQuery 表达式 xq 建立的 E-XSIEQ 机，其中状态用圆圈表示，双圈表示结果状态，S_1，S_2 是 for 变量绑定状态；弧代表状态转换，其中虚线表示子孙轴类型的转换，实线表示孩子轴类型的转换，弧上的标记表示节点测试。状态 S_0 因包含子孙轴类型的转换弧而具有到自身的转换弧。

XSIEQ-XQ 中导航器的目标是导航得到所有变量的结果。变量的完整导航信息不仅包括关联 XPath 式的路径，还

包括前缀变量，因为变量之间存在依赖关系。

定义 3.3 对于一个 FLWOR 表达式中的某一变量 v_j，如果 v_j 的定义为 for v_j in v_i+p_j 或者 let $v_j:=v_i+p_j$，其中“+”表示串联，p_j 是路径，则称 v_j 直接依赖于 v_i，记为 $v_i \xrightarrow{p_j} v_j$。

依赖关系满足传递性，即

$$v_i \xrightarrow{p_j} v_j, v_i \xrightarrow{p_k} v_k \Rightarrow v_i \xrightarrow{p_{jk}} v_k \text{ (transitivity rule)}$$

其中 p_{jk} 是串联 p_j 和 p_k 得到的 XPath 式，称 v_k 传递依赖于 v_i。任意变量 v_k 传递依赖于根变量，从根变量到 v_k 的路径称为 v_k 的导航路径。一般地，将直接依赖和传递依赖统称为变量绑定依赖。

根据上述定义，变量的完整导航信息可以由导航路径描述。如果在导航前通过解析 XQuery 脚本取得变量的导航路径，则可以重用 PBuf 算法完成导航功能。在 PBuf 算法里缓冲区的层次数和导航路径中含谓词的位置步的个数相同。由于变量之间存在依赖关系，若为每个变量的导航路径设置一个缓冲区必然会导致不同缓冲区的若干层之间有信息和操作的冗余。

根据变量之间的依赖关系，变量的导航路径被自然地分成若干段路径，如果为每个变量设置的缓冲区对应的只是关联 XPath 式的路径，那么将大大减少缓冲区之间的重复计算，但此时不能应用 PBuf 算法，为此，我们设计了如下的 TreeBuf 查询算法。

4 基于 TreeBuf 算法的查询

4.1 数据结构

TreeBuf 算法需要两种数据结构：变量依赖树和原子表集合，来支撑路径导航。它们在导航前由系统解析 XQuery 式获得。XQuery 集合中的所有变量和关联 XPath 式组织在变量依赖树中作为输入，导航得到的中间结果组织在原子表中。

4.1.1 变量依赖树

一个变量有且仅有一个前缀变量同时又可以作为多个变量的前缀变量，所以在一个 FLWOR 表达式中定义的各变量根据依赖关系组织形成一棵变量依赖树。在单文档查询中，查询的 XQuery 集合中所有变量在同一棵树中。

定义 4.1 变量依赖树 $VarTree=(V,E,P)$。V 是结点集合，E 是边集合，P 是边标记集合。$\forall v_i \in V$，v_i 表示变量，根结点表示根变量 r；$e=(v_i,v_j)\in E$ 表示直接依赖 $v_i \xrightarrow{p_j} v_j$，其中 $p_j \in P$ 表示 v_j 的关联 XPath 式的路径。

由例 1 的查询 xq 构造的 VarTree 如图 4，树中变量 a、b 之间，b、d 之间具有直接依赖关系；a、d 之间具有传递依赖关系；r 是根变量。

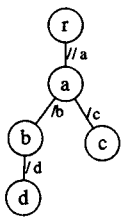


图 4 例 1 的变量依赖树

在变量依赖树中，任意一个结点的孩子结点之间不会存在相同的导航路径，因为在为查询的 XQuery 集合构造变量依赖树时，如果变量的导航路径每一段都相同，则对应树中的同一个结点。构造变量依赖树后，查询只需要收集树中各结点的结果。因为变量在树中的结点共享，使得查询的变量集合大大减小。

4.1.2 原子表

定义 4.2 一次查询中变量关联的 XPath 式匹配的值称为绑定序列。以单个变量作为属性，变量的绑定序列作为元组序列的表结构，称为变量表。对任意两个变量值 val_1、val_2，如果 val_1、val_2 在 XML 文档中是父子元素的关系，则称 val_1、val_2 之间具有上下文关系，变量表之间根据值的上下文关系进行的连接称为上下文连接。

定义 4.3 原子表的属性由变量依赖树中的 $n(n>0)$ 个变量组成，每个原子表有且仅有一个 for 变量属性，作为主码，其它属性列由直接依赖于该 for 变量的 let 变量组成，表的元组由各变量的变量表连接而成。两张原子表之间具有直接依赖关系当且仅当它们的主码之间具有直接依赖关系。如果原子表 T_i，T_j 之间满足 $T_i \rightarrow T_j$，则对 T_i 的每个元组 tup，T_j 中都有一组元组 S_{tup} 与之存在上下文关系，称 S_{tup} 为原子表 T_j 的一个分组。tup 和 S_{tup} 中的每一个元组之间存在上下文索引。

在例 1 的 xq 中，为变量 a、b、c、d 可以构造变量表，变量 a、c 和变量 b、d 分别组成原子表 T_{ac} 和 T_{bd}。T_{ac}、T_{bd} 之间具有直接依赖关系。

4.2 TreeBuf 查询算法

TreeBuf 查询算法在 E-XSIEQ 机上的状态转换、谓词计算和候选结果收集过程与基于 PBuf 的 XSIEQ 机[8]相同，算法的独特部分体现在候选结果的缓存和组织上，下面着重讨论这两部分。

4.2.1 缓冲树

TreeBuf 算法的缓冲区实现和 PBuf 略有不同，对变量 v 的关联 XPath 的路径 p 而言，如果 p 有 m 个带谓词的位置步，则 p 的缓冲区有 $m+1$ 层，第 $i(0\leqslant i\leqslant m-1)$ 层保存的候选结果满足 ls_i 到 ls_{m-1} 中的所有谓词，第 m 层保存未进行谓词计算的候选结果。

各变量对应的缓冲区根据变量之间的依赖关系，同样组织成树型结构，称为缓冲树。

定义 4.4 给定一棵变量依赖树 $VarTree=(V,E,P)$，其对应的缓冲树 $BufTree=(VP,EP,B)$，其中 VP 是结点集合，EP 是边集合，B 是结点标记集合。$\forall v_i \in V$，$\exists vp_i \in VP$，vp_i 表示变量 v_i 的关联 XPath 式的路径；$b_i \in B$ 表示 vp_i 的缓冲区；$BufTree$ 中的根结点 rp 表示 $VarTree$ 中根变量 r 的路径，它实际并不存在，是一个虚结点，rp 没有缓冲区。$\forall ep=(vp_i,vp_j)\in EP$，$\exists e=(v_i,v_j)\in E$ 且 $v_i \xrightarrow{p_j} v_j$。

引入缓冲树后变量 v 的导航路径就是从缓冲树的根结点到 v 对应结点中各结点路径的串联。

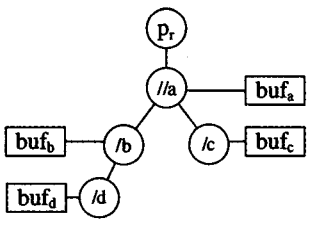


图 5 例 1 的 BufTree

图5是由例1xq的变量的缓冲区组成的缓冲树$BufTree$，它有4个缓冲区buf_a、buf_b、buf_c、buf_d，由于4个XPath式都没有谓词，因此缓冲区都只有1层，p_r没有缓冲区。

4.2.2 算法的缓冲机制

在XSIEQ-XP中，若XPath式p含有谓词，则当XSIEQ机运行回溯到p的最左分支状态时，一定能确认在此前收集到的p的候选结果是否匹配。

在XSIEQ-XQ中，对XQuery查询中的任意变量v，关联XPath式的路径为p，p含有m个带谓词的位置步，v的候选结果需要满足导航路径上的所有谓词才能确认，所以候选结果的确认时机是E-XSIEQ机运行回溯到p在缓冲树中第2层的祖先结点的最左分支状态。于是对v的一个候选结果而言，提升操作分为两种：一个缓冲区内部相邻层之间的提升；缓冲树中父子缓冲区之间的提升。

根据定义4.4和候选结果缓存过程的描述，TreeBuf算法在缓冲树上的基本操作有：

1) add操作。E-XSIEQ机运行回溯到p的结果状态时，则调用add操作，将相关的候选结果项添加到p的缓冲区中的第m层；

2) promote操作。当回溯到p的结果状态时(设回溯到的XML节点编号为xid)，如果p在缓冲树中有孩子结点q，调用promote操作将q的缓冲区第0层中XML节点编号大于xid的所有候选结果项提升到p的缓冲区第m层，同时清除q的第0层；回溯到ls_i($0\leqslant i\leqslant m-1$)对应的分支状态时，调用promote操作将检查$ls_i$中各谓词的状态，若均为真则将缓冲区的第$i+1$层中XML结点编号大于xid的各XPath式的候选结果项提升到第i层，同时清除第$i+1$层；

3) accept操作。当回溯到p的第2层祖先结点s的最左分支状态时，在执行promote操作后，可调用accept操作输出s的缓冲区中第0层的数据项，即最终匹配的数据项，包括s以及s的所有子孙结点的结果。

4.2.3 缓冲区的结果组织方式

缓冲树中任意结点需要缓存以该结点为根结点的子树的所有结点的候选结果，在缓冲树中let变量结点都是叶子结点，所以直接缓存自身的结果；for变量结点的孩子有for变量结点和let变量结点两种，由于for变量和直接依赖于它的let变量的值之间是一对一的关系，因此使用原子表组织for变量结点和它的let变量孩子结点的值，而for变量和for变量的值之间是一对多的关系，所以在for变量结点上使用上下文索引关联其for变量孩子结点的值。

算法中用于结果组织的基本操作为：

combine操作。p是for变量v关联的XPath式的路径，以v为主码的原子表为T，p的孩子结点中forXPath式的路径为$fp_1,\cdots,fp_n$($n\geqslant 0$)、letXPath式的路径为$lp_1,\cdots,lp_m$($m\geqslant 0$)，当E-XSIEQ机运行回溯到p的结果状态时，对p执行add操作并且执行promote操作，提升所有孩子结点的缓冲区第0层结果的同时，新建T的一个元组u：u中主码值为p的add操作值，u的其余分量值为从$lp_1,\cdots,lp_m$的缓冲区得到的promote值，u的上下文索引指向从$fp_1,\cdots,fp_n$的缓冲区得到的promote值。

使用原子表组织结果后系统在导航阶段完成变量表到原子表的连接，整体性能得到了提升。因为流查询的顺序扫描特性使变量之间上下文关系在导航阶段的判断所需的时间复杂度是$O(l)$，提前连接不会增加导航时间，同时减少了结果构造时间。

4.3 运行示例

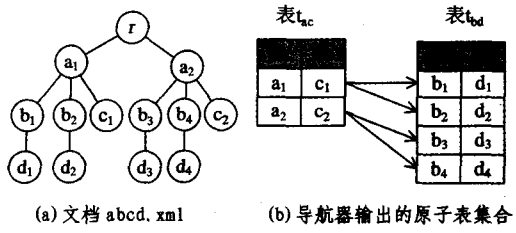
TreeBuf算法在缓冲区内的提升机制和PBuf算法相同，所以演示过程忽略TreeBuf算法中缓冲区内的提升机制，突出缓冲区之间的提升机制和候选结果的组织方式。于是在例1中给出的内嵌XPath式都没有谓词。

图6(a)是演示用到的XML片段abcd.xml的树型表示，abcd.xml中元素的下标不是标记的一部分，仅仅表示几个同名元素在文档中的出现顺序。

表1给出了例1在abcd.xml上的运行过程中缓冲区的变化、候选结果的组织方式和时机。

在endElement(d_1)中回溯到/d的结果状态，执行add操作。将d_1加入buf_d。在endElement(b_1)中回溯到/b的结果状态同时也是for变量绑定状态。新建元组u_{bd1}，对buf_d执行promote操作后d_1加入buf_b，同时执行add操作，将b_1加入buf_b。b_1、d_1作为u_{bd1}的两个分量，buf_b保存对u_{bd1}的引用。在endElement(d_2)、endElement(b_2)中执行类似操作。在endElement(c_1)中执行add操作，将c_1加入buf_c。在endElement(a_1)中回溯到//a的结果状态同时也是for变量绑定状态，对buf_b、buf_c执行promote操作，对buf_a执行add操作，新建元组u_{ac1}，u_{ac1}的两个分量是a_1、c_1，u_{ac1}的索引指针指向u_{bd1}、u_{bd1}，因为此时已经满足执行accept操作的所有条件，所以可以调用accept操作，输出buf_a中的第0层的数据项。后面的操作类似。

运行后得到的原子表集合如图6(b)，此集合将作为构造器的输入。



(a) 文档abcd.xml　　(b) 导航器输出的原子表集合

图6 示例的输入文档和输出集合

5 TreeBuf算法在XPath查询上的应用

在XSIEQ-XP中PBuf算法让每个XPath式p有自己的缓冲区并独立计算谓词，当XPath式之间有相同的前缀位置步序列时，PBuf算法就会存在大量重复的计算。为此提出了XPath式的前缀共享计算。

定义5.1 对于XPath式集合$P=\{p_i \mid i=1,2,\cdots,n,n\geqslant 2\}$，$p_i$的位置步从左至右是$l_i^1,\cdots,l_i^{m_i}$($m_i>0$)，如果$P$中的所有$p_i$($1\leqslant i\leqslant n$)有相同的前缀位置步序列$l^1,\cdots,l^r$($0<r\leqslant\min(m_i)$，$i=1,\cdots,n$)，则构造变量$pre$，$pre$关联XPath式的前缀变量是根变量$r$，导航路径为$p_{pre}$，$p_{pre}$的位置步序列为$l^1,\cdots,l^r$，同时构造变量$v_i$关联$p_i$，并将$p_i$改写成$pre+p'_i$，$p'_i$的位置步序列是$l_i^{r+1},\cdots,l_i^{m_i}$，对$p_{pre}$的计算称为集合$P$的前缀共享计算。

表 1　例 1 在 abcd.xml 上运行时候选结果的缓存和组织过程

	XML	$\langle a_1\rangle$	$\langle b_1\rangle$	$\langle d_1\rangle$	$\langle/d_1\rangle$	$\langle/b_1\rangle$	$\langle b_2\rangle$	$\langle d_2\rangle$	$\langle/d_2\rangle$	$\langle/b_2\rangle$	$\langle c_1\rangle$	$\langle/c_1\rangle$	$\langle/a_1\rangle$
缓冲区	buf_a												u_{ac1}
	buf_b					u_{bd1}	u_{bd1}	u_{bd1}	u_{bd1}	u_{bd1}，u_{bd2}	u_{bd1}，u_{bd2}	u_{bd1}，u_{bd2}	
	buf_c											c_1	
	buf_d				d_1				d_2				
元组创建时机	u_{ac}												$u_{ac1}(a_1,c_1)$
	u_{bd}					$u_{bd1}(b_1,d_1)$				$u_{bd2}(b_2,d_2)$			
元组间的索引创建时机													U_{ac1} / u_{bd2} u_{bd1}
	XML	$\langle a_2\rangle$	$\langle b_3\rangle$	$\langle d_3\rangle$	$\langle/d_3\rangle$	$\langle/b_3\rangle$	$\langle b_4\rangle$	$\langle d_4\rangle$	$\langle/d_4\rangle$	$\langle/b_4\rangle$	$\langle c_2\rangle$	$\langle/c_2\rangle$	$\langle/a_2\rangle$
缓冲区	buf_a												u_{ac1}，u_{ac2}
	buf_b					u_{bd3}	u_{bd3}	u_{bd3}	u_{bd3}	u_{bd3}，u_{bd4}	u_{bd3}，u_{bd4}	u_{bd3}，u_{bd4}	
	buf_c											c_2	
	buf_d				d_3				d_4				
元组创建时机	u_{ac}												$u_{ac2}(a_2,c_2)$
	u_{bd}					$u_{bd3}(b_3,d_3)$					$u_{bd4}(b_4,d_4)$		
元组间的索引创建时机													U_{ac1} U_{ac2} / u_{bd2} u_{bd1} u_{bd4} u_{bd3}

例 2　XPath 查询中有 3 个 XPath 式，p_1:/a [c]/b [e]/d；p_2:/a [c]/b [e]/d [g]/f；p_3:/a [c]/b [e]/d/f。p_1、p_2、p_3 有相同的前缀 p_{pre}:/a [c]/b [e]，由变量 pre 关联，p'_1、p'_2、p'_3 分别为/d、/d [g]/f、/d/f，变量 v_i($1\leqslant i\leqslant 3$)关联 $pre+p'_i$，对 p_{pre} 的计算就是 p_1、p_2、p_3 的前缀共享计算。

前缀共享计算将 XPath 式改写成类似于 XQuery 查询中的分段结构，所以对改写后 XPath 式可以很自然地应用 TreeBuf 算法进行查询。

应用 TreeBuf 算法后，所有变量和 XQuery 查询中 for 变量有相同的缓冲区操作和候选结果组织方式。和 PBuf 算法相比，TreeBuf 算法因为前缀共享计算而提高了查询的时空效率，同时没有降低系统的实时性，结果的输出时机仍然相同。

6　实验结果与分析

笔者用 Java 实现了基于 TreeBuf 算法的 E-XSIEQ 机，并与基于 PBuf 算法的 XSIEQ 机进行时间性能对比测试。因为 TreeBuf 算法和 PBuf 算法适用的 XPath 集合不同，TreeBuf 适用于前缀共享的 XPath 式集合，PBuf 适用于普通的绝对 XPath 式集合，为了保证两种算法输入的查询集合的一致性，实验中 TreeBuf 算法用到的 XQuery 集合 Q_1 来源于本实验室实现的一个 XQuery 查询生成器，而 PBuf 算法用到的查询集合是从 Q_1 中提取内嵌 XPath 式经过转化后得到的绝对 XPath 式集合 Q_2，Q_1、Q_2 具有相同的查询结果。因为测试重点是系统的查询性能，所以实验主要选择了 XPath 的各种特性组合，没有关注 FLWOR 的特性组合。实验考核的因素包括 XPath 式中的普通谓词概率 p_b，存在性谓词概率 p_e，通配符“*”的概率 w 和子孙轴“//”的概率 d 等。实验平台为 Windows XP 操作系统，CPU 为 P4，主频为 1.6G，内存为 256M。实验用到的 XML 文件使用 XMark[10] 生成，大小是 1M。

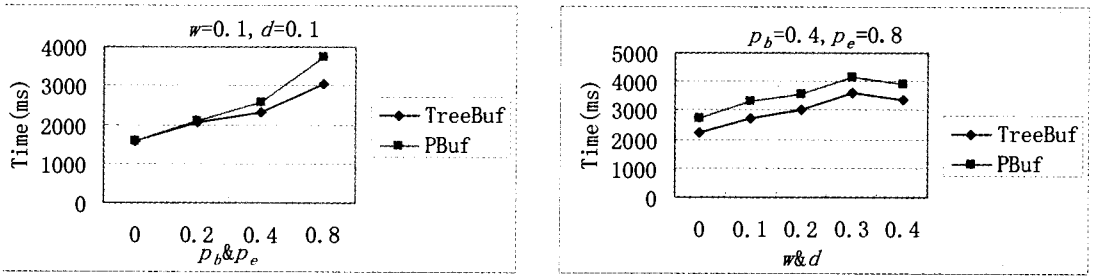


图 7　查询时间比较

图 7 为实验的部分结果，实验中 XQuery 的个数是 100，每个 XQuery 中包含 10 个 XPath 式。从图中可以看出在 XPath 特性相同情况下，TreeBuf 算法的查询时间总体上要快于 PBuf，根据文[8]中的结论，PBuf 算法性能优于 YFilter，所以在 XQuery 查询中使用 TreeBuf 算法的查询性能是较高的；在固定通配符、子孙轴概率后，随着谓词概率的增加，TreeBuf 和 PBuf 的查询时间差增加，因为谓词的增加使得 TreeBuf 算法的共享计算相应增加；固定 $p_b=0.4$、$p_e=0.8$ 后，w 和 d 的变化不会改变两种算法的时间比率，因为两种算法使用的状态转换机制相同；在 w、d 的各种概率组合下 TreeBuf 算法普遍要比 PBuf 算法快 15% 以上，说明 TreeBuf 算法共享计算的优化效果是稳定的。

7　相关工作介绍

FluXQuery[5] 应用于缓存有限的情况，所以系统致力于尽可能地减少查询缓存开销。FluXQuery 在 XQuery 的基础上定义了一套内部的查询语言 FluX，运行时系统首先将 XQuery 转换成 FluX 再进行查询。FluX 语言提供了比 XQuery 更多的空间优化机会，但因此要牺牲支持力度，不支持 FLWOR 表达式中的 let、order 子句以及 XPath 式内的“*”和“//”，当查询变得复杂时 FluXQuery 不再适用。Raindrop[6] 继承于 Rainbow[11]，Rainbow 是一个数据库查询

引擎。Raindrop因此使用了代数与自动机相结合的查询技术，并运用了数据库中的概念将系统分为四层：语义层、逻辑层、物理层和执行层，使得系统的优化机会更多，但是Raindrop中的代数特征使得系统相对其它流查询引擎占用更多的空间，不利于数据量很大的查询。TurboXPath[7]使用解析树取代传统的自动机作为查询的核心数据结构，有较高的查询效率。同时它还可以支持复杂的XPath式，除了文[8]的XPath特性外，还能支持反向轴，但是对XQuery的支持力度关注不够，因而不能支持复杂XQuery，如order子句。

结论 本文提出的TreeBuf算法是对文[8]中PBuf算法的继承，拥有PBuf的各种优点，也是对PBuf算法的改进。TreeBuf算法可以同时应用在XQuery查询和XPath查询上，并且通过前缀共享计算提高查询效率。

目前所实现的XQuery查询系统也存在一些不足，如对XQuery的支持力度仍然不够完善，不支持动态元素构造器、聚集函数等等。尽管在第5节说明了TreeBuf算法对XPath查询的改进，但是对于如何提取绝对XPath式的公共前缀，还没有成熟的算法。对系统支持力度的进一步扩展和寻找高效的前缀查找算法将作为下一步工作。

参考文献

1 Diao Yanlei. Altinel M, Franklin M J, et al. Path sharing and predicate evaluation for high-performance XML filtering. ACM Transations on Database Systems, 2003, 28(4): 467～516. http://yfilter.cs.berkeley.edu/code_release.htm
2 Chan Chee Yong, Felber P, Garofalakis M N, et al. Efficient Filtering of XML Documents with XPath Expressions. VLDB Journal, Special Issue on XML, 2002, 11(4): 354～379
3 Green T J, Miklau G, Sucia D. Processing XML streams with deterministic automata and stream indexes. ACM TODS, 2004, 29(4)
4 Gupta A, Suciu D. Stream Processing of XPath Queries with Predicates. In: SIGMOD, 2003. 419～430
5 Koch C, Scherzinger S, Schweikardt N, et al. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In: Proc. VLDB, 2004. 228～239
6 Su Hong, Rundensteiner A, Mani M. Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams. VLDB, 2004. 1293～1296
7 Josifovski V, Fontoura M, Barta A. Querying XML streams. VLDB, 2005, 14(2): 197～210
8 张昱，吴年．一种逐层提升缓冲的XML流查询自动机．小型微型计算机系统(已录用)
9 http://www.w3.org/TR/xquery/
10 Schmidt A, Waas F, Kersten M, et al. XMark: A Benchmark for XML Data Management. In: Proc. VLDB, 2002. 974～985
11 Zhang Xin, Dimitrova K, Wang Ling, et al. Rainbow: Multi-XQuery Optimization Using Materialized XML Views. In: Proc. ACM, SIGMOD, 2003. 671

(上接第95页)

时刻(t)	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
高进程状态	s_0^H	s_0^H	s_1^H	s_2^H	s_2^H	s_3^H			
高进程共享属性(direction,position)	⊥,0	⊥,0	⊥,0	⊥,0	⊥,0	down,53			
低进程状态	s_0^L	s_1^L	s_2^L	s_3^L	s_4^L	s_5^L	s_6^L	s_7^L	s_8^L
低进程共享属性(direction,position)	⊥,0	⊥,0	⊥,0	up,55	up,55	⊥,⊥	down,52	down,52	down,58

由此，根据上面表格我们可以清楚地看到：高进程状态 s_3^H 能够改变该共享客体磁臂 O 的属性；低进程状态 s_4^L，s_6^L 能够察觉这一改变，并且低进程状态 s_4^L，s_6^L 根据自身前后状态发生的变化能够察觉共享客体属性的变化。即低进程Low可以感知到磁臂在执行高进程High请求时是磁头的移动方向是down，根据电梯调度算法和高低进程之间的通信是按照事先约定好的编码方法进行通讯的，所以此时低进程可以感知到刚才高进程High请求的是53号柱面，从而泄露一个比特的信息。

隐通道传递信息的方法是感知型的，这是一种逻辑操作，无需通过安全模型的检验。隐通道的这一特点，决定了传统的隐通道的分析方法，即在语法分析基础上展开的基于系统顶级描述或源代码的分析，无法发现这种隐通道问题。本文提出的基于操作语义的隐通道分析方法，可有效地解决这类问题。

参考文献

1 McHugh J. Covert Channel Analysis: A Chapter of the Handbook for the Computer Security Certification of Trusted Systems [R]. Portland State University, December 1995
2 Denning D E, Denning P J. Certification of Programs for Secure Information Flow. Communications of the ACM[J], 1977, 20(7): 504～513
3 Bell D E, LaPadula L J. Secure computer system: Unified exposition and MULTICS interpretation: [TechRep MTR-2997]. The MITRE Corporation, 1976
4 McHugh J. An information flow tool for Gypsy. In: Computer Security Applications Conference[C], ACSAC 2001. Proceedings 17th Annual Dec. 2001. 191～201
5 Goguen J A, Meseguer J. Security Policies and Security Models. In: Proceedings of the IEEE Symposium on Security and Privacy [C], Oackland, California, 1982. 11～20
6 Kemmerer RA. Shared resource matrix methodology: An approach to identifying storage and timing channels. ACM Trans on Computer Systems[J], 1983. 256～277
7 Kemmerer R A. Covert Flow Trees: A Visual Approach to Analyzing Covert torage Channels. IEEE Transactions on Software Engineering[J], 1991, 17(11): 1166～1185
8 Venkatraman B R, Newman-Wolfe R E. Capacity Estimation and Auditability of Network Covert Channels. In: Security and Privacy, IEEE Symposium[C], May 1995. 186～198
9 Wang C D, Ju Shiguang. Research on the methods of search and elimination in covert channel. In: Grid and Cooperative Computing [C], LNCS, PT 1 3032: 1 2004. 988～99
10 Goldschlag D M. Several Secure Store and Forward Devices. In: Proc. of the Third ACM Conference on Computer and Communications Security[C], New Delhi, India, March 1996. 129～137
11 Kang M H, Moskowitz I S. A pump for rapid, reliable, secure communication. In: 1st ACM Conference on Computer and Communications Security [C], Fairfax, Virginia, November 1993. 119～129
12 Dogu T M, Ephremides A. Covert information transmission through the use of standard collision resolution algorithms. Lect Notes Comput Sc, 2000, 1768: 419～433
13 Karger P A, Wray J C. Storage Channels in Disk Arm Optimization. In: IEEE Symposium on Security and Privacy, 1991. 52～61
14 Ju Shiguang, Song Xiaoyu. On the Formal Characterization of Covert Channel. Lecture Notes in Computer Science, 2004, 3309: 155～160
15 Trusted Computer System Evaluation Criteria[R]. Department of Defense, U.S.A. December 1985
16 Hu Wei-Ming. Reducing timing channels with fuzzy time. In: IEEE Symposium on Research in Security and Privacy, Oakland, California, May 1991. 8～20
17 Melliar-Smith P M, Moser L E. Protection against covert storage and timing channels. In: Proceedings 4th IEEE Computer Security Foundations Workshop, June 1991. 209～214
18 Eckmann S T. Eliminating Formal Flows in Automated Information Flow Analysis. In: Proceedings of the IEEE Symposium on Security and Privacy, 1994. 30～38
19 Son S H, Mukkamala R, David R. Integrating security and real-time requirements using covert channel capacity. IEEE T KNOWL DATA EN, 2001, 13(5): 862～862
20 陆汝钤．计算机语言的形式语义．科学出版社，1992
21 冯玉琳，李京，黄涛．对象语义理论和行为约束推理．计算机学报，1993, 16(11)
22 卿斯汉．高安全等级安全操作系统的隐蔽通道分析．软件学报，2004, 15(12)