M^2:一种有效的XPath求值方法

陈荣鑫[1,2] 廖湖声[1] 陈维斌[3] 叶文来[2]
(北京工业大学计算机学院 北京 100124)[1] (集美大学计算机工程学院 厦门 361021)[2]
(华侨大学计算机学院 泉州 362021)[3]

摘 要 XPath求值性能极大关系到XML应用程序的整体性能。提出的M^2(Matrix Match)方法基于节点关系矩阵查询。根据XML区间编码构建关系矩阵,通过查询原语序列的执行实现XPath表达式求值。M^2的导航式特点使得XPath的各种查询语义容易实现,包括反向轴操作和以谓词表达的分支查询。对应相同XML数据的关系矩阵可以被不同查询复用。关系构建和查询求值的过程中,基于循环的处理方式适合并行化优化。与现有XPath求值性能的比较结果表明,M^2是一种有效的求值方法。

关键词 XPath求值,XML编码,关系矩阵,查询原语
中图法分类号 TP311.13 **文献标识码** A

M^2:An Effective Method for XPath Evaluation

CHEN Rong-xin[1,2] LIAO Hu-sheng[1] CHEN Wei-bin[3] YE Wen-lai[2]
(College of Computer Science, Beijing University of Technology, Beijing 100124, China)[1]
(Computer Engineering College, Jimei University, Xiamen 361021, China)[2]
(Computer Science College, Huaqiao University, Quanzhou 362021, China)[3]

Abstract The performance of XPath evaluation greatly affects XML applications. This paper presented M^2 (Matrix Match) method based on node relation matrix query. Relation matrix is constructed from XML region encoding, while XPath evaluation is realized by execution of query primitive sequence. The navigation feature tends to comply with XPath semantics and support the implementation of reversed axis query and branching query expressed in predicates. The existing relation matrix from the same data can be reused by different queries. The loop processing style in matrix construction and evaluation is suit for parallelism. Experiments show M^2 is an effective XPath evaluation method.

Keywords XPath evaluation, XML encoding, Relation matrix, Query primitive

1 引言

随着网络的普及和网络服务的发展,XML作为信息交换标准和存储标准,在各个领域里广泛应用。XML是半结构化数据,用户对其进行查询和变换等操作需要通过特定的语言实现。通常XML的查询通过XQuery等查询语言实现;XML的变换通过XSLT等语言实现。XPath用于操作XML文档树获取满足条件的节点,是XQuery和XSLT等语言的重要组成部分,其求值性能直接关系到各种XML应用系统的整体性能。

XPath即XML路径语言,是一种用于对XML文档树中的节点寻址的语言[1],能实现对满足数据模型[2]的XML数据按约束条件进行查询求值。XPath已经成为W3C组织的推荐标准,被业界广为接受。目前存在各种XPath求值的实现方案,在满足查询语义[3]规范要求的同时,如何提高求值性能成为研究热点。典型的实现机制包括传统的导航式[4]方法,以及后来发展的各种基于查询模式匹配的Twig方法[5-7]。Twig方法可以高效解决部分查询问题,如含AD/PC的轴操作,但其对某些XPath语义的支持能力往往有限,比如复杂的谓词求值、反向轴操作等。导航式方法的优点是能很好地保持查询语义,容易发展成完备的查询代数,有利于查询优化。在许多优秀的XML查询引擎中,比如Galax[4],Qizxopen[8]和Saxon[9]等,XPath的求值方法采用导航式。

本文提出的M^2,其特点是把预计算的节点关系存储在关系矩阵中,查询时用低代价的关系查找替代关系计算,大大提高了查询性能。XML查询一般都是非改写操作,由于每个XML数据对应一个关系矩阵,存储的关系序列化后可以复用,以适应对相同数据不同查询的情况。此外,由于M^2方法在关系构建和查询过程中涉及数据循环处理,操作的数据对象容易划分,适合采用数据并行方式进行优化。其导航式特点便于实现XPath的各种查询语义,从而提高了实用性。

本文第2节介绍XPath求值的相关背景知识,比较Twig

到稿日期:2010-03-26 返修日期:2010-07-20 本文受福建省自然科学基金项目(2008J04005)和北京市自然科学基金项目(4082003)资助。
陈荣鑫(1975-),男,博士生,讲师,主要研究方向为软件自动化与数据库技术,E-mail:ch2002star@163.com;廖湖声(1954-),男,教授,博士生导师,主要研究方向为软件自动化;陈维斌(1957-),男,教授,主要研究方向为数据库技术;叶文来(1976-),男,讲师,主要研究方向为软件工程。

和导航式实现方法；第 3 节从关系存储设计、查询原语设计和查询原语组织 3 个方面阐述 M^2 设计方法；第 4 节用实验验证 M^2 方法的性能和可行性；最后总结并展望未来工作。

2 XPath 求值

XPath 求值就是根据节点关系条件定位 XML 树的节点的过程。路径表达式是 XPath 表达式的基本形式，描述了各个求值步骤的顺序。XPath 语法[3]上用'/'符号来区分操作步骤，每一步操作根据当前上下文节点或节点序列定位到 XML 树中的另一个节点或节点序列。每一步可包含 3 种成分：

- 轴操作：指定按某种节点关系遍历 XML 树；
- 节点测试：用于筛选节点名称和类型，带通配符号"*"表示不进行筛选；
- 谓词描述：或分支查询，可有 0 或多个，用于筛选节点的属性、子节点特征或位置。

这 3 种成分可以组合成复杂的 XPath 查询，控制 XML 树的遍历和节点选择。基本的节点求取轴操作包括以下几种，分别用两个字母代表括号内字串的内容：PA('parent'::)双亲，CH('child'::'或'/')孩子；AN('ancestor'::)祖先，DE('descendant'::'或'//')后代；PS('preceding-sibling'::)左兄弟，FS('following-sibling'::)右兄弟；AS('ancestor-or-self'::)祖先或自身，DS('descendant-or-self'::)后代或自身；PP('preceding'::)之前，FF('following'::)之后。以上是 5 对互为反向的操作，其中 PA，AN，PS，AS，PP 属于反向轴操作，其余为正向轴操作。此外，还有 SS('self'::)求取自身和 AT('attribute'::'或'@')求取属性这两个正向轴操作。在实际应用中，使用最频繁的操作是求 PA/CH 的 PC 关系求值和求 AN/DE 的 AD 关系求值。XPath 语法可以简化表示为

```
Path::= Step('/' Step)*
Step::= Axis Tag | Axis Tag ['Pred']
Axis::=PA | CH | AN | DE | PS | FS | AS | DS | PP | FF |
       SS | AT
Tag::= string | *
Pred::= Path | Pred 'or' Pred | Pred 'and' Pred | 'not' Pred
```

XPath 求值的功能可完善程度和性能高低很大程度上依赖于求值机制的选择。目前常见的两类 XPath 求值实现机制有 Twig 和导航式。Twig 方法一般采用某种编码（如区间编码）进行关系判别；用辅助结构如栈、链表等完成条件筛选和结果记录，最后进行查询结果枚举输出。目前有多种 Twig 算法如 TwigStack[5]，Twig²Stack[6]，TwigList[7]等。其优点是：一般能够非常高效地进行简单轴处理，如 PC，AD 查询，而且内存消耗较低。全局 Twig 查询可以有效避免各种耗时的连接操作。Twig 查询通过节点编码的关系判断，一般包含模式构建和结果枚举这两个步骤，较难以并行化优化；而且大都仅支持 XPath 简单的查询类型，对带内容的复杂谓词求值和反向轴操作的支持十分有限。Twig 方法往往构造较为复杂，功能可扩展性差，需要各种优化才能达到实用程度。导航式方法一般是根据由 XML 解析获得的 XML 树节点关系，每步用查询原语操作，向下一步传递筛选结果。假设 S1，S2 为 XPath 查询步，则导航查询的求值语义实现可以表示为 E[s1/s2]=for $dot in E[S1] return E[S2]。其中 E 表示在当前上下文环境中的求值函数，$dot 表示变量，求值是一个迭代处理过程。由于导航式方法容易实现各种 XPath 查询语义，有利于和完备的 XML 查询代数[10]整合，便于查询优化。然而存在执行效率往往较低的问题。

导航式能实现完备的 XPath 查询语义，容易满足实际应用需求，但仍需尽可能提高其执行性能。目前多核环境日益普及，并行化措施是提高性能的自然选择。我们提出的 M^2 方法基于导航式实现机制，所采用的数据结构容易进行数据划分，使得处理满足易并行条件，适合于并行化优化。

3 Matrix Match 方法

M^2（Matrix Match）设计的基本思路是把导航式查询中的关系计算转化为对可复用的关系矩阵的查询操作。采用矩阵存储关系编码，查询过程利用计算代价较低的循环匹配方法实现，避免了较复杂的节点关系计算，从而实现对 XPath 的高效求值。关系矩阵的构建占用很大一部分时间，由于只要 XML 数据不变，关系矩阵可复用，构建的代价就可以被摊薄，使整个查询方案确实可行。此外，基于矩阵的数据结构和循环匹配算法便于并行优化。M^2 方法包含的基本设计步骤有支撑关系查询的存储设计和查询原语设计，以及根据 XPath 查询语义进行原语组织。

3.1 关系存储设计

3.1.1 XML 节点编码与关系判断

XML 数据经过语法解析，获得每个节点的信息，节点信息可采用区间编码表示。典型的区间编码如 Zhang 编码[5]，记录每个节点起始位置、终止位置和在 XML 树中所在层次。一个节点的简单区间编码形如 NodeCode(nodeID, nodeType, nodeNameId, begin, end, level)。其中 nodeID 为节点 ID，是每个节点的唯一标示，反映了文档序，为遍历和排序所用；nodeType 表示节点的类型，通常包括元素类型、属性类型等；nodeNameID 是节点名字索引值，为了方便存储和查找，元素节点表示节点名 ID，属性节点表示属性名称 ID；begin，end 是节点的起始和终止位置，可以用文档中实际位置表示；level 表示节点在 XML 中的层次。XML 文档解析后，产生的节点信息为后续节点关系矩阵的建立提供数据来源。

图 1(a)是编码的示意，各圆圈内字符串字母是节点的标签名称，带有数字以区别不同节点。方括号外的数字是按文档序排列的节点 ID，里面的数据分别是节点开始位置、结束位置和在 XML 树中的层次。从编码可以判断节点间关系，如最重要的 PC，AD 关系的求值规则如下。

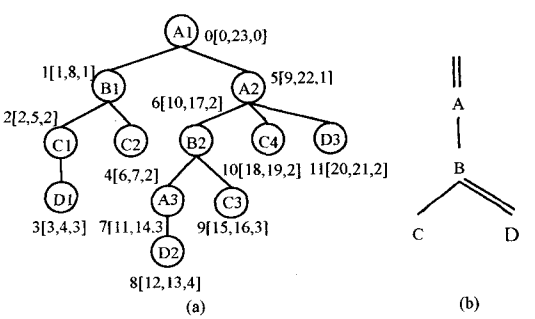

图 1 XML 编码和 XPath 查询

$E[U/child::V]=\{v \mid v \text{ is the child of } u, u\in U, v\in V\}$或

$E[U/attribute::V]=\{v \mid v \text{ is the attribute of } u, u\in U, v\in V\} \Leftarrow \text{beging}(u)<\text{begin}(V) \wedge \text{begin}(V)<\text{end}(u) \wedge \text{level}$

$(u)=\text{level}(V)_{-1}$；

$E[U/parent::V]=\{v|v \text{ is the parent of } u, u\in U, v\in V\}\Leftarrow \text{begin}(V)<\text{begin}(u)\wedge\text{begin}(u)<\text{end}(v)\wedge\text{level}(v)=\text{level}(u)_{-1}$；

$E[U/descendant::V]=\{v|v \text{ is the descendant of } u, u\in U, v\in V\}\Leftarrow\text{begin}(u)<\text{begin}(v)\wedge\text{begin}(v)<\text{end}(u)$；

$E(U/ancestor::V)=\{v|v \text{ is the ancestor of } u, u\in U, v\in V\}\Leftarrow\text{begin}(v)<\text{begin}(u)\wedge\text{begin}(u)<\text{end}(v)$。

用 $E[expr]$表示在当前环境中对 XPath 表达式求值，⇐右侧表达式是判断条件。而对于 preceding 和 following 的求值规则如：

$E[U/preceding::V]=\{v|v \text{ before } u \text{ in doc order}, u\in U, v\in V\}\Leftarrow\text{begin}(u)>\text{end}(v)$；

$E(U/following::V)=\{v|v \text{ after } u \text{ in doc order}, u\in U, v\in V\}\Leftarrow\text{begin}(v)>\text{end}(u)$。

兄弟关系无法从节点的 begin，end 和 level 这 3 个变量的比较直接推出。可以通过变换原表达式，用语义等价的表达式求取：

$E[U/preceding\text{-}sibling::V]$

$= E[let\ \$e:= U/self::node()\ \text{return}\ \$e/parent::node()/child::V\ [.<<\$e]]$；

$E[U/following\text{-}sibling::V]$

$= E[let\ \$e:= U/self::node()\ \text{return}\ \$e/parent::node()/child::V\ [.>>\$e]]$。

3.1.2 关系存储

用 $R[M\rightarrow N]$表示获取节点 M 到 N 的关系值计算，根据不同关系查询，约定关系值 RTYPE 如下：

• 祖后关系(AD)：对 $M//N$，$R[M\rightarrow N]$=RTYPE. DE=1 表示 N 是 M 的子孙；则 $R[N\rightarrow M]$=RTYPE. AN=−1 表示 M 是 N 的祖先。

• 父子关系(PC)：对 M/N，$R[M\rightarrow N]$=RTYPE. CH=2 表示 N 是 M 的孩子；则 $R[N\rightarrow M]$=RTYPE. PA=−2 表示 M 是 N 的父亲。

• 属性关系(AT)：对 $M@N$，$R[M\rightarrow N]$=RTYPE. AR=3 表示 N 是 M 的属性；则 $R[N\rightarrow M]$=RTYPE. AA=−3 同样表示 N 是 M 的属性。

• 兄弟关系(SB)：对 $M\backsim N$，$R[M\rightarrow N]$=RTYPE. FS=4 表示 N 是 M 的右兄弟；则 $R[N\rightarrow M]$=RTYPE. PS=−4 表示 M 是 N 的左兄弟。XPath 规范没有求兄弟关系的简记符号，这里用'∽'代替。

• 前后关系(PF)：$R[M\rightarrow N]$=RTYPE. PP 表示 N 是 M 的前驱节点；则 $R[N\rightarrow M]=RTYPE.\ FF=-$RTYPE. PP 表示 M 是 N 的后继节点。

• 无关系(NN)：RTYPE. NN=0。

若不考虑求自身关系 SS 的情况，XML 任意两个节点之间的关系满足 $R\in((AD\vee PC)\vee(AT)\vee(SB)\vee NN)\wedge(PF)$。

鉴于每两两节点关系在关系矩阵中仅能存储一个值，需要分析关系之间的联系，以便选择有效的存储关系类型，避免冗余和残缺。反向关系值对仅符号不同，可以仅存一个值，比如正值。存在反向关系值对的包括 AN=→DE，PA=→CH，AR=→AA，PS=→FS，AS=→DS，PP=→FF。其中 PP=→FF，由于存在于所有两两节点，而节点 nodeID 反映了文档序，为了简化，PF 关系查询时直接由节点 ID 比较判断，故不必存储 PP，FF 值。在查询 AS 和 DS 关系时，由于 AS=AN∪SS，DS=DE∪SS，因此 AS⇔AN，DS⇔DE，为简化存储，不保留 AS/DS 关系，在查询时，只需要在原语中分别判断为 AN，DE，然后加入 SS 项即可。由于从语义上看，AD 关系包含 PC，在矩阵中约定 AD 不包括 PC，在求 AD 的查询中需考虑 PC。把 AT 当成特殊的 PC，在矩阵中存储其关系值，方便属性关系的查询。综上分析，矩阵中出现的关系可仅保留 DE，CH，AR，FS。对应关系值为 1，2，3，4。

关系节点保留两两节点之间的关系信息，定义为 Matrix-Node(nodeID1，nodeID2，value)，其中 nodeID1，nodeID2 是两个存在关系的节点的 ID，value 存储关系值。关系矩阵指用于存储关系节点的数据结构，逻辑上是 $N\times N$ 的有向矩阵。图 2 是实例图 1 的关系矩阵逻辑结构。图中阴影部分包括关系的下半三角，以及关系值为 0 的情况。由此可见，关系矩阵中，若不计无关系的 0 值，关系值都为正值。N 节点列表按照文档序排列，即节点 ID 从小到大排列，矩阵的构建和查询过程都能使比较操作得到优化。

N: node list (in document order)

M: node list

	A1	B1	C1	D1	C2	A2	B2	A3	D2	C3	C4	D3
A1		2	1	1	1	2	1	1	1	1	1	1
B1			2	1	2	4	0	0	0	0	0	0
C1				2	4	0	0	0	0	0	0	0
D1					0	0	0	0	0	0	0	0
C2						0	0	0	0	0	0	0
A2							2	1	1	1	2	2
B2								2	1	2	4	4
A3									2	4	0	0
D2										0	0	0
C3											0	0
C4												4
D3												

图 2 关系矩阵的逻辑结构

一般来说，XML 关系矩阵属于稀疏阵。图 3 是实际存储的数据结构。考虑存储存在关系的任意两个节点 M 和 N 的关系信息 $M\rightarrow N$，需要包含一个索引数组用于存储 M 节点 ID。每个索引数组项指向一个非定长数组，用于存储与 M 对应的 N 节点 ID 和关系信息。查找关系时，只需要根据 M 节点 ID 信息，直接定位到行，再对该行列表遍历，获取与 M 节点有特定关系的 N 节点。这样每个关系节点实际只需要存储 N 节点 ID 和关系值两个信息即可。

N: node list (in document order)

M: node list

A1	2	1	1	1	2	1	1	1	1	1	1
B1	2	1	2	4							
C1	2	4									
D1											
C2											
A2	2	1	1	1	2	2					
B2	2	1	2	4	4						
A3	2	4									
D2											
C3											
C4	4										
D3											

图 3 关系矩阵的实际存储

由于无法从 Zhang 区间编码信息直接计算兄弟关系，目

前我们的实现中暂时不存储兄弟关系，这样兄弟轴计算需通过本文 3.1.1 节说明的等价语义查找。由于无法直接通过查找关系矩阵获得结果，效率比较低，考虑到兄弟轴计算的应用场合一般不多，这种设计基本能满足需求。未来考虑对兄弟关系也进行预计算，在关系矩阵中存储兄弟关系值，同样可实现高效的兄弟轴操作。

3.1.3 建立关系矩阵

首先进行关系计算，基于节点的区间编码条件判别两两节点之间的关系。算法 1 是根据本文 3.1.1 节描述的求值规则进行关系计算，获取节点 ID 为 nodeID1 的节点到节点 ID 为 nodeID2 的节点的关系值。该算法仅完成 PC，AD 等重要关系的计算，未进行兄弟关系计算；而且由于 PF 关系不存储，故不计算 PF 关系。第 1 行是从 nodeID 获取相应的节点编码指针。第 2 行设定如果不满足节点位置条件，关系值为 0，这里避免了反向关系的重复计算。

算法 1 计算关系值

```
CalcRelation(nodeID1,nodeID2)
1:N1←NodeCode[nodeID1],N2←NodeCode[nodeID2];
2:if(N1.begin<N2.begin∧N2.begin<N1.end)
3:   if(N1.level! =N2.level-1)
4:      relation←RTYPE.DE;
5:      else if(N2.nodeType=ELEMENT)
6:              relation←RTYPE.CH;
7:      else relation←RTYPE.AR;//nodeType=ATTRIBUTE
8:else relation←RTYPE.NN;
9:return relation;
```

其次是存储计算结果，把生成的关系节点加入关系矩阵。算法 2 描述了关系矩阵构建框架，用双重迭代获取两两节点之间的关系信息。为了简化表示，节点 ID 是按文档序连续增长的整数。由于 CalcRelation(nodeID1,nodeID2)=-CalcRelation(nodeID2,nodeID1)，关系相互性实际只要求一次即可。第 2 行 j 循环迭代从 $i+1$ 到 N，总满足 $N_i>N_j$，保证了每两个 ID 只执行一次关系计算，即上三角阵计算，避免重复计算和冗余存储。

算法 2 构建关系矩阵

```
BuildRelationMatrix()
1:for each node id Ni from 0 to N
2:   for each node id Nj from i+1 to N
3:      r←CalcRelation(Ni,Nj);
4:      if(relation! =0)
5:         rNode←new MatrixNode(Ni,Nj,r);
6:         relationMatrix.Add(rNode);
```

矩阵空间复杂度为 $O(N^2)$。但由于实际应用中，一般矩阵填充率不高，而且只需存储半三角内容。相对于 XML 数据文档，矩阵所需的存储空间不大，效果可以接受。我们在第 4 节实验部分验证了这个观点。未来考虑优化设计存储方式，采用压缩技术加索引的方法进一步节省存储空间，而不需改变现有查询机制。

3.2 查询原语设计

XPath 求值表达式由各个查询步骤组成，每种查询步骤需要设计相应的查询原语。查询原语实现查询匹配过程，每个节点不必进行关系计算，而是通过查找关系矩阵，返回匹配结果。整个 XPath 的求值就是查询原语的组合执行。算法 3 是用于求子孙节点的查询原语算法。求取其他关系类型节点的查询原语基本上有相似的实现算法，这里从略。

算法 3 求子孙节点

```
GetDescendant(inputSeq,nameString,isGetFollow)
1:nameTest←false;
2:if(nameString="*") nameTest←true;
3:else nameID←nameIDtable.getNameID(nameString);
4:  for each M∈inputSeq
5:     mList←MNode[M.nodeID];
6:     for each N∈mList
7:        nodeNID←N.getNNodeID();
8:        relation←N.getRelationType();
9:        if((nameTest=true∨(RC[nodeNID].nameID=nameID))
          ∧(relation=RTYPE.DE∨relation= RTYPE.CH))
10:        if(isGetFollow=true)
11:           result.add(N);
          else
12:           result.add(M);
13:           break;
14:return result;
```

第 2，3 行进行节点名称测试判断，如果是通配符，不进行名称测试；反之，需要先从节点名称哈希表中获取名称 ID，以便后续比较。MNode 表示关系矩阵的 XML 节点索引列表。第 5 行根据输入节点 ID，从索引列表中获取节点对应的关系列表，在第 6－13 行内进行条件判断。第 10 行判别是一般求值或谓词求值。若 isGetFollow=true 是一般求值，结果加入 N 节点，即正向节点；反之是谓词求值，结果加入 M 节点，即反向节点。注意，由于谓词求值相当于过滤操作，一旦满足条件就可以终止该 M 节点的条件判断，用第 13 行直接跳出循环。最后第 14 行返回满足条件的节点序列。由于仅进行简单比较，扫描矩阵的代价很低，算法能高效执行。

XML 树的分支查询在实际应用中十分普遍，分支查询是通过 XPath 的谓词来表示的。我们用节点条件过滤方法实现分支求值的谓词原语，如算法 4。

算法 4 过滤节点

```
FilterInput1byInput2(input1,input2)
1:for each M∈input1
2:     mList←MNode[M.NodeID];
3:     for each N∈mList
4:        tmpNID←N.getNNodeID();
5:        relation←N.getRelationType();
6:           if((relation=RTYPE.CH∨relation=RTYPE_DE)∧in-
          put2.contain(tmpNID)=true)
7:           result.add(M);
8:           break;
9:return result;
```

根据 input2 节点序列，过滤 input1 节点系列，需要两层嵌套循环进行条件判断。第 2 行根据输入 input1 节点序列中节点 M 的 ID，从关系矩阵的索引表中获取对应的关系列表。在第 3－8 行检测关系列表中的关系节点是否有满足包含子节点或子孙节点关系，且节点 ID 出现在 input2 序列中的情况。如果是，则该 M 节点就满足过滤条件。当 input2 排序后，input2.contain 可进行二分法快速查找，从而提高了求值性能。

各种查询原语的实现大都需要查询关系矩阵，通过循环检测关系矩阵中的关系节点来完成查询求值。这些循环计算过程适合并行化处理。此外，由于把节点的关系计算变成代价较低的关系查找，节省了计算量，因此查询原语能高效执行。

3.3 查询原语组织

XPath 求值是通过执行查询原语序列实现的。完成 XPath 表达式分析后，根据查询语义产生相应的查询原语序列。以下用两个典型的查询案例，说明用查询原语的组织实现 XPath 求值。

例 1　对于图 1(b)中的查询，可能是以下同构的 XPath 表达式之一，例如//A/B[//D]/C,//A/B[C]//D,//A/B[C and //D],//A[B[C]//D]等等，主要区别在返回值不同。如对//A/B[//D]/C，查询解析后生成以下查询原语调用的执行语句。

```
1:input1←GetDescendant(input0,A. name,true);
2:input2←GetChild(input1,B. name,true);
3:input3←GetDescendant(input2,D. name,false);
4:result←GetChild(input3,C. name,true);
5:return result;
```

第 1 行的输入 input0 是当前上下文的节点序列，节点序列可能仅包含一个文档根节点。第 3 行是谓词求值，即查询子孙关系但返回关系左边序列，也可以用过滤语句实现如下：

```
tmpResult←GetDescendant(input2,D. name,true);
input3←FilterInput1byInput2(input2,tmpResult);
```

先正向求值返回右边序列，再用结果过滤输入的节点序列。

例 2　计算/A[B //C]/ * //D，包含谓词和通配符查询。

```
1:input1←GetChild(input0,A. name,true);
2:input2←GetChild(input1,B. name,true);
3:input3←GetDescendant(input2,C. name,true);
4:input4←FilterInput1byInput2(input1,input3);
5:input5←GetChild(input4," * ",true);
6:result←GetDescendant(input5,D. name,true);
7:return result;
```

第 4 行用过滤操作实现谓词求值，第 5 行的用通配符表示不进行节点名字检测。

查询原语执行的返回结果是节点 ID 序列。以节点 ID 做索引，可从数据模型中定位获取具体的节点信息。需要注意的是，根据 XPath 查询语义要求，结果输出要考虑到文档序与重复问题。一般可以在查询的最后结果处对节点 ID 排序，然后消除重复，以获得符合规范要求的结果形式。

3.4 并行化设计

首先给出一个数据并行化的方法，如算法 5。

算法 5　数据并行

```
ParallelProcess(dataSet[],Task)
1:CountDownLatch latch←new CountDownLatch(sizeof(data));
2:Executor exec←new Executor();
3:for each data dt in dataSet[]
4:      exec. execute(Task(dt));
5:      latch. await();
6:exec. shutdown();
```

输入参数是各个数据集合信息。第 1 行新建一个以数据集内数据块数目为参数的计数器，用于任务同步计数；第 2 行新建一个多线程执行服务，可以指定工作线程数目；第 4 行指派各个线程并行执行任务；第 5 行保证所有子任务的同步；完成所有任务后，第 6 行关闭线程服务。

用 para_for 作为数据并行描述关键字，para_for 的语义等价于算法 5 的 ParallelProcess 调用，即

E(para _for dataExp Exp)=E(ParallelProcess(data-Range,Task))

where dataRange←PackData(dataExp);

Task←PackTask(Exp)。

这里用 E 表示表达式求值函数；用 PackData 辅助函数表示对数据范围表达式 dataExp 进行解析；PackTask 辅助函数表示对执行语句 Exp 进行任务封装。

这样，我们可以用 para_for 描述关系矩阵的构建和查询的并行化。比如算法 2 的第 1 行是外循环，考虑作并行处理，可以改写为 1:para_for each node id Ni from 0 to N。同样，若对查询时过滤节点并行化，算法 4 的第 1 行可以改写为 1:para_for each M∈input1。显然 M^2 方法的并行化处理十分容易。

4 实验与分析

为了验证 M^2 方法在 XPath 查询应用中的效果，我们进行查询性能的对比测试。作为参照的测试对象是 Qizxopen3 和 Saxon8.9，由于二者均为开源项目，且具备良好的查询性能，被广为关注。它们的 XPath 求值基本采用文档树导航方式进行，使用的语言环境均为 JDK，与我们的项目情况一致，有较好可比较性。此外，选取了一个 Twig 查询的典型方法 TwigList[7] 的实现作为参照。考虑到通常情况下关系矩阵可以复用，故仅对查询求值测试，未加入 XML 解析和关系矩阵构建的执行时间。对不同测试案例下，通过获取 M^2 方法和一般导航式方法在不包含 XML 解析的查询阶段的执行时间来进行性能比较。同样，对于 TwigList 来说，执行时间不包括标签流的编码预处理，仅包括节点序列构造时间和结果枚举时间。数据来自通用测试平台 TreeBank[11]，是一个 82Mb 的 XML 文档，采用以下几个典型的测试案例：

Q1. //S//NP/PP/NN

Q2. //S [. //VP] [. //NP]//VP//PP [. //IN]//NP//VBN

Q3. //EMPTY[. //VP/PP//NNP][. //S[. //PP//JJ]//VBN]//PP/NP//_NONE_

Q4. //S/VP//PP[. //NN][. //NP[. //CD]/VBN]/IN

Q5. //NP[. //CD]/ * /V

Q6. //S//VP/parent::SINV/following-sibling:: *

Q1 是一般的路径查询；Q2 是带单层谓词的查询；Q3，Q4 是嵌套谓词查询；Q5，Q6 是带有通配符查询，其中 Q6 还有反向轴和兄弟轴操作。本实验在配备有 AMD Athlon II X4 620 CPU(2.6Ghz)和 2Gb 内存的 PC 上完成，软件环境是 JDK1.6 和 Windows XP 操作系统。图 4 是测试结果。图中纵坐标最大值已限定为 4500ms，实际上测得 Qizxopen3 对 Q2 的平均执行时间为 8203ms。M^2 和优化实现的 Saxon8.9 性能较为

接近，多数查询中，M^2 执行性能甚至略好于 Saxon8.9。由于我们在目前的实现中节点兄弟关系未存入关系矩阵，对兄弟关系的查询通过语义转换实现，故 Q6 时 M^2 执行效率较低。如果改为预存储查询，将会改善 Q6 的执行情况。由于 Twig 查询不支持反向轴和兄弟轴操作，TwigList 没有进行 Q6 的测试。其它几个测试发现 M^2 方法和 Twig 方法的性能很接近，而本测试是 M^2 尚未完成并行优化的情况，并行化后相信能进一步提高其性能。

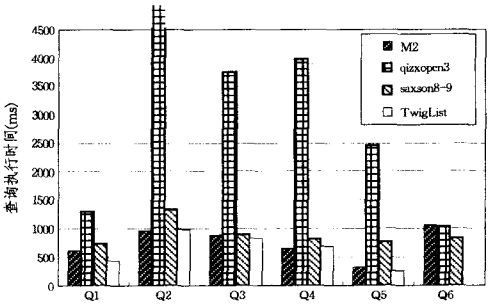

图 4　XPath 求值的性能比较

M^2 查询方法的一个重要特点是依赖于关系矩阵。关系矩阵的大小反映了内存占用大小，直接影响其实际可用性。表 1 显示了不同 XML 数据大小情况下节点数和关系数的对比。我们利用通用测试平台 XMark[12] 的 XML 数据生成工具，在不同参数下生成不同大小的 XML 数据，测试获得相应的 XML 节点数量、关系数量，并计算它们之间的比率，发现 XMark 数据的比率基本呈现稳定。这说明关系数量并没有因为 XML 数据中节点数的增加而快速增加，关系矩阵不会造成存储空间快速膨胀问题，反映了 M^2 方法具有良好的可扩展性。

表 1　节点数与关系数的对比

	27kb	116kb	1.1Mb	11.3Mb	115Mb
节点数	460	2.1k	21.1k	206.1k	2.0M
关系数	2130	9.8k	95.1k	937.7k	9.3M
比率	4.63	4.69	4.52	4.55	4.55

结束语　本文提出的 M^2 方法，基本思路是采用关系矩阵来存储预计算的 XML 节点关系，在查询求值时用代价较低的关系查找代替关系计算。M^2 方法先根据 XML 的区间编码计算节点关系值，以此构建关系矩阵。针对不同 XPath 查询步骤设计对应的查询原语，XPath 求值通过查询原语的组织实现。对于相同的 XML 数据，构建的关系矩阵可以被不同查询复用，构建关系矩阵的代价可以被各个查询摊薄。M^2 的导航式特点使得 XPath 的各种查询语义容易实现，比如反向轴操作和以谓词表达的分支查询等。实验结果表明 M^2 是一种有效的求值方法。由于构建关系矩阵和查询求值的过程中基于循环的处理方式适合并行化优化，考虑到目前多核计算环境的普及，未来的工作重点是实现关系构建和查询过程的并行化。此外，拟通过压缩技术和索引方法优化关系矩阵存储方式，提高内存利用效率。

参 考 文 献

[1] Berglund A, Boag S, Chamberlin D, et al. XML Path Language (XPath) 2.0 [EB/OL]. http://www.w3.org/TR/xpath20, 2007

[2] Fernandez M, Malhotra A, Marsh J, et al. XQuery 1.0 and XPath 2.0 Data Model (XDM) [EB/OL]. http://www.w3.org/TR/xpath-datamodel, 2007

[3] Draper D, Fankhauser P, GmbH I, et al. XQuery 1.0 and XPath 2.0 Formal Semantics [EB/OL]. http://www.w3.org/TR/xquery-semantics, 2007

[4] Fernandez M, Simeon J, Choi B, et al. Implementing XQuery 1.0: The Galax Experience [C]// Proceedings of the VLDB Conference. 2003: 1077-1080

[5] Bruno N, Koudas N, Srivastava D. Holistic twig joins: optimal XML pattern matching [C]// Proceedings of the SIGMOD Conference. 2002: 310-321

[6] Chen S, Li H, Tatemura J, et al. Twig²Stack: bottom-up processing of generalized-tree-pattern queries over XML documents [C]// Proceedings of the 32nd VLDB Conference. 2006: 283-294

[7] Qin L, Xu Yu J, Ding B. TwigList: make twig pattern matching fast [C] //In Proceedings of the DASFAA Conference. 2007: 850-862

[8] Axyana software. Qizxopen [CP/OL]. http://www.axyana.com/qizxopen, 2009

[9] Kay M. SAXON: The XSLT and XQuery Processor [CP/OL]. http://saxon.sourceforge.net, 2009

[10] Re C, Simeon J, Fernandez M. A Complete and Efficient Algebraic Compiler for XQuery [C] //Proceedings of the 22nd ICDE Conference. 2006: 14-26

[11] University of Pennsylvania. The Penn Treebank Project [DB/OL]. http://www.cis.upenn.edu/~treebank, 1999

[12] CWI. XML Benchmark Project [DB/OL]. http://www.xml-benchmark.org, 2009

(上接第 147 页)

[12] Seshia S, et al. A Translation of Statecharts to Esterel [A]// Proceedings of the 1st World Congress on Formal Methods (FM'99) [C]. LNCS 1709. Toulouse, France, 1999: 983-1007

[13] Dwyer M B, Avrunin G S, Corbett J C. Patterns in Property Specifications for Finite-State Verification [A]// Proceedings of ICSE'99 [C]. Los Angeles, 1999: 411-420

[14] McMillan K. Symbolic model checking: An approach to the state explosion problem [M]. Kluwer Academic Publishers, 1993

[15] Bryant R E. Graph-based algorithms for boolean function manipulation [J]. IEEE Transactions on Computers, 1986, 35(8): 677-691

[16] Harel D, Naamad A. The STATEMATE semantics of Statecharts [J]. ACM Transactions on Software Engineering Methodology, 1996, 5(4): 293-333

[17] Harel D, et al. On the formal semantics of Statecharts (extended abstract) [A]// Proceedings of Symposiumon Logic in Computer Science [C]. Ithaca, New York, 1987: 54-64