

F-Index:一种加速 Twig 查询处理的扁平结构索引*

周军锋^{1,2+}, 孟小峰¹, 蒋瑜¹, 谢敏¹

¹(中国人民大学 信息学院,北京 100872)

²(燕山大学 计算机科学与技术系,河北 秦皇岛 066004)

F-Index: A Flattened Structural Index for Speeding up Twig Query Processing

ZHOU Jun-Feng^{1,2+}, MENG Xiao-Feng¹, JIANG Yu¹, XIE Min¹

¹(Information School, Renmin University of China, Beijing 100872, China)

²(Department of Computer Science and Technology, Yanshan University, Qinhuangdao 066004, China)

+ Corresponding author: Phn: +86-10-62515575, E-mail: zhoujf@ysu.edu.cn, <http://www.ruc.edu.cn>

Zhou JF, Meng XF, Jiang Y, Xie M. F-Index: A flattened structural index for speeding up twig query processing. *Journal of Software*, 2007,18(6):1429–1442. <http://www.jos.org.cn/1000-9825/18/1429.htm>

Abstract: How to process twig query quickly and correctly has attracted much attention in research society recently. Filtering query irrelevant elements before query execution is an important step for reducing elements scanned at query processing. As a flattened structural index, F-Index is proposed to filter out all query irrelevant index nodes, thus query irrelevant elements can be filtered out rapidly and mostly, especially when it is processing deeply nested XML documents with a complex structure. After filtering, a new efficient query algorithm based on the remaining elements is proposed to accelerate query processing. Experimental results on various datasets indicate that twig query's performance can be improved significantly by using F-Index.

Key words: XML; query optimization; twig query; filtering; structural index

摘要: 如何快速、有效地处理 twig 形式的查询是 XML 查询处理的关键问题,通过过滤与查询无关的元素可以减少查询中需要处理的元素数目,从而提高查询的执行效率.提出一种扁平结构索引 F-Index,能够快速过滤所有与查询无关的索引结点,进而过滤掉查询无关的元素,在处理深度嵌套的复杂结构 XML 文档时具有很大的优势.提出一种新的查询算法,能够有效处理过滤后剩余元素的匹配问题.基于不同数据集的实验表明,使用 F-Index 进行过滤可以极大地提高查询处理的性能.

关键词: XML;查询优化;twig 查询;过滤;结构索引

中图法分类号: TP311 文献标识码: A

XML 作为事实上的数据描述和交换标准已经得到了广泛的应用,有关 XML 的各种处理技术成为大量研

* Supported by the National Natural Science Foundation of China under Grant Nos.60573091, 60273018 (国家自然科学基金); the National Basic Research Program of China under Grant No.2003CB317000 (国家重点基础研究发展计划(973)); the Key Project of Ministry of Education of China under Grant No.03044 (国家教育部科学技术重点项目); the Program for New Century Excellent Talents in University (新世纪优秀人才支持计划)

Received 2006-04-20; Accepted 2006-06-30

究者关注的热点问题.XML 文档包含具有层次嵌套关系的元素信息,可以很自然地用一棵文档树来表示,Twig (小树)查询就是从文档树中找到与 Twig 匹配的元素序列.一个 Twig 查询可以用一个带有结点标注信息的树来表示,树中结点之间的边表示父子(PC)或者祖先后代(AD)关系.例如,XPath 查询//book [//appendix]//figure 用于检索所有 book 后代中的 figure 结点,并且这些 book 结点的后代中必须有 appendix 结点.

为了快速求解用户提交的查询,可先在 XML 文档的结构索引上过滤,从而减少需要处理的元素数量以提高查询算法的整体性能.即整个查询时间由两部分组成,分别是过滤时间和从剩余元素中求解的时间.

XML 文档结构的不确定性导致了不同 XML 文档结构的巨大差异性.如表 1 所示,对于 XMark 数据集而言,1-index 索引结点只有 514 个;而对于 TreeBank 数据集而言,1-index 索引结点的数量却达到了 338 748 个.虽然通过索引进行过滤可以减少处理的元素数量,但对索引本身的处理却没有得到足够的关注.在不同数据集对应的结构索引上进行过滤,其过滤性能将因文档结构的不同而出现较大差异.

Table 1 Comparison of different datasets' features

表 1 不同数据集特征比较

Dataset	Size (M)	Tags	Doc. nodes	1-index nodes	Doc. nodes/1-index nodes
XMark	115	74	1 666 315	514	3 242
TreeBank	84	250	2 437 666	338 748	7

由于过滤后的索引结点之间已经满足查询要求的 AD 或 PC 关系,且同一个索引结点对应的元素之间不存在相互嵌套的情况,因此,充分利用这一特点可以避免整体匹配算法中的很多冗余操作.

本文的主要贡献可以表述如下:

(1) 提出了一种新的扁平结构索引 F-Index,可以快速过滤掉所有与查询无关的索引结点,XML 文档的结构索引越复杂,F-Index 的优势越明显.

(2) 提出了一种新的查询处理方法,可以快速得到满足条件的元素序列.

(3) 通过实验,对本文提出的基于过滤的查询方法的各项指标及整体性能进行了验证.

本文第 1 节介绍相关工作.第 2 节介绍基于过滤的查询处理策略.第 3 节介绍 F-Index 的基本结构和 F-Index 的构建过程.第 4 节描述通过 F-Index 进行过滤的具体算法.第 5 节介绍本文提出的查询匹配算法.第 6 节介绍 *轴的处理.第 7 节展示实验结果.第 8 节为本文的结论.

1 相关工作

导航式查询^[1-5]需要对某些元素进行多边扫描,因此速度最慢;结构连接^[6-13]减少了扫描的元素数量,但会产生大量的中间结果;整体匹配算法^[14,15]在求解时将查询作为一个整体来处理,因而速度得到了较大的提升,TwigStack^[14]是其中的典型代表,但其仅在处理 AD 关系的查询时有较好的性能,算法本身的处理逻辑限定了当查询中出现了 PC 关系的路径时,需剔除不满足条件的 AD 序列,且对 Twig 形式的查询而言,需要执行路径的合并操作.文献[15]引入了 XR-Tree 索引,可以跳过不必要的数据库访问和比较操作,由于 XML 文档元素分布的不确定性,效率无法保证^[16].

已有的工作表明,可以通过在各种结构索引上执行过滤来提高查询执行效率.DataGuide^[17],1-index^[18]和 F&B 索引^[19]是其中的典型代表.利用索引进行过滤的工作主要体现在文献[16,20-22]中.文献[16,20]指出,通过索引过滤可以减少查询处理的元素数量,但没有对结构复杂数据集的过滤效果进行验证.文献[21,22]在 TreeBank 数据集上的实验效果显示,使用结构索引进行过滤尽管可以过滤掉大量查询无关的元素,但所用时间远多于没有进行过滤的情况.处理索引时,除了文献[22],只有基于导航的方式用于过滤.

为了表示方便,本文的后续部分以“/”或者“PC”表示结点之间的父子关系,以“//”或者“AD”表示结点之间的祖先后代关系,大写字母表示元素对应的 Tag 名,如 A,B 等,大写的斜体字母表示索引结点的名称,如 *A*,*B* 等,小写字母表示文档元素,如 *a*,*b* 等.

2 基于过滤的查询处理

基于过滤的查询处理可分为两个步骤来完成:第 1 步过滤掉查询无关的索引结点,第 2 步在过滤后剩余元素中寻找满足条件的解.

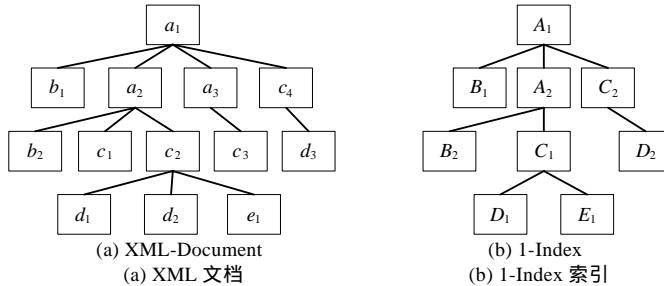


Fig.1 XML document and the corresponding 1-index structure

图 1 XML 文档及 1-index 索引结构

例 1:对于查询//C[//D]E 而言,为了得到满足条件的结果,可以使用 TwigStack 算法求解.这时需要处理的元素是 $c_1 \dots c_4, d_1 \dots d_3, e_1$.若先进行过滤,如图 1(b)所示,可得到满足条件的 1-index 索引结点为 C_1, D_1 和 E_1 ,则在第 2 步求解的时候处理的元素将变成 $c_1, c_2, c_3, d_1, d_2, e_1$,即 c_4, d_3 在第 1 步已经被过滤.注意到,即使经过过滤仍有某些查询无关的元素存在,如: c_1, c_3 ,因此,需要在第 2 步使用某种查询算法以得到最终满足条件的解.

基于以上考虑,本文提出了一种新的扁平结构索引,其基本思想就是:提前记录不同结点之间的可达性信息,并将这些可达性信息进行有效组织,使得在过滤时只需访问极少量结点即可完成过滤,从而避免已有方法在遇到复杂结构索引时性能下降的问题.

3 F-Index 的组织结构

3.1 F-Index的组织结构

我们用 $\langle start, end, level \rangle$ 编码方式对 1-index 中的结点进行编码.原来的 1-index 结点按照各自 Tag 组织成不同的链表,链表中的索引结点按照 $start$ 值升序排列,如图 2 所示,我们称其为 F-Index 的辅助查询结构.

性质 1. 如果索引结点 A 对应 n 个 Tag 为 B 的后代索引结点,其中第 1 个后代结点为 B_i ,那么这 n 个后代结点在如图 2 所示的存储结构中就是从 B_i 开始的 n 个结点.

假定 A 表示 1-index 索引结点,则 A 的后代结点可以表示为 $S = \{n_1, n_2, \dots, n_k\}$,这里, $n_i (1 \leq i \leq k)$ 表示 A 的一个后代结点.为了加速存取,根据性质 1,将 S 按照不同 Tag 划分为互不相交的子集,即 $S = S_1 \cup S_2 \cup \dots \cup S_m (m \leq k)$ 并且 $S_i \cap S_j = \emptyset ((1 \leq i \leq m) \wedge (1 \leq j \leq m) \wedge (i \neq j))$,这里, m 是 Tag 的数量,对任意子集 $S_i (1 \leq i \leq m)$,用 $|S_i|$ 表示其元素个数,同时记录该子集的第 1 个结点 $n_{S_i,1}$.根据性质 1,如果要得到 A 的 Tag 为 B 的后代结点,只需在图 2 中找到 $n_{B,1}$ 的位置,然后顺序存取 $|S_B|$ 个结点即可.

我们用一个链表元素记录这些信息,如图 3 所示.图 3(a)是链表元素的结构图,图 3(b)为图 3(a)的一个实例.其中,AncestorNode 值为 A_1 ,表示 A_1 是祖先结点且含有 Tag 为 C 的后代结点;ChildNode 值为 C_2 ,表示 A_1 对应的 Tag 为 C 的孩子结点是 C_2 ,当没有这样的索引结点存在时,用 -1 表示;MinDescNode 表示 A_1 对应的 Tag 为 C 的后代中 $start$ 值最小的结点,这里为 C_1 ,nDescCount 表示 A_1 对应的 Tag 为 C 的后代索引结点的个数.从图 1(b)可以看出,有 2 个 Tag 为 C 的结点符合要求,即 C_1, C_2 都是 A_1 的后代.提取 A_1 所有后代结点的方法就是根据 MinDescNode 定位到如图 2 所示的辅助查询结构中的 C_1 ,顺序向后取 2 个结点即可.

图 3(b)记录了 A_1 的所有 Tag 为 C 的后代结点的情况.从图 1(b)可以看出,有多个 Tag 为 A 的索引结点都含有 Tag 为 C 的孩子,所有这样的结点用一个链表组织起来,链表中的元素按照 AncestorNode 升序排列,如图 3(e)所示,该链表的内容记录了当前结构索引中所有 Tag 为 A 的索引结点到 Tag 为 C 的索引结点的可达性信息.

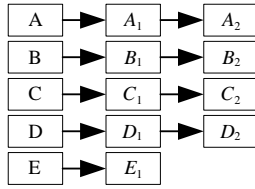


Fig.2 Query aided structure

图 2 辅助查询结构

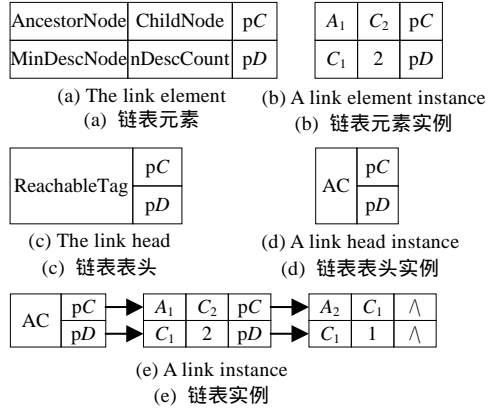


Fig.3 The link structure

图 3 链表结构

图 3(c)是该链表的表头元素示意图,图 3(d)是链表表头实例,ReachableTag 表示 1-index 结构中具有可达性关系的两个 Tag 名组成的哈希主键,我们以图 3(d)为例进行说明,这里,AC 表示某些 Tag 为 A 的 1-index 结点含有 Tag 为 C 的后代结点.pC 是一个指针,指向下一个 Tag 为 A 的索引结点 A_i ,且 A_i 必须含有 Tag 为 C 的孩子结点,pD 也是一个指针,指向下一个 Tag 为 A 的索引结点 A_m ,且 A_m 必须含有 Tag 为 C 的后代结点.图 3(a)的 pC 和 pD 与图 3(c)中的 pC 和 pD 的意义相同.

图 3(e)完整地刻画了图 1(b)所示的结构索引中 Tag 为 A 的索引结点包含后代的 Tag 为 C 的索引结点的情况,我们用不同的链表记录不同 Tag 集合的结点之间的可达性信息,所有这些链表一起构成了如图 4 所示的 F-Index.

性质 2. F-Index 记录了其所处理的结构索引中所有非叶结点与其他结点的可达性信息.

性质 3. 对于任意简单路径 //A/B,从 pC 指针连接起来的链表中可以直接获取所有 Tag 为 A 的 1-index 结点,并且这些索引结点含有 Tag 为 B 的孩子结点.

性质 4. 对于任意简单路径 //A/B,从 pD 指针连接起来的链表中可以直接获取所有 Tag 为 A 的 1-index 结点,并且,这些索引结点含有 Tag 为 B 的后代结点.

性质 5. 对于任意 Twig //A//[B]/C,通过 AB 和 AC 可以找到两个对应的 pD 指针,两个指针同时推进,比较两个指针所指链表元素中的 AncestorNode,即比较 Tag 为 A 的索引结点,若相等,则是满足条件的索引结点,且这些结点的后代中同时拥有 Tag 为 B 和 C 的结点,从而过滤掉仅包含 Tag 为 B 或者 C 的 A 索引结点.

这一性质同样适用于 //A/[B]/C 或者 //A//[B]/C 形式的 Twig.对于结构索引中 Tag 为 A 的叶子结点,F-Index 没有记录,因此无须处理.性质 5 是性质 3 和性质 4 的扩展,过滤时只需处理 A 对应的非叶结点即可,并且无须将查询拆分为 //A/B 和 //A/C 分别处理后再执行合并操作.

性质 6. 对于任意含有多个非叶结点的 Twig,例如 //A//[B]/C//[D]/E,可直接提取 Tag 为 A 的索引结点,这些结点含有 Tag 为 C 的孩子结点,并且后代中同时拥有 Tag 为 B,D 和 E 的结点,这样就可以过滤掉只能到 B 和 C,但不能到 D 和 E 的 A 结点.

性质 6 是性质 5 的扩展,它可以使我们在处理复杂查询时,减少合并中间结果的次数.

例 2:我们以图 5 中 3 个具有代表性的例子来说明如何利用 F-Index 快速处理不同种类的查询过滤问题.

对于 Q_1 ,先根据 C 和 D 找到相应的链表,根据 C 和 D 之间是 AD 关系从表头元素中取出 pD 指针,顺着 pD 指针的移动,依次取出相应的结点序列即可,这里满足条件的结果序列是 C_1D_1, C_2D_2 .

对于 Q_2 ,先根据 CD 找到相应的 pD 指针,然后根据 CE 找到相应的 pC 指针,比较二者所指的链表元素中的 AncestorNode 是否相等,若相等则提取相应序列;若不等,则最小 AncestorNode 值对应的指针指向下一个链表元素再作比较,直到某个链表的表尾为止.根据图 4 可以轻易求得满足 Q_2 的索引结点序列为 $C_1D_1E_1$.

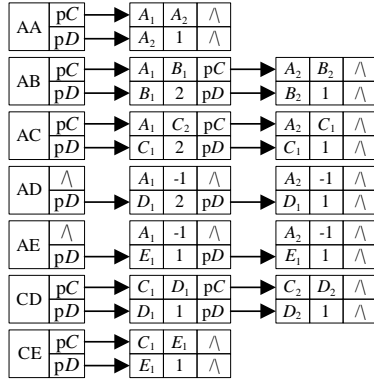


Fig.4 Structure of F-Index

图 4 F-Index 的组织结构

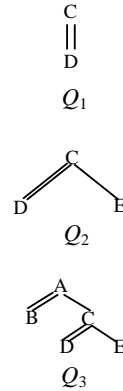


Fig.5 Query examples

图 5 查询用例

Q_3 是含有多个分支结点的查询,其具体的操作过程将在第 4 节进行详细说明.

F-Index 的特点可归纳如下:

- (1) 通用性.F-Index 可建立在不同的结构索引之上,并非针对某一种特定的结构索引.
- (2) 高效性.体现在以下 3 个方面:

针对某个查询,F-Index 需要处理的索引结点数量在最坏情况下是查询中非叶结点对应的索引结点个数.

在处理带分支的查询时,进行合并操作的次数很少.若查询中只有一个非叶结点,则无须合并操作.

针对构建 F-Index 的索引而言,F-Index 可以过滤掉所有的查询无关索引结点.

3.2 F-Index的构造

由于需要记录给定索引中所有非叶结点到任意 Tag 对应的索引结点的可达性信息,因此要对这些信息进行统计,整个构建的过程就是对给定结构索引的一次遍历过程.限于篇幅,详细的构建步骤在这里不予说明.

针对给定结构索引中任意一个非叶结点,F-Index 用一个链表元素记录其到任意一个 Tag 对应的结点的可达性信息,最坏情况下,记录该结点需要 n_{Tag} 个链表元素,因此,F-Index 的空间复杂度最坏情况下为 $O(n_{index} \cdot n_{Tag})$,这里, n_{index} 表示结构索引中的结点数, n_{Tag} 表示 Tag 数.对于给定的索引而言,其所用空间远没有这么多,原因是对于索引中的叶结点而言,不用记录其可达性信息,且每个索引结点只能到达部分 Tag 对应的索引结点.

4 基于 F-Index 的过滤算法

定义 1. 对每个需要处理的非叶查询结点 A,将其可以到达的所有后代结点的 Tag 及其与 A 的关系进行记录,这些后代结点及其与 A 的关系就是 A 结点的可达性信息.

例 3:对于如图 5(c)所示的查询 Q_3 而言,非叶结点有 A 和 C,我们以 A 为例进行说明,A 的可达性信息中包括 A 可以到达的所有后代结点 B,C,D,E,其中,A 和 C 是 PC 关系,A 和 B,D,E 都是 AD 关系,A 的孩子结点有 B 和 C.

4.1 算法描述

算法 1. *FilterData(indexInfo)*.

- 1: *curBranchNode*=*filterInfo*.GetFirst();
- 2: **while** (*curBranchNode*)
- 3: *ToTagList*=*curBranchNode*.GetToTagList();

```

4:   while (not bFilterEnd)
5:     if (FindMatchedSeq())
6:       RecordMatchedSeq(curBranchNode);
7:       AdvanceAllPointer(ToTagList);
8:     else
9:       break;
10:    curBranchNode=filterInfo.GetNext();
11: RecordFilterResult(FilteredResult);
12: return FilteredResult;

```

Procedure FindMatchedSeq()

```

1: while (all pointers in ToTagList are not equal)
2:   pCur=FindMinPointer(ToTagList)
3:   AdvancePointer(pCur);
4:   if (pCur=null)
5:     return false;
6: return true;

```

Procedure RecordMatchSeq(*curNode*)

```

1: childNode=curNode.FirstChild();
2: while (childNode)
3:    $Q_i$ =curNode.GetChildNodeValue();
4:    $P_i$ =childNode.GetNodeValue();
5:   while( $Q_i=P_i$ )
6:     NodeList=Node( $P_i$ ).GetRecordedList();
7:     MergeNodeList(curNode,NodeList);
8:      $P_i$ =childNode.GetNextNodeValue();
9:     childNode=curNode.NextChild().

```

查询结点的可达性信息都存放在变量 *filterInfo* 中.第 1 行, *filterInfo*.GetFirst() 得到第 1 个被处理的查询结点, 对于 Q_3 就是 C.第 2~9 行描述了如何处理当前分支结点 *curBranchNode*,第 3 行 *curBranchNode*.GetToTagList() 得到当前被处理结点的可达性信息,返回值保存在 *ToTagList* 中. *FilterEnd*() 用于判断是否存在空指针,若存在则处理完毕.否则, *FindMatchedSeq*() 用于寻找下一个匹配的序列, *RecordMatchedSeq*(*curBranchNode*) 用于记录匹配的结果,同时所有指针前移.第 10 行的 *filterInfo*.GetNext() 用于得到下一个被处理的查询结点,若返回值为空,则整个处理过程结束,第 11 行用于记录最终返回的结果,第 12 行返回结果.

过程 *FindMatchedSeq*() 用于寻找匹配的索引结点,若 *ToTagList* 中所有指针所指链表元素的 *AncestorNode* 不等,则第 2 行找到一个不是指向最大值的指针并将其前移.若前移之后该指针为空,则返回 false.若 *ToTagList* 中所有指针所指的值相等,则返回 true,表示找到一个匹配的结果.

过程 *RecordMatchSeq*(*curNode*) 用于记录匹配的索引结点,第 1 行 *curNode*.FirstChild() 找到 *curNode* 的第 1 个孩子结点,第 3 行 *curNode*.GetChildNodeValue() 用来得到前面处理完毕的查询结点 Q_i ,这个结点用于与当前结点的孩子结点 P_i 执行合并操作,第 4 行使用 *childNode*.GetNodeValue() 返回 P_i ,若 $Q_i=P_i$,则第 6 行 *Node*(P_i).GetRecordedList() 返回的结果在第 7 行通过 *MergeNodeList*(*curNode*,*NodeList*) 执行合并操作.第 8 行的 *curNode*.NextChild() 用于得到下一个孩子结点.

定理 1. 使用算法 1 过滤后,任意一个索引结点序列中的结点之间已经满足查询要求的 PC 或者 AD 关系.
证明:略.

例 4:我们以图 5(c)中的 Q_3 为例说明使用 F-Index 进行查询处理的基本情况.

首先对 Q_3 进行分解,以先序遍历的顺序记录所有非叶结点的可达性信息并将其入栈,这里的入栈顺序是先 A 后 C.过滤的时候,按照出栈的先后顺序对栈中元素进行处理,即先处理 C,再处理 A.

(1) 处理 C:根据 CD 和 CE 找到两个链表,根据相应的 AD 或者 PC 关系定位到相应的 pC 或者 pD 指针,在两个指针同时向后扫描的过程中找到满足条件的索引结点序列.根据图 4,可以得出满足条件的结果是 $C_1D_1E_1$.

(2) 处理 A:使用 AB,AD,AE 得到 3 个 pD 指针,然后用 AC 得到一个 pC 指针,4 个指针同时前进,当所指的链表元素中的 AncestorNode 相同时,得到一个中间结果序列;第 1 次得到的索引结点序列是 $A_1B_1C_2$, A_1 对应的 Tag 为 C 的孩子是 C_2 ,而在处理 C 时得到的序列中没有 C_2 ,因此不用进行合并操作,4 个指针同时向后走;第 2 次得到的结点序列是 $A_2B_2C_1$, A_2 对应的 Tag 为 C 的孩子是 C_1 ,因此,合并一次即可得到最终满足查询的索引结点序列 $A_2B_2C_1D_1E_1$.

4.2 性能分析

过滤的过程其实就是对查询中非叶结点对应的索引结点进行一次扫描的过程,因此,F-Index 过滤算法的时间复杂度在最坏情况下为 $O(n)$,这里, n 表示查询树中非叶结点对应的索引结点的个数.

5 查询匹配算法

假定查询树中每个结点对应一个元素数据流,数据流中的每个元素按照 $(start, end, level)$ 方式进行编码,且元素按照 $start$ 值的升序排列.每个查询结点 q 附加唯一一个光标变量 C_q ,如图 6 所示, C_q 指向 q 对应数据流中的某个元素.在不引起歧义的情况下,我们用“ C_q ”或“元素 C_q ”表示 C_q 所指的元素. C_q 可以指向当前所指元素的下一个元素,该操作用 $Advance(C_q)$ 来表示;相应地,可以通过 $C_q.start, C_q.end$ 以及 $C_q.level$ 来取得相应的元素属性,开始处理时,所有查询结点对应的光标指向相应数据流的第 1 个元素.

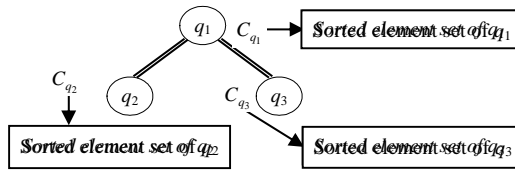


Fig.6 Cursors during execution

图 6 执行过程中的光标

定义 2. 解决方案(solution). 对于以 q 为根的查询子树而言,称 q 具有解决方案,当且仅当 q 满足以下两个条件中的任意一个:

(1) 若 q 是叶结点,则 C_q 不空.

(2) 若 q 不是叶结点,则 q 的孩子结点都有解决方案且 $C_q.start < C_{q'}.start \wedge C_q.end > C_{q'}.end$. 这里, q' 是 q 的任意一个孩子结点.

定理 2. 假定 $S = \{q_1, q_2, \dots, q_n\}$ 是查询 Q 的结点集, Q 的根结点是 q_1 ,如果 q_1 有解决方案,那么,元素序列 $C_{q_1}C_{q_2}\dots C_{q_n}$ 是 Q 的一个匹配序列.

证明:如果所有查询结点对应的当前元素已经构成一个解决方案,定义 2 可以保证这些元素之间的 AD 关系.既然每个元素都对应了一个索引结点,根据定理 1 可知,这些索引结点之间已经满足查询要求的 PC 关系,因此,这些元素之间的 PC 或者 AD 关系满足查询的要求.

5.1 算法描述

算法 2 的第 1 行使用 FindSolution()判断根结点是否具有解决方案,若返回值为 true,则在第 2 行使用 RecordMatchedResult()记录匹配的序列.FindSolution()用于查找当前查询结点 q 的解决方案,从根结点 q 开始,

如果 C_{q_i} (q_i 是 q 的一个孩子结点) 是 C_q 的后代, 那么递归调用 FindSolution 用以判断是否 q_i 具有解决方案. 如果 q 的所有孩子都具有解决方案, 且 C_q 和所有孩子的当前元素都满足 AD 关系, 则返回 true, 否则返回 false. RecordMatchedResult() 用于记录根结点当前元素的 $(start, end)$ 范围内所有的匹配元素序列.

算法 2. CHTwigQuery($root$).

```

1: while (FindSolution( $root$ ))
2:   RecordMatchedResult();
Procedure FindSolution( $q$ )
1: for  $q_i$  in children( $q$ ) do
2:   while ( $C_{q_i}$  is not a descendant of  $C_q$ )
3:     while ( $C_{q_i}.start < C_q.start$ )
4:       Advance( $C_{q_i}$ );
5:       if (end( $q_i$ ))
6:         return false;
7:     while ( $C_{q_i}.start > C_q.end$ )
8:       Advance( $C_q$ );
9:       if (end( $q$ ))
10:        return false;
11:    if (!FindSolution( $q_i$ ))
12:      return false;
13: return true.

```

算法 3 描述了基于 F-Index 进行查询处理的整个过程, 首先在索引结构上执行过滤操作, 如第 1 行所示, 然后对于过滤得到的每一个索引结点序列, 将序列中的索引结点对应的数据流设定到查询中的结点上, 如第 3 行, 最后执行查询操作并输出查询结果, 如第 4 行所示.

算法 3. FilterQuery($root$).

```

1: FilteredResult = FilterData(indexInfo);
2: for each Seq in FilteredResult;
3:   AppendSeqToQuery(Seq);
4:   CHTwigQuery( $root$ );

```

Procedure RecordMatchedResult (q)

```

1: for each  $q_i$  in children( $q$ ) do
2:   while (!end( $q_i$ ) &  $C_{q_i}.end < C_q.end$ )
3:     if (RecordMatchedResult( $q_i$ ))
4:       AddResultToParent( $q_i, q$ );
5:     Advance( $q_i$ ).

```

例 5: 求满足查询 //A//C/D 要求的元素序列.

通过执行过滤算法, 得到满足条件的 1-index 结点序列为 $A_1C_1D_1, A_1C_2D_2$ 和 $A_2C_1D_1$, 其对应的元素序列如图 7 所示. 首先处理 $A_1C_1D_1$, 通过执行算法 2 的第 1 行, 3 个光标 C_A, C_C, C_D 分别指向 a_{1-}, c_{2-} 和 d_1 , 然后记录满足条件的结果序列, 即执行算法的第 2 行, 得到满足条件的两个结果序列 $a_1c_2d_1$ 和 $a_1c_2d_2$. 这里记录的是在 a_1 范围内的所有满足条件的元素序列. 然后处理 $A_1C_2D_2$, 3 个光标 C_A, C_C, C_D 同时前进, 分别指向 a_{1-}, c_4 和 d_3 , 然后继续寻找满足条件的序列, 这里是 $a_1c_4d_3$. 最后处理 $A_2C_1D_1$, 3 个光标 C_A, C_C, C_D 分别指向 a_{2-}, c_2 和 d_1 , 通过执行算法 2 的第 2 行, 得到满足条件的元素序列为 $a_2c_2d_1$ 和 $a_2c_2d_2$.

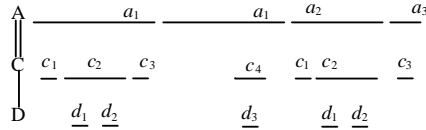


Fig.7 The elements after filtering

图 7 过滤后的元素

5.2 性能分析

使用 TwigStack 算法进行第 2 阶段的查询处理,存在两个问题:(1) 需对过滤后的元素重排序,这一步需耗费的时间为 $O(mn^2)$,这里, m 表示每个索引结点对应的平均元素数目, n 是索引结点数;(2) 需要去除不满足查询要求的路径序列,再对剩余简单路径执行合并操作.使用文献[21]的方法无需排序操作,对每个查询结点的所有数据流,每次选择最小的元素,这是一次 $O(k_{index}/n)$ 操作,这里, k_{index} 为过滤后的索引结点数, n 为查询结点数.

使用 CHTwigQuery 算法,每提取一个符合查询要求的完整序列需要的时间是 $O(n/m)$,这里, n 表示算法 CHTwigQuery 处理根结点时需要扫描的元素数目, m 表示根结点($start, end$)内所有满足条件的结果序列数.

6 *轴的处理

为了充分利用 F-Index 的特性,我们使用扩展区间编码标注索引结点,形式为($start, end, level, parentStart$),其中, $parentStart$ 等于其父结点的 $start$ 值,可以通过比较不同结点的 $parentStart$ 值来确定其是否是兄弟结点.

性质 7(查询转换规则). 如果查询 Q 包含*轴,则 Q 可根据以下规则转换为不带*的查询 Q' :

(1) $Q=A//B$ 可转换为 $Q'=A//B$,约束条件为 $C_A.level+2=C_B.level$.

(2) $Q=A//*/B$ 或 $Q=A/*//B$ 或 $Q=A/*//*/B$ 可以转换为 $Q'=A//B$,约束条件为 $C_A.level+2 \leq C_B.level$.

(3) $Q=A/*[B]/C$ 可转换为 $Q'=A//[B]/C$,约束条件为

$$C_A.level+2=C_B.level \wedge C_A.level+2=C_C.level \wedge C_B.parentStart=C_C.parentStart.$$

(4) $Q=A/*[B]/C$ 可以转换为 $Q'=A//[B]/C$,约束条件为

$$C_A.level+2 \leq C_B.level \wedge C_A.level+2 \leq C_C.level \wedge C_B.parentStart=C_C.parentStart.$$

(5) Q 是单路径查询且其中两个查询结点 A, B 间包含 n 个连续的*轴,约束条件为 $C_A.level+n+1 \leq C_B.level$.

(6) Q 是 twig 并且*轴含有 n 个孩子结点,则转换类似于(3)和(4).

以上 6 种情况可作为对包含*轴的查询进行转换的依据.除了这 6 种情况,仍有一些无法转换的情况,这时的处理方式就是枚举所有可能的情况.文献[23]中仅涉及到包含*轴的单路径的处理,本文的方法对其进行了扩展,可以高效地处理部分包含*轴的 twig 查询.

例 6:查询 $A[*[B]/C]/D$ 可以转换为等价形式 $A//[B]/[C]/D$,其约束条件为

$$(C_A.level+2=C_B.level) \wedge C_A.level+2=C_C.level \wedge (C_B.parentStart=C_C.parentStart).$$

7 实验

7.1 实验环境和数据集

本文的实验在一台基本配置为 P4 2.8GHz, 1G RAM, 80G 硬盘的 PC 上运行,底层操作系统是 Windows XP. 我们在 VC++6.0 环境下实现了本文提出的查询过滤算法、TwigStack^[14]、iTwigStack^[21]及 SegSJ^[22]算法.所用数据集包括 XMark^[24]和 TreeBank^[24].本文实验中 F-Index 基于 1-index 索引进行构建.

7.2 实验所用查询和评价标准

针对 XMark 的查询包含 QX1~QX8,针对 Treebank 的查询为 QT1~QT4,见表 2.

Table 2 Queries used in the experiment
表 2 实验所用查询

Query No.	XPath expression	Query No.	XPath expression
QX1	/site/regions/africa/item/description/parlist/listitem/text/keyword	QX7	//listitem[//bold]/text/emph
QX2	/site/closed_auctions/closed_auction[annotation/description[parlist/listitem/text[keyword[bold]]]]/price	QX8	//listitem[//bold]/text[//emph]/keyword
QX3	/site/closed_auctions//emph	QT1	//S//ADJP[//MD]
QX4	/site/people/person[//age]/education	QT2	//S[//JJ]/NP
QX5	//site/people/person/name	QT3	//S//VP//PP[//NP//VBN]/IN
QX6	//text[//bold]/emph/keyword	QT4	//S//VP//PP[//NP//VBN]/IN

为了说明 F-Index 及本文提出的查询算法的有效性,我们从 5 个方面对基于 F-Index 的查询算法的各项性能进行了全面比较: F-Index 的存储空间; 基于 F-Index 的过滤性能; 基于 F-Index 过滤后的查询时间; 基于 F-Index 的查询算法的整体性能; 基于 F-Index 的查询算法的扩展性.

7.3 实验结果及其分析

7.3.1 F-Index 的存储空间

表 3 是基于本文所用数据集建立的 F-Index 与 1-index 结点数量比较.可以看出,XMark 数据集上,F-Index 的大小基本上稳定在 1-index 的 2.2 倍;Treebank 数据集上,F-Index 是 1-index 的 2.5 倍.与各自数据集的元素数量相比,都是可接受的,均远远小于 $O(n_{index} \cdot n_{tag})$ 的理论上限.

Table 3 Size of F-Index and 1-index
表 3 F-Index 和 1-index 的大小比较

Dataset (size)	XMark (1M)	XMark (12M)	XMark (35M)	XMark (57)	XMark (93M)	XMark (116M)	Treebank (84M)
Elements	17 132	167 865	501 498	832 911	1 337 383	1 666 315	2 437 666
1-index nodes	421	502	514	514	514	514	338 748
F-Index nodes	1 039	1 118	1 130	1 130	1 130	1 130	846 228
F-Index nodes/1-index nodes	2.5	2.2	2.2	2.2	2.2	2.2	2.5

7.3.2 基于 F-Index 的过滤性能

(1) 过滤前后元素数量对比

如图 8(a)和图 8(b)所示,使用 F-Index 过滤后,元素数目明显减少,最明显的是 QX1,从 322 630 缩减到 2 632.同时可以看出,在结构复杂的文档上过滤,效果比较明显,如在 Treebank 上过滤后剩余元素数量为原来的 1/6~1/3. XMark 数据集的结构相对规则,不同查询的过滤效果不同,如 QX4 就没有过滤掉任何元素.对比不同算法的过滤效果可以看出,SegSJ 过滤效果最好,原因是其基于 F&B 索引进行过滤,所有与查询无关的元素均被过滤掉.

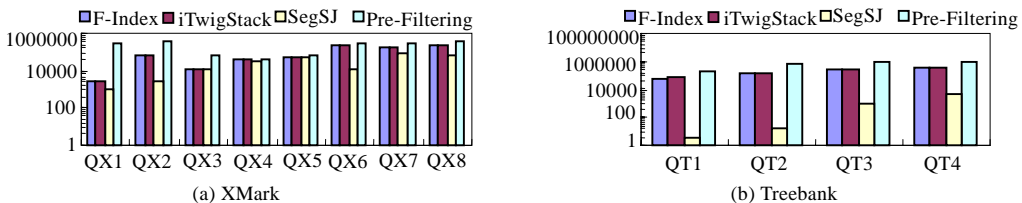


Fig.8 Comparison of elements number
图 8 元素数量对比

(2) 不同算法过滤时处理的索引结点数量对比

如图 9(a)和图 9(b)所示,F-Index 处理的索引结点远少于其他两种算法,但 QX1 及 QX2 上多于 iTwigStack,原因是 QX1 是 PC-path,第 1 个查询结点与索引的根结点匹配,用导航式搜索只需处理少量结点,QX2 是 PC-twig

且第 1 个结点与根结点匹配,其处理的结点数量很少.SegSJ 使用的是 F&B 索引,处理的索引结点数量很多,但因其先在 1-index 索引上过滤,然后在 FB 索引上过滤,处理 Treebank 时的结点少于 iTwigStack.iTwigStack 在 Treebank 上的性能急剧下降,原因是 Treebank 的 1-index 结构庞大,导航式处理需扫描的结点数急剧膨胀.

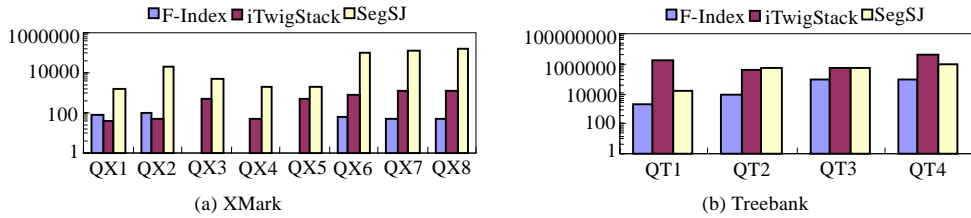


Fig.9 Comparison of processed index nodes number

图 9 处理的索引结点数量对比

(3) 不同过滤算法运行时间对比

从图 10(a)和图 10(b)可以看出,对于 XMark 而言,由于结构相对简单,需要处理的索引结点数量不多.使用 F-Index,过滤时间可以忽略;对于 Treebank 数据集,其过滤时间也很少.在 Treebank 上过滤,所需时间最多的是 QT4,需要 465ms,相对数量庞大的索引结点而言,这个时间是可以忽略的.而 iTwigStack 采用导航方式在索引上进行过滤,XMark 上用时最多的是 QX2,需要 141ms,Treebank 上用时最多的是 QT4,需要 40 968ms.虽然 SegSJ 可过滤掉所有查询无关的元素,但基于 F&B 索引过滤,其处理的索引结点的数量远多于 1-index 索引的结点数量,尤其是 Treebank 数据集,几乎每个元素对应一个 F&B 索引结点,过滤等同于对元素的二次操作.

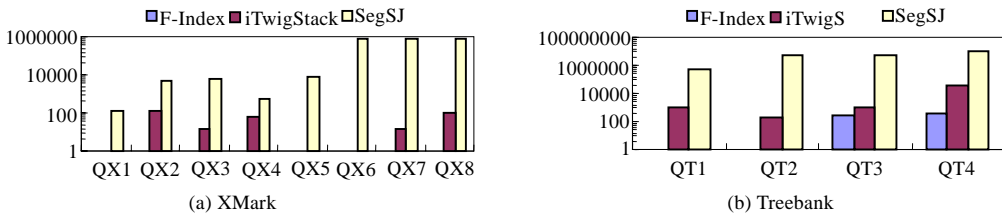


Fig.10 Comparison of filtering time

图 10 过滤时间对比

7.3.3 基于 F-Index 过滤后的查询时间

从图 11(a)和图 11(b)来看,F-Index 所用时间都远少于 iTwigStack 和 SegSJ,原因是 SegSJ 的查询实际用的是 TwigStack 算法,而 iTwigStack 使用的是变异的 TwigStack 算法,详细原因参见第 5.2 节的分析.

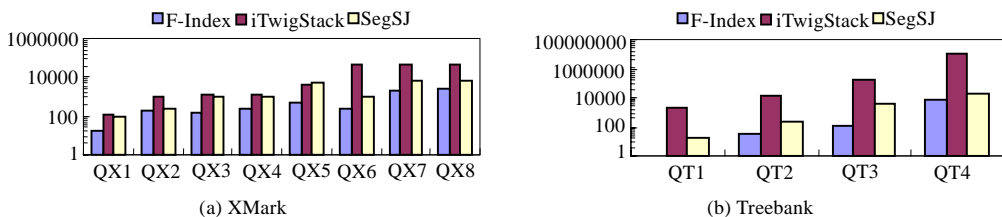


Fig.11 Comparison of running time after filtering

图 11 过滤后的查询时间对比

7.3.4 基于 F-Index 的查询算法的整体性能

整体性能是指完成一个查询需要的过滤时间和过滤后的查询用时的总和.从图 12(a)及图 12(b)可以看出,由于采用了 F-Index 来加速过滤,且有本文提出的查询算法的协助,基于 F-Index 的查询算法的整体性能远好于

其他 3 种算法,其中过滤用时及过滤后的查询用时均已在前面展示.

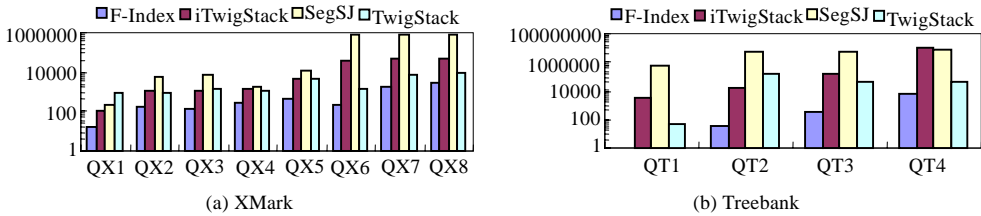


Fig.12 Comparison of the running time of different query algorithms

图 12 不同查询算法的运行时间对比

7.3.5 基于 F-Index 的查询算法的扩展性

图 13 选取了具有代表性的两种查询(PC-twig,AD-twig)来说明基于 F-Index 的查询算法的扩展性.可以看出,基于 F-Index 的查询算法随着数据集的增大,所用查询时间均少于其他 3 种算法.

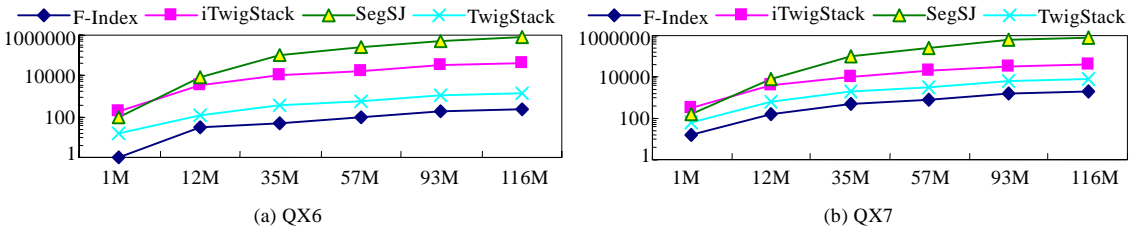


Fig.13 Comparison of the scalability of different query algorithm

图 13 不同查询算法的扩展性对比

7.3.6 实验结果分析

过滤算法在简单结构索引上的时间开销是可以忽略不计的,而结构简单的文档其索引结点的嵌套可能性很低,大部分情况下不存在递归的情况.因此,我们提出的查询算法可以获得优于 TwigStack 的查询性能.当结构索引变得很大时,过滤时间不可忽略,但从实验结果来看,F-Index 可以很好地适应结构极度复杂的文档,其过滤时间相对于 iTwigStack 和 SegSJ 来说是可以忽略的.由于本文提出的查询处理方法可以快速提取满足条件的结果序列,与 F-Index 结合进行查询处理,可以获得非常好的处理性能.

8 结 论

本文在深入分析国内外最新工作的基础上,针对已有的基于过滤进行查询的方法在遇到复杂结构 XML 文档时效率下降的问题,提出了一种用于加速查询处理的通用扁平结构索引 F-Index,可以快速过滤查询无关的数据,从而加速查询执行过程,尤其是当所处理的文档具有深度嵌套的复杂结构时,F-Index 的优势得以充分体现.针对过滤后索引结点之间已经满足查询要求的特点,提出一种有效查询算法用于处理过滤后剩余元素的查询问题.基于不同数据集的实验结果表明,本文提出的基于 F-Index 的查询处理方法可以极大地提高查询处理的性能.

致谢 感谢王小峰、张新、陈妍在解决本文研究问题的过程中给出的富有建设性的建议.

References:

[1] McHugh J, Widom J. Query optimization for XML. In: Malcolm PA, Maria EO, Patrick V, Stanley BZ, Michael LB, eds. Proc. of the 25th Int'l Conf. on Very Large Data Bases (VLDB). Edinburgh: Morgan Kaufmann Publishers, 1999. 315-326.

- [2] Michael J, Franklin MJ. Efficient filtering of XML documents for selective dissemination of information. In: Amr EA, Michael LB, Sharma C, Umeshwar D, Nabil K, Gunter S, Kyu YW, eds. Proc. of the 26th Int'l Conf. on Very Large Data Bases (VLDB). Cairo: Morgan Kaufmann Publishers, 2000. 53–64.
- [3] Gottlob G, Koch C, Pichler R. Efficient algorithms for processing XPath queries. In: Stéphane B, Akmal BC, Mong LL, Jeffrey XY, Zoé L, eds. Proc. of the 28th Int'l Conf. on Very Large Data Bases (VLDB). Hong Kong: Morgan Kaufmann Publishers, 2002. 95–106.
- [4] Halvreson A, Burger J, Galanis L, Kini A, Krishnamurthy R, Rao AN, Tian F, Viglas SD, Wang Y, Naughton JF, DeWitt DJ. Mixed mode XML query processing. In: Johann CF, Peter CL, Serge A, Michael JC, Patricia GS, Andreas H, eds. Proc. of the 29th Int'l Conf. on Very Large Data Bases (VLDB). Berlin: Morgan Kaufmann Publishers, 2003. 225–236.
- [5] Lu SC, Meng XF, Lin C, Wang Y. Navigation implementation for XQuery in OrientX. Journal of Computer Research and Development, 2004,41(10):1815–1822 (in Chinese with English abstract).
- [6] Zhang C, Naughton J, DeWitt D, Luo Q, Lohman G. On supporting containment queries in relational database management systems. In: Aref WG, ed. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD). ACM, 2001. 425–436.
- [7] Li Q, Moon B. Indexing and querying XML data for regular path expressions. In: Peter MGA, Paolo A, Stefano C, Stefano P, Kotagiri R, Richard TS, eds. Proc. of the 27th Int'l Conf. on Very Large Data Bases (VLDB). Rome: Morgan Kaufmann Publishers, 2001. 361–370.
- [8] Al-Khalifa S, Jagadish HV, Koudas N, Patel JM, Srivastava D, Wu Y. Structural joins: A primitive for efficient XML query pattern matching. In: Agrawal R, Dittrich K, Ngu AHH, eds. Proc. of the 18th Int'l Conf. on Data Engineering (ICDE). San Jose: IEEE Computer Society, 2002. 141–152.
- [9] Chien SY, Vagena Z, Zhang D, Tsotras VJ, Zaniolo C. Efficient structural joins on indexed XML documents. In: Stéphane B, Akmal BC, Mong LL, Jeffrey XY, Zoé L, eds. Proc. of the 28th Int'l Conf. on Very Large Data Bases (VLDB). Hong Kong: Morgan Kaufmann Publishers, 2002. 263–274.
- [10] Jiang H, Lu H, Wang W, Ooi BC. XR-Tree: Indexing XML data for efficient structural joins. In: Umeshwar D, Krithi R, Vijayaraman TM, eds. Proc. of the 19th Int'l Conf. on Data Engineering (ICDE). Bangalore: IEEE Computer Society, 2003. 253–264.
- [11] Lam F, Shui WM, Fisher DK, Wong RK. Skipping strategies for efficient structural joins. In: Yoonjoon L, Jianzhong L, Kyuyoung W, Doheon L, eds. Proc. of the Database Systems for Advances Applications (DASFAA). Jeju Island: Springer-Verlag, 2004. 196–207.
- [12] Wang J, Meng XF, Wang S. Structural join of XML based on range partitioning. Journal of Software, 2004,15(5):720–729 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/15/720.htm>
- [13] Wang J, Meng XF, Wang Y, Wang S. Target node aimed path expression processing for XML data. Journal of Software, 2005,16(5):827–837 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/16/827.htm>
- [14] Bruno N, Koudas N, Srivastava D. Holistic twig joins: Optimal XML pattern matching. In: Michael JF, Bongki M, Anastassia A, eds. Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD). Madison: ACM, 2002. 310–321.
- [15] Jiang HF, Wang W, Lu H, Yu JX. Holistic twig joins on indexed XML documents. In: Johann CF, Peter CL, Serge A, Michael JC, Patricia GS, Andreas H, eds. Proc. of the 29th Int'l Conf. on Very Large Data Bases (VLDB). Berlin: Morgan Kaufmann Publishers, 2003. 273–284.
- [16] Moro MM, Vagena Z, Tsotras VJ. Tree-Pattern queries on a lightweight XML processor. In: Klemens B, Christian SJ, Laura MH, Martin LK, Per-Ake L, Beng CO, eds. Proc. of the 31st Int'l Conf. on Very Large Data Bases (VLDB). Trondheim: ACM, 2005. 205–216.
- [17] Goldman R, Widom J. DataGuides: Enabling query formulation and optimization in semistructured databases. In: Matthias J, Michael JC, Klaus RD, Frederick HL, Pericles L, Manfred AJ, eds. Proc. of 23rd Int'l Conf. on Very Large Data Bases (VLDB). Athens: Morgan Kaufmann Publishers, 1997. 436–445.
- [18] Milo T, Suciu D. Index structures for path expressions. In: Catriel B, Peter B, eds. Proc. of the Int'l Conf. on Database Theory (ICDT). Jerusalem: Springer-Verlag, 1999. 277–295.

- [19] Kaushik R, Bohannon P, Naughton JF, Korth HF. Covering indexes for branching path queries. In: Michael JF, Bongki M, Anastassia A, eds. Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data. Madison: ACM (SIGMOD), 2002. 133–144.
- [20] Barta A, Consenc MP, Mendelzon AO. Benefits of path summaries in an XML query optimizer supporting multiple access methods. In: Klemens B, Christian SJ, Laura MH, Martin LK, Per-Ake L, Beng CO, eds. Proc. of the 31st Int'l Conf. on Very Large Data Bases (VLDB). Trondheim: ACM, 2005. 133–144.
- [21] Chen T, Lu JH, Ling TW. On boosting holism in XML twig pattern matching using structural indexing techniques. In: Fatma Ö, ed. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD). Baltimore: ACM, 2005. 455–466.
- [22] Wang W, Wang HZ, Lu HJ, Jiang HF, Lin XM, Li JZ. Efficient processing of XML path queries using the disk-based F&B index. In: Klemens B, Christian SJ, Laura MH, Martin LK, Per-Ake L, Beng CO, eds. Proc. of the 31st Int'l Conf. on Very Large Data Bases (VLDB). Trondheim: ACM, 2005. 145–156.
- [23] Hou S, Jacobsen HA. Predicate-Based filtering of XPath expressions. In: Ling L, Andreas R, Kyuyoung W, Jianjun Z, eds. Proc. of the 22nd Int'l Conf. on Data Engineering (ICDE). Atlanta: IEEE Computer Society, 2006. 53–64.
- [24] Miklau G. XMLdata repository. 2002. <http://www.cs.washington.edu/research/xmldatasets>

附中文参考文献:

- [5] 陆世潮,孟小峰,林灿,王宇. OrientX 中的 XQuery 的导航式处理. 计算机研究与发展, 2004, 41(10): 1815–1822.
- [12] 王静,孟小峰,王珊. 基于区域划分的 XML 结构连接. 软件学报, 2004, 15(5): 720–729. <http://www.jos.org.cn/1000-9825/15/720.htm>
- [13] 王静,孟小峰,王宇,王珊. 以目标节点为导向的 XML 路径查询处理. 软件学报, 2005, 16(5): 827–837. <http://www.jos.org.cn/1000-9825/16/827.htm>



周军锋(1977 -),男,陕西西安人,博士生,讲师,主要研究领域为 XML 数据库.



蒋瑜(1981 -),男,硕士生,主要研究领域为 XML 数据库.



孟小峰(1964 -),男,博士,教授,博士生导师,CCF高级会员,主要研究领域为 Web 数据集成,XML 数据库,移动数据管理.



谢敏(1984 -),男,硕士生,主要研究领域为 XML 数据库.