ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.7, July 2009, pp.1812−1824
doi: 10.3724/SP.J.1001.2009.03312

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种支持高效 XML 路径查询的自适应结构索引*

张 博[1+], 耿志华[1], 周傲英[1,2]

[1](复旦大学 计算机科学与工程系,上海 200433)

[2](华东师范大学 海量计算研究所,上海 200062)

Adaptive Structural Index for Efficient Processing of XML Path Queries

ZHANG Bo[1+], GENG Zhi-Hua[1], ZHOU Ao-Ying[1,2]

[1](Department of Computer Science and Engineering, Fudan University, Shanghai 200433, China)

[2](Institute of Massive Computing, East China Normal University, Shanghai 200062, China)

+ Corresponding author: E-mail: zhangbo@fudan.edu.cn

Zhang B, Geng ZH, Zhou AY. Adaptive structural index for efficient processing of XML path queries. *Journal of Software*, 2009,20(7):1812−1824. http://www.jos.org.cn/1000-9825/3312.htm

Abstract: This paper proposes an adaptive structural index: AS-Index (adaptive structural index), which can avoid the problem of the existing indexes. AS-Index is based on F&B-Index. It consists of F&B-Index, Query-Table and Part-Table. Frequent queries are kept in Query-Table avoiding redundant operations in query processing. Based on Query-Table an efficient bottom-up query processing is also proposed for answering infrequent queries using the frequent queries in Query-Table. Part-Table is used for optimizing the queries with descendant edges. The existing adaptive structural indexes need to traverse the whole document for adaptation, and their adaptation granularity is XML element node. For AS-Index, the adaptation granularity is F&B-Index node which includes a set of XML element nodes, and its adaptation is an efficient and incremental process that supports branch queries. The experimental results demonstrate that this index significantly outperforms the previous structural indexes in terms of query processing and adaptation efficiencies. For large XML documents, compared with the existing adaptive structural indexes, AS-Index is more scalable

Key words: XML; adaptive index; structural index; query processing

摘 要: 提出了一种新的自适应结构索引:AS-Index(adaptive structural index),能够克服现有静态索引和自适应索引的缺陷,具备高效的查询和调整性能.AS-Index 建立在 F&B-Index 的基础之上,其索引结构包括 F&B-Index,Query-Table 和 Part-Table.Query-Table 能够记录频繁查询,避免了查询过程中的冗余操作.并且,在 Query-Table 的基础上提出了自底向上的查询处理过程,能够充分利用现有的频繁查询高效地回答非频繁查询.Part-Table 用于优化包含祖先后裔边的查询,进一步提高了查询性能.现有的自适应结构索引的调整粒度是 XML 元素节点,调整过程往往需要遍历整个文档.而 AS-Index 是基于 F&B-Index 节点的增量调整,其过程是局部的,高效的,并且能够支持复杂分支查询的调整.实验结果表明,AS-Index 在查询和调整性能上优于现有的 XML 结构索引.同时,相比于现有的自适应结构索

* Supported by the National Natural Science Foundation of China under Grant No.60673137 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2006AA01Z103 (国家高技术研究发展计划(863))

Received 2007-11-15; Accepted 2008-03-14

引,AS-Index 针对大规模文档具有更加优良的可扩展性.

关键词: XML;自适应索引;结构索引;查询处理
中图法分类号: TP311 **文献标识码:** A

XML 已经成为 Internet 上数据交换和集成的事实标准.随着应用领域的推广,XML 文档数量不断增多,用户对 XML 数据进行检索和查询的效率也提出了更高的要求.针对这一问题,该领域已经提出了多种适用于不同查询类型的 XML 索引.XML 索引是基于 XML 文档建立的数据结构,在该索引结构上执行查询可以降低查询的时间开销.

可以根据 XML 文档中的多种对象建立索引,包括文本、元素、属性以及结构等信息.如图 1 所示,XML 索引根据索引对象可以分为文本索引、元素索引和结构索引.其中,文本索引通过索引出现在 XML 文档中的文本信息来方便查询.例如,可以使用倒排表对 XML 文档建立文本索引.元素索引着眼于快速确定节点之间的父子或祖先-后代关系,以此缩小搜索的范围,避免遍历 XML 文档中不相关的元素节点.XML 结构索引对 XML 文档中的路径信息进行处理,此类索引一直是研究的热点.路径表达式的处理效率对 XML 的查询性能至关重要,通用的路径表达式处理技术是遍历 XML 文档中的所有可能路径,同时,在此过程中找出匹配路径表达式的所有元素.但是,这种计算方法需要穷举 XML 文档的所有路径,开销较大.结构索引(或称为路径索引)的思想则是根据 XML 文档中路径的相似性,把相关路径进行合并或重新组织.这种索引结构与原始 XML 文档相比,在不损失查询性能和查询能力的前提下具有更少的节点和路径.因此,结构索引能够减小查询搜索范围,显著地提高路径表达式的处理效率.

近年来,研究者提出了多种 XML 结构索引.图 1 列出了结构索引的发展流程.可以看到,XML 结构索引是研究热点.根据结构索引能否依照查询负载进行调整,可以把结构索引分成静态索引和自适应结构索引两种类型.

(1) 静态索引根据 XML 文档建立之后,不能根据查询负载进行调整

DataGuide[1]是半结构化数据库中所有路径简单、精确的结构概括,可以作为 XML 数据库的一个模式图.DataGuide 为从根节点出发的所有路径建立索引,在 DataGuide 中,每条标签路径是唯一的.1-Index[2]的建立过程是比较两个节点的标签和从根节点到此节点的入边路径是否相同,根据比较结果把节点分配到对应的分类中.Fabric[3]使用 patricia trie 来索引树结构数据中的路径和内容信息.Fabric 索引建立的方法是将数据中的路径编码为字符串,然后使用 designator(特殊的字符或字符串,由字母或者数字组成)对数据路径进行编码,再使用 patricia trie 组织编码信息.$A(k)$-Index[4]根据 XML 元素节点在路径上的局部相似性划分等价类,尽量把结构相似的节点分到一个组中,减小索引的大小.它通过参数 K 来控制生成索引的精细程度,K 值越大,分组越细致,当 K 值增大到无穷大时,得到的索引等价于 1-Index.$UD(K,L)$-Index[5]综合考虑节点的局部前向和后向相似关系来构建索引图,通过对前向相似度的支持,可以有效地提高分支路径查询效率.F&B-Index[6]作为分支路径查询的覆盖索引,其分支路径查询的性能较以前的索引有很大的提高.Disk-Based F&B-Index[7]则对 F&B-Index 进行了扩展,设计了减少 I/O 操作的磁盘布局方法.

(2) 自适应结构索引则根据查询负载的变化进行结构调整

APEX(adaptive path index for XML data)[8]是针对单支查询的自适应结构索引.此方法通过统计查询负载中频繁使用的路径进行调整,针对频繁单支查询具有很好的查询性能.$D(K)$-Index[9]根据查询负载的需求,为不同节点分配不同的相似度,并且规定,对于局部相似度为 K 的节点,其父节点的局部相似度最小为 K–1,这样就保证所有的查询都可以在 $D(K)$-索引中实现,而不必去访问原数据图.$M(K)$-Index[10]的提出是为了克服 $D(K)$-Index 过于精细的分组问题.与 $D(K)$-Index 一样,$M(K)$-Index 也使用后向的局部相似关系,允许不同的节点有不同的相似度.为了防止过度精细地分组,$M(K)$-Index 在构建和更新索引的过程中,综合考虑查询负载的变化和查询返回的结果.

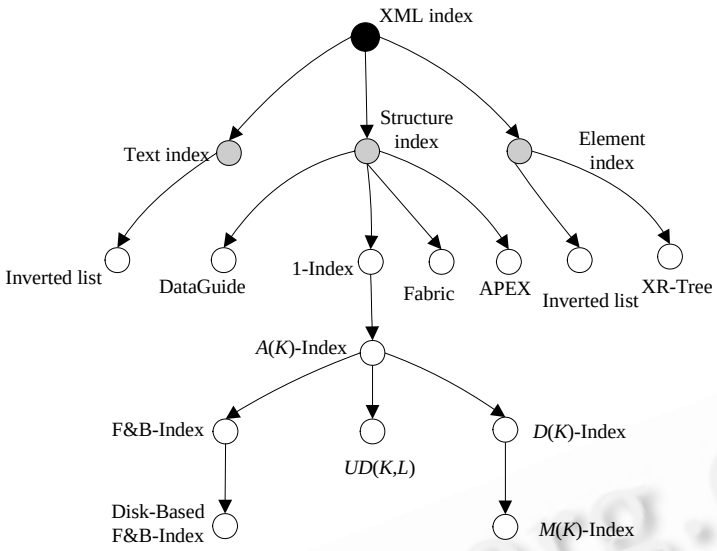


Fig.1 Category of XML index

图 1 XML 索引分类

以上部分对 XML 结构索引相关工作和发展思路进行了介绍.但是现有的结构索引还存在许多问题,影响了其查询性能和使用范围,主要问题列举如下:

(1) 静态结构索引的冗余操作

DataGuide,1-Index,Fabric,*A*(*k*)-Index 以及 F&B-Index 等是针对 XML 文档的静态索引.一方面,此类索引不考虑用户具体的查询负载,容易产生冗余操作.例如,当存在频繁查询时,针对相同的查询需要重复执行查询过程,影响了索引的查询性能;另一方面,针对含有祖先后裔边的查询,此类索引需要遍历当前节点的所有子孙节点,增大了查询开销.

(2) 自适应结构索引的局限性

APEX,*D*(*k*)-Index 以及 *M*(*k*)-Index 等自适应结构索引能够根据查询负载的变化对索引结构进行调整.但是,现有的自适应结构索引只能根据单支查询进行调整,而不能支持更加复杂的查询类型,例如分支查询.而且,现有的自适应结构索引不具备良好的可扩展性,其调整和查询只适用于较小的 XML 文档.对于规模较大的 XML 文档,例如 100M 的文档,现有的自适应结构索引的查询和调整性能会迅速降低,影响了自适应结构索引的应用范围.

本文针对现有的 XML 结构索引的问题,提出了一种新型的自适应结构索引:AS-Index.文中使用的 XML 查询语言为 XPath[11],针对其查询子集{/,//,[]}进行查询处理.AS-Index 是基于 F&B-Index 建立的,由于 F&B- Index 是支持分支查询的覆盖索引,这使得我们的 AS-Index 能够直接回答查询,而不用去访问 XML 文档.本文的主要贡献如下:

1) 本文提出了一种新型的自适应结构索引:AS-Index.AS-Index 由 F&B-Index,Query-Table 和 Part-Table 组成.Query-Table 可以直接回答频繁查询,避免了冗余操作.在 Query-Table 的基础上,提出了高效的自底向上的查询方法,能够充分利用现有的频繁查询来回答非频繁查询.Part-Table 用于存储频繁的查询部分,进一步提高了频繁查询的利用率,优化了包含祖先后裔边的查询.
2) 本文提出了针对 AS-Index 的调整算法.AS-Index 的调整过程是局部范围中的增量调整,其调整粒度是 F&B-Index 节点(包含一组 XML 元素节点),比以 XML 元素节点为粒度的现有的自适应结构索引具有更好的调整性能,而且能够支持含有分支查询的调整过程.
3) 针对基本的索引结构提出了一系列的优化策略,进一步提高了查询性能.

4) 通过实验验证,本文提出的 AS-Index 比现有的结构索引具有更加高效的查询和调整性能.针对较大的 XML 文档,AS-Index 具有很好的可扩展性.

1 预备知识

1.1 基本概念

本节介绍本文中使用到的相关概念,包括树模式、查询包含、主要路径和查询规模.

定义 1. 树模式(tree pattern):在 XPath 子集{/,//,[]}中的查询可以表示为一棵树模式[12],查询表达式中的每个标签可以表示为树模式中的一个节点,表达式中的轴表示为树模式中的对应边.

定义 2. 查询包含(query containment):对于两个查询 Q_1,Q_2.我们说 Q_1 包含 Q_2,当且仅当满足以下 3 个条件:

(1) 对于 Q_1 树模式中的每一个节点,在 Q_2 中存在一个节点与其对应.

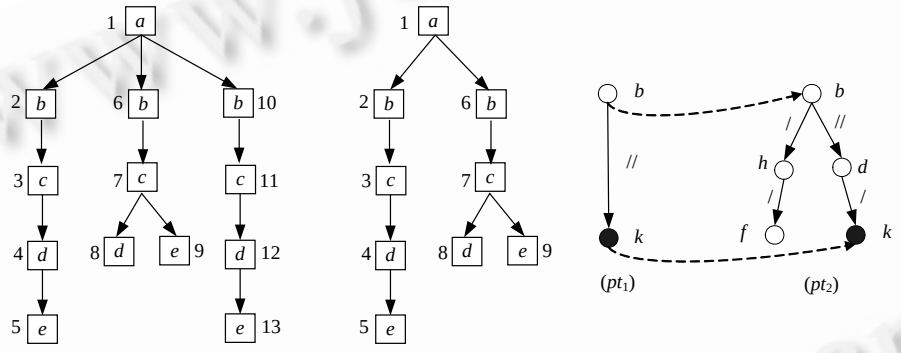
(2) 对于 Q_1 中的父子边,Q_2 中存在一个对应的父子边;对于 Q_1 中的祖先后裔边,Q_2 中存在一条路径与其对应.

(3) Q_1 的根节点和输出节点与 Q_2 的根节点和输出节点一一对应.

定义 3. 主要路径(main path):查询的树模式中,从根节点到输出节点的路径称为主要路径,主要路径上的节点数称为主要路径长度.

定义 4. 查询规模(query size):查询的树模式中的节点数称为此查询的查询规模.

例如,对于查询 Q_1=//*b*//*k*,Q_2=//*b*[*h*/*f*]//*d*/*k*.图 2(c)是这两个查询的树模式 pt_1,pt_2.Q_1 包含 Q_2,树模式 pt_1 和 pt_2 之间的箭头表示它们之间的包含映射.其中,Q_2 的查询规模=5,主要路径=//*b*//*d*/*k*,主要路径长度=3.



(a) XML data tree (b) F&B-Index (c) Tree patterns and homomorphism

(a) XML数据树 (b) F&B-Index (c) 树模态和同态

Fig.2 XML example

图 2 XML 实例

1.2 XML数据处理过程

我们在图3中介绍了AS-Index的数据处理过程,主要包括初始化(initialization)、查询处理(query processing)和调整(adaptation).初始化负责为 XML 数据建立结构索引并完成系统所需数据结构的初始化;查询处理是使用索引完成查询的过程;调整过程负责根据查询负载的变化情况来调整索引结构.AS-Index 由 F&B-Index,Query-Table 和 Part-Table 组成.Query-Table 和 Part-Table 分别用于存储频繁的查询和部分查询(或称为查询步骤).我们将在下文中具体介绍每个处理过程.

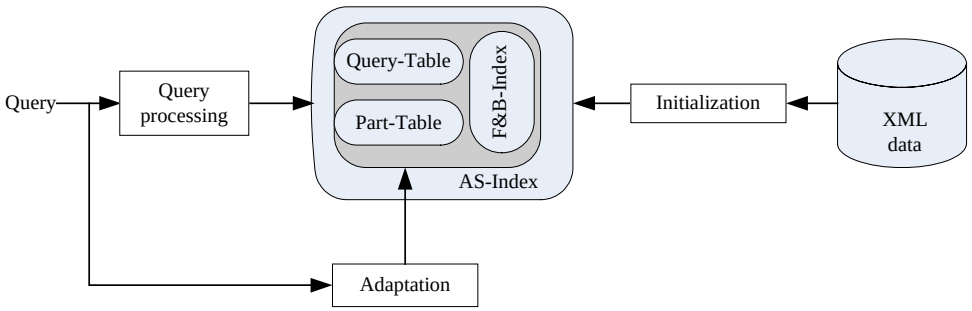


Fig.3 Data process of AS-Index

图 3 AS-Index 的数据处理过程

2 自适应结构索引 AS-Index

AS-Index 主要由 3 个部分组成:F&B-Index,Query-Table 和 Part-Table.

F&B-Index 是针对 XML 文档的最小覆盖索引.覆盖索引的概念源于关系数据库.当需要反复查询一个表中的数据时,数据库系统可以为这个表建立一个覆盖查询语句中所有属性的索引,这样可以通过该索引来进行查询,而不用去参考对应的表实例.图 2(b)是针对图 2(a)中 XML 文档的 F&B-Index.

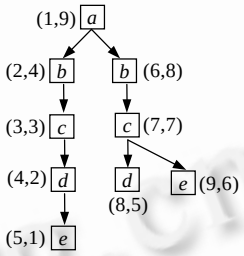
用户提出的频繁查询保存在 Query-Table 中.Query-Table 的结构如图 4(a)所示.Query-Table 是一个以频繁查询作为关键字的哈希表.每个关键字对应着孩子查询(children)和节点指针(pointers).孩子查询是这个查询包含的子查询,节点指针指向 F&B-Index 节点.由此结构可以知道,针对一个频繁查询 Q,其查询结果由两部分组成,一部分是其孩子查询中的 F&B-Index 节点,另一部分是 Q 的节点指针直接指向的 F&B-Index 节点.在保存频繁查询的时候,同时为相互包含且具有相同输出节点的频繁查询建立包含层次关系.例如,在图 4(a)所示的 QueryTable 中,查询//d 包含查询//c/d,且具有相同的输出节点,因此将查询//c/d 作为//d 的孩子查询.

Id	Query	Children	Pointers
1	//*a*		(1)
2	//*b*		(2)(6)
3	//*c*		(3)(7)
4	//*d*	{6}	
5	//*e*		(5)(9)
6	//*c*/*d*		(4)(8)

(a) Query-Table

QPart	PreCode	SufCode	Pointers
a	(1,9)		(1)
b	(2,4)(6,8)		(2)(6)
c	(3,3)(7,7)		(3)(7)
d	(4,2)(8,5)		(4)(8)
e	(5,1)(9,6)		(5)(9)
c/*e*	(9,6)		(4)(8)

(b) Part-Table



(c) F&B-Index

Fig.4 AS-Index example

图 4 AS-Index 实例

我们把查询中的部分查询称为一个查询部件,频繁的查询部件存储在 Part-Table 中.Part-Table 的结构如图 4(b)所示.Part-Table 是一个哈希表,其中以查询部件(QPart)作为关键字.每个关键字对应一组前缀编码(PreCode)、一组后缀编码(SufCode)和对应节点指针.当一个查询部件主要用于向下查询过程中的时候,存储的是此部件对应的前缀编码,即符合这个部件查询的最后一个 label 的一组编码,并保存满足查询部件的最后一个标签的节点指针;如果一个查询部件用于向上验证的过程,则存储此部件的后缀编码,即符合这个查询部件的第 1 个 label 的一组编码,并保存满足查询部件的第 1 个标签的节点指针.如图 4(b)所示,只含有一个标签的部件查询一直保存在 Part-Table 中,这时候,前缀编码与后缀编码相同,把对应的编码保存在 PreCode 中.“*c*/*e*”是一个频繁的查询部件,主要用于向下的查询过程,把符合此查询部件的节点 e 的编码保存在前缀编码中.

3 AS-Index 的生成和调整

3.1 初始化过程

在初始化过程中,我们介绍 AS-Index 的生成方法.首先建立 F&B-Index,使用⟨start,end⟩编码[13]对 F&B-Index 中的节点进行编码.然后对 Query-Table 和 Part-Table 进行初始化.在初始化过程中,系统首先把形如“//label”的查询作为默认的频繁查询,加入到 Query-Table 中,并把形如“label”的部分查询和对应的编码加入到 Part-Table 中.初始化过程需要遍历 F&B-Index,把具有相同标签的节点的指针添加到 Query-Table 和 Part- Table 的对应查询中.在图示中,使用节点的 start 编码来表示指向每个节点的指针.例如在图 4(a)中,对于查询//*b*,其查询结果为图 4(c)中的两个节点 *b*,对应的 start 编码为 2 和 6.在 Part-Table 中,对于形如“label”的查询部件,由于其前缀编码和后缀编码相同,只需记录其前缀编码.在调整过程中,认为形如“//label”的查询和形如“label”的查询部件始终是频繁的,因此需要一直保存在 Query-Table 和 Part-Table 中.

3.2 针对Query-Table的调整

根据用户的查询情况,需要对 Query-Table 进行调整,调整的过程分为两个阶段:添加新的频繁查询和删除旧的频繁查询.

添加新的频繁查询

添加新查询的任务主要是调整 Query-Table 中查询的包含关系,然后将新查询添加到 Query-Table 中.算法如 Algorithm 1 所示,对于一个新查询 Q_n,首先找到第 1 个包含此查询,且具有相同输出节点的查询 Q_{first}.然后递归地访问 Q_{first} 的孩子查询,只要找到包含 Q_n 的查询,就探索此查询的孩子查询,一直找到一个查询 Q_{final},它的所有孩子查询都不包含 Q_n.

然后需要测试 Q_{final} 中的 F&B-Index 节点,如果节点满足查询 Q_n,将此节点加入到 Q_n 的结果中,并从 Q_{final} 中删除.对于 Q_{final} 的每个孩子查询,如果其包含的所有 F&B-Index 节点都符合 Q_n,就把此查询添加为 Q_n 的孩子查询.如果是部分节点符合 Q_n,这部分 F&B-Index 节点添加到 Q_n 的结果中.最后把 Q_n 添加为 Q_{first} 的孩子查询.

Algorithm 1. ***UpdateQueryTable***(Q_n).

1 找到第 1 个包含 Q_n 的查询 Q_{first},使得 *outputNode*(Q_n)==*outputNode*(Q_{first})
2 访问 Q_{first} 包含的孩子查询,每次只要找到第 1 个包含 Q_n 的孩子查询 Q_1,我们就继续访问 Q_1 的孩子查询,直到找到一个查询 Q_{final},Q_{final} 的孩子查询中没有包含 Q_n 的查询.
3 For 每个 F&B-Index 节点 $N_i \in Result(Q_{final})$ do
4 If N_i 符合查询 Q_n then
5 添加 N_i 到 *Result*(Q_n),从 *Result*(Q_{final})中删除 N_i
6 For 每个孩子查询 $SubQuery \in Result(Q_{final})$ do
7 If *SubQuery* 中所有的 F&B-Index 节点符合 Q_n do
8 添加 *SubQuery* 为 Q_n 的孩子查询,并从 Q_{final} 的孩子查询中删除 *SubQuery*
9 ElseIF *SubQuery* 中部分 F&B-Index 节点符合 Q_n do
10 把符合 Q_n 的部分 F&B-Index 节点添加到 *Result*(Q_n)中
11 添加 Q_n 为 Q_{first} 的孩子查询

删除旧的频繁查询

当 Query-Table 中的查询不再是频繁查询的时候,需要把它从 Query-Table 中删除.对于一个待删除查询 Q_d,首先找到包含 Q_d 的查询 Q_p,然后将 Q_d 含有的 F&B-Index 节点和孩子查询添加到 Q_p 的结果中,然后从 Q_p 的孩子查询中删除 Q_d.

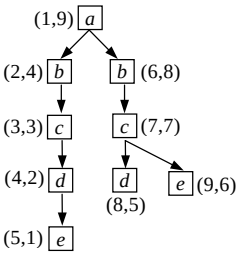
例如,图 5(a)的 Query-Table 是在图 4(a)的基础上添加了频繁查询//*c*/*e*,删除了查询//*c*/*d*.

Id	Query	Children	Pointers
1	//*a*		(1)
2	//*b*		(2)(6)
3	//*c*		(3)(7)
4	//*d*		(4)(8)
5	//*e*	{6}	(5)
6	//*c*/*e*		(9)

(a) Query-Table

QPart	PreCode	SufCode	Pointers
a	(1,9)		(1)
b	(2,4)(6,8)		(2)(6)
c	(3,3)(7,7)		(3)(7)
d	(4,2)(8,5)		(4)(8)
e	(5,1)(9,6)		(5)(9)
b/*c*/*e*		(6,8)	(6)

(b) Part-Table



(c) F&B-Index

Fig.5 AS-Index after adaptation

图 5 调整后的 AS-Index

3.3 针对Part-Table的调整

根据用户的查询情况,每个阶段需要对 Part-Table 进行调整,由于其中存储的查询部件相互之间没有关联,调整过程比较直观,即插入新的频繁查询部件,然后删除非频繁查询部件.例如,图 5(b)的 Part-Table 是在图 4(b)的基础上添加了查询部件 *b*/*c*/*e*,删除了查询部件 *c*/*e*.

4 XML 查询处理

本节介绍 AS-Index 支持的查询处理方法,包括基于遍历操作的查询和基于联接操作的查询.

4.1 基于遍历操作的查询处理

Query-Table 用来存储频繁查询,它具有两个功能:一方面,当用户提出的查询是频繁查询的时候,可以在 Query-Table 中直接找到查询结果,避免了在 F&B-Index 上的冗余遍历操作;另一方面,如果用户提出的查询不能在 Query-Table 中直接找到,可以在 Query-Table 中找到包含此查询的频繁查询,然后通过对频繁查询的结果进行验证,找到需要的结果.

使用 Query-Table 的查询过程如 Algorithm 2 所示.对于一个查询 Q_n,首先到 Query-Table 中确认 Q_n 是否为频繁查询,如果 Q_n 包含在 Query-Table 中,可以直接获得 Q_n 的查询结果.如果 Q_n 是非频繁查询,首先到 Query-Table 中找到查询 Max*Query*,Max*Query* 满足的条件是,自下向上在 Q_n 的主要路径上找到第 1 个与 Max*Query* 的输出节点具有相同标签的节点,然后根据此节点把 Q_n 分成前缀部分(prefix)和后缀部分(suffix),Max*Query* 包含前缀部分,并且 Max*Query* 是找到的主要路径最接近 Q_n 的查询.在 Query-Table 中可以直接获得 Max*Query* 的结果,即一组 F&B-Index 节点,然后再向上验证节点是否满足 Max*Query* 与 Q_n 的前缀部分的差异,即 Q^{up};如果满足,则再进一步以此 F&B-Index 节点为当前节点,执行后缀部分的查询 Q^{down}(Q^{down}=*suffix*),从而获得 Q_n 的结果.

例如:Q_n=//*b*/*c*/*d*/*e*,在如图 4 所示的 Query-Table 中找到 Max*Query*=//*c*/*d*,根据节点 *d*,把 Q_n 分成两部分,*prefix*=//*b*/*c*/*d*,*suffix*=/*e*.然后根据 Max*Query* 和 *prefix* 得到 Q^{up}=.\.*b*,Q^{down}=/*e*,首先验证 Max*Query* 中的 F&B-Index 节点是否符合 Q^{up},如果符合,再以此 F&B-Index 节点为当前节点,执行 Q^{down} 获得 Q_n 的最后结果.

Algorithm 2. *ExecuteQuery*(Q_n).

1. If Query-Table 包含查询 Q_n do
2. 返回 Q_n 的结果,程序终止
3. Max*Query*=*null*
4. For Query-Table 中每一个查询 Q_i do
5. *pos*=*LastMatch*(Q_n,*outputNode*(Q_i))
6. (*prefix*,*suffix*)=*SplitQuery*(Q_n,*pos*){自下向上找到 Q_n 主路径上的第 1 个与 Q_i 的输出节点相同的节点,

然后从此节点把 Q_n 分成 *prefix* 和 *suffix* 两个部分}

```
If Qi 包含 prefix then
  If MaxQuery==null then
    MaxQuery=Qi
  Else
    If MainLength(Qi)比 MainLength(MaxQuery)更接近 Qn then
      MaxQuery=Qi
Qup=GetComposingQuery(MaxQuery,Prefix)
Qdown=suffix
For MaxQuery 中每一个 F&B-Index 节点 Ni do
  If IsSatisfy(Ni,Qup)==True do
    Output Eval(Ni,Qdown)
```

4.2 基于联接操作的查询处理

上面我们使用 Query-Table 优化了频繁查询,但是在查询过程中,当每次遇到祖先后裔边的查询时,一方面,针对向下查询的过程,都需要遍历当前节点的所有子孙节点;另一方面,对于向上验证的过程,需要向上遍历所有的祖先节点.这样的操作成为查询过程的性能瓶颈.

在 AS-Index 初始化的时候,系统已经对 F&B-Index 进行了编码.在查询的过程中,每当遇到祖先后裔边时,把查询根据祖先后裔边进行分解,然后使用联接操作[14]来判断祖先后裔关系.比如两个节点 u 和 v,u 是 v 的祖先,当且仅当 $u.start<v.start$ 并且 $u.end>v.end$.对于父子边,继续使用导航操作来完成查询.在这个过程中分解出来的以祖先后裔边开始、中间只包含父子边的查询部分称为一个查询部件.在查询过程中,系统需要统计频繁的查询部件并存储在 Part-Table 中.

使用 Part-Table 进行查询的过程如 Algorithm 3 所示.在向上验证和向下查询的过程中,遇到父子边,使用遍历的方法;遇到祖先后裔边时,要到 Part-Table 中寻找对应的查询部件,然后使用文献[14]中的联接方法进行操作.在向上验证的过程中,使用祖先后裔边分离出查询部件,然后按照从长到短的顺序,依次在 Part-Table 中寻找此查询部件的后缀部分,找到相应后缀,使用后缀编码进行联接操作.在向下查询的过程中,同样按照祖先后裔边分离出查询部件,然后按照从长到短的顺序,依次在 Part-Table 中寻找此查询部件的前缀部分,找到相应前缀,使用前缀编码进行联接操作.

例如:对于一个查询 Q=//b[//c/e],当前的 AS-Index 如图 4 所示.首先在 Query-Table 中找到包含此查询的频繁查询 Q_m=//b,然后对 Q_m 结果中的每一个 F&B-Index 节点,需要向下查询,判断其是否满足分支谓词//c/e,由于遇到了祖先后裔边,抽取出查询部件 c/e,可以在 Part-Table 中找到对应的查询部件,并获得其前缀编码,通过联接操作,可以避免冗余的子树遍历.

Algorithm 3. *EvaluateWithPartTable*(*CurrentNode*).

```
If 向下查询 do
  If 遇到父子边 do
    向下导航遍历
  Else
    截取祖先后裔边之间的查询部件 Qpart
    依次在 Part-Table 中寻找 Qpart 的前缀
    获得对应前缀的前缀编码,进行联接操作
Else
  If 遇到父子边 do
    向上导航验证
```

11. Else
12. 截取祖先后裔边之间的查询部件 Q_{part}
13. 依次在 Part-Table 中寻找 Q_{part} 的后缀
14. 获得对应后缀的后缀编码,进行联接操作*

5 优化策略

本节针对现有的方法进一步优化包含判断和调整过程.

5.1 优化包含判断

在使用 AS-Index 进行查询和调整的过程中,经常需要判断两个查询之间的包含关系,而包含关系的判断需要在两棵树模式之间进行映射操作,时间开销较大.因此在这一部分,本文提出一些过滤方法,用来避免不必要的映射操作,以加速包含判断.

1) 查询规模:对于两个查询 Q_1 和 Q_2,其查询规模分别为 $size(Q_1)$和 $size(Q_2)$,Q_1 包含 Q_2 需要满足 $size(Q_1)\leq size(Q_2)$.
2) 主要路径长度:对于两个查询 Q_1 和 Q_2,其主要路径长度为 $mainlength(Q_1)$和 $mainlength(Q_2)$,Q_1 包含 Q_2 需要满足 $mainlength(Q_1)\leq mainlength(Q_2)$.

证明:由查询包含的定义可知,当查询 Q_1 包含 Q_2 时,Q_1 中的每个节点在 Q_2 中都有一个对应的节点,因此只有 $size(Q_1)\leq size(Q_2)$,Q_1 才可能包含 Q_2.同理,当 Q_1 包含 Q_2 时,Q_1 的主要路径和 Q_2 的主要路径是对应的,即 Q_1 的主要路径中的每个节点在 Q_2 的主要路径中都有对应的节点,因此只有 $mainlength(Q_1)\leq mainlength(Q_2)$,$Q_1$ 才可能包含 Q_2. □

在需要判断包含关系的时候,首先利用以上两个条件进行过滤,只有满足以上这两个条件的查询,才进一步进行包含判断.这样就避免了不必要的包含判断,进一步加速了查询和调整过程.

5.2 优化调整过程

在调整过程中,另一个比较耗时的过程是添加新的频繁查询到 Query-Table 中.在添加过程中,需要获得一个新查询 Q_n 的结果,并确定包含 Q_n 的查询,这个过程相当于对 Q_n 重新执行了一次查询.为了能够实现高效的调整过程,在对 Q_n 进行查询的时候,就记录 Q_n 包含的所有 F&B-Index 节点,并记录包含这些节点的拥有最少 F&B-Index 节点的查询,这个查询就是要找的 Q_{final}.这样就能快速定位 Q_{final},然后再判断 Q_{final} 的子孙查询与 Q_n 的关系.

6 实 验

6.1 实验环境

本文提出的 AS-Index 在 Java 1.4 环境中实现.实验环境为 Windows XP 平台,主频为 3.2.GHz 和 2G 内存.实验中将比较 AS-Index 与 F&B-Index 的查询性能,并比较 AS-Index 与 APEX 的查询和调整性能.APEX 是最好的自适应索引之一[8],其结构与 AS-Index 结构比较近似,对频繁查询使用一个哈希树结构进行存储,查询效率比较高.

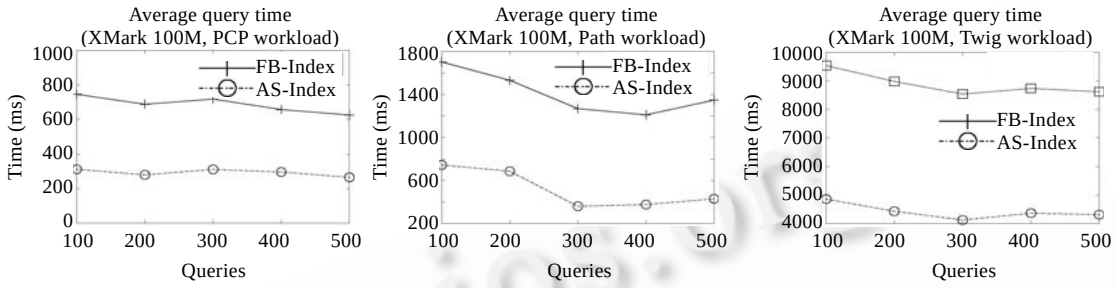
实验中使用的数据集为 XMark[15].XMark 是一个模拟拍卖网站的数据集.为了验证 AS-Index 对文档大小的适应性和可扩展性,实验中分别选用了 20M,50M 和 100M 的文档进行测试.本文使用的测试查询由 Y-Filter[16] 中的查询生成器自动产生.实验主要针对 3 种查询类型,分别是 PCP,Path 和 Twig 查询.PCP 查询是以祖先后裔边开始的单支查询,在查询表达式中只包含父子边;Path 查询是单支查询,祖先后裔边可以出现在查询表达式中的任何地方;Twig 查询表示分支查询.

实验的查询负载设置包括 500 个查询,其中以每 100 个查询为一组.在每组查询中,我们设定频繁查询的比

例为 60%,每两组查询之间有 30%的频繁查询互相重叠.每次提交一组查询,执行一组查询后,我们使用其中的频繁查询对索引结构进行调整.我们统计比较查询负载的平均查询和调整时间.

6.2 与静态索引的比较

图 6 表示在 100M 的 XMark 数据集上 F&B-Index 和 AS-Index 的查询性能.图 6(a)~图 6(c)分别针对 PCP,Path 和 Twig 这 3 种查询,横坐标为查询数目,每次执行 100 个查询.纵坐标为平均查询时间.如图 6 所示,通过使用本文提出的优化技术,AS-Index 对 3 种不同类型查询的查询性能都优于 F&B-Index.查询性能的提高,一方面是通过 Query-Table,这一结构能够非常高效地获得频繁查询的结果,而不用重复进行查询操作.对非频繁查询,也能从其包含查询中采用自底向上的查询过程,快速确定其中符合非频繁查询的结果;另一方面,通过使用 Part-Table,AS-Index 可以对包含祖先后裔边的查询进行优化,避免了对祖先或子孙节点的全局遍历.



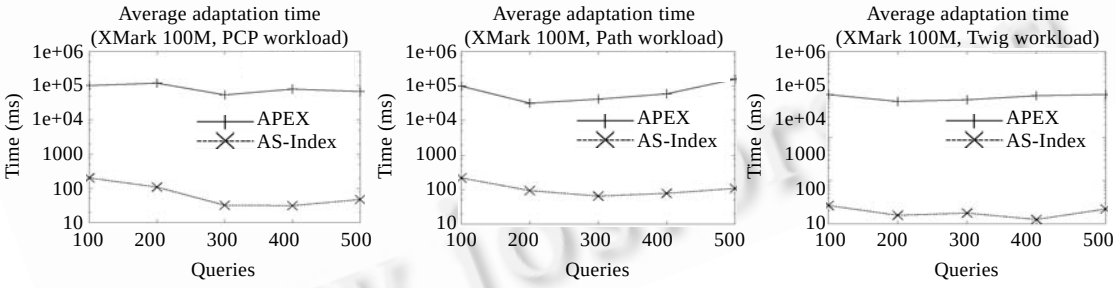
(a) PCP query (a) PCP 查询　(b) Path query (b) Path 查询　(c) Twig query (c) Twig 查询

Fig.6 Comparison of static index on evaluation

图 6 比较静态索引查询结果

6.3 与自适应结构索引的比较

图 7 是 AS-Index 和 APEX 在 3 种查询类型下的调整性能数据,横坐标为查询数目,每执行 100 个查询后调整 1 次.纵坐标为平均调整时间.因为 APEX 只能针对单支路径进行调整,所以,我们对 APEX 的调整方法进行了改进.针对 Twig 查询,我们首先将查询分解为多个单支查询,然后再分别根据分解后的单支查询进行调整.



(a) PCP adaptation (a) PCP 调整　(b) Path adaptation (b) Path 调整　(c) Twig adaptation (c) Twig 调整

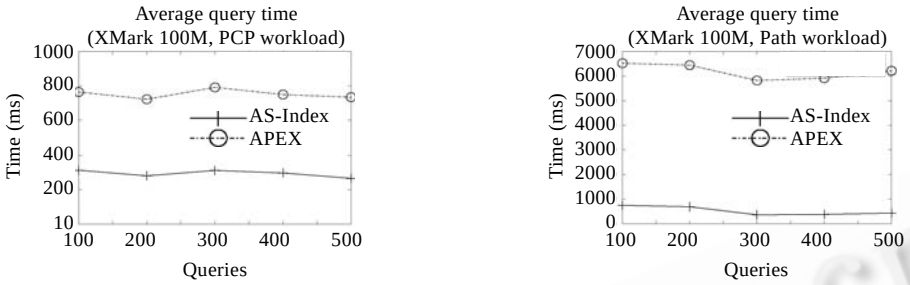
Fig.7 Comparison of dynamic index on adaptation

图 7 比较自适应结构索引调整结果

如图 7 所示,AS-Index 的调整性能优于 APEX.这是因为两种索引的调整范围和粒度不同.APEX 在调整过程中需要处理整个 XML 文档,调整粒度是针对 XML 文档中的元素节点.而 AS-Index 的优化技术则是建立在 F&B-Index 上,系统只需调整 Query-Table 中的层次关系和 Part-Table 中的对应项;AS-Index 的调整粒度是 F&B-Index 的节点,其数目要远远小于元素节点的数量.由于缩小了调整范围,所以 AS-Index 的系统性能得到了

极大的提升.

在图 8 中,我们分别测试比较了 AS-Index 与 APEX 的查询性能.因为 APEX 不支持分支查询,所以,我们只测试了 PCP 和 Path 两种查询类型.如图 8 所示,AS-Index 能够充分利用频繁查询来回答非频繁查询,而且针对 Path 查询类型,AS-Index 优化了祖先后裔边的查询过程.因此,AS-Index 的查询性能比 APEX 有了非常显著的提高.



(a) PCP query
(a) PCP 查询

(b) Path query
(b) Path 查询

Fig.8 Comparison of dynamic index on evaluation

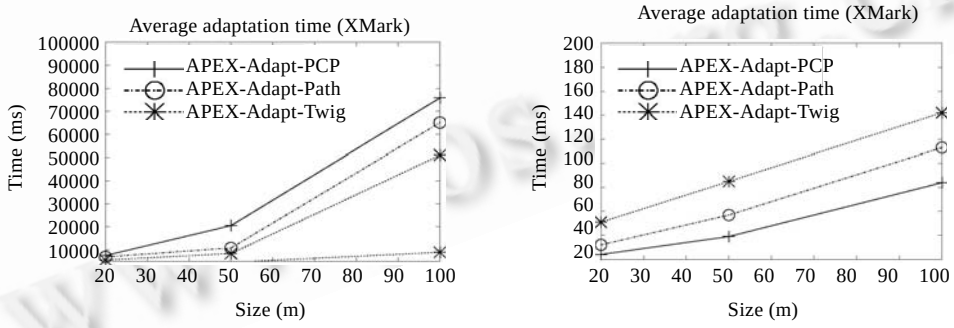
图 8 比较自适应结构索引查询结果

6.4 可扩展性

图 9 和图 10 是 AS-Index,APEX 和 F&B-Index 这 3 种索引结构针对 XML 文档大小变化的可扩展性比较.实验中分别测试了在 20M,50M 和 100M 的 XMark 文档上的平均查询和调整时间.

图 9 中,我们测试了 AS-Index 和 APEX 的调整性能的可扩展性.APEX 和 AS-Index 的调整时间都随着 XML 文档的增大而增加,而 AS-Index 的增加幅度要小于 APEX.由以上介绍可知,AS-Index 是针对 F&B-Index 节点而作的局部调整,而 APEX 的调整过程是针对 XML 文档元素节点的全局判断.因此,AS-Index 在调整性能上具有更好的可扩展性.

图 10 中,我们测试了 AS-Index 和 F&B-Index 查询的可扩展性.随着文档的增大,AS-Index 和 F&B-Index 的查询时间都在增长.但是,AS-Index 的增加幅度明显小于 F&B-Index.这是因为,频繁查询能够直接在 Query-Table 中找到结果,而不受文档大小的影响.而且,本文提出的针对祖先后裔边的查询优化也显著减少了不必要的遍历操作,提高了系统性能.

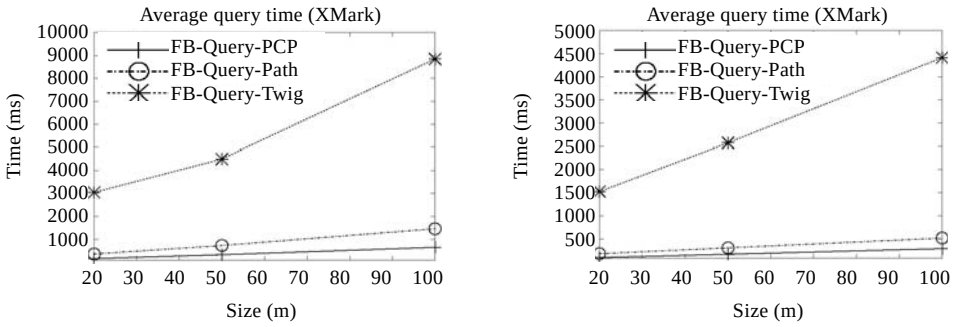


(a) APEX adaptation
(a) APEX 调整性能

(b) AS-Index adaptation
(b) AS-Index 调整性能

Fig.9 Scalability of adaptation

图 9 调整操作的可扩展性



(a) F&B-Index query
(a) F&B-Index 查询性能

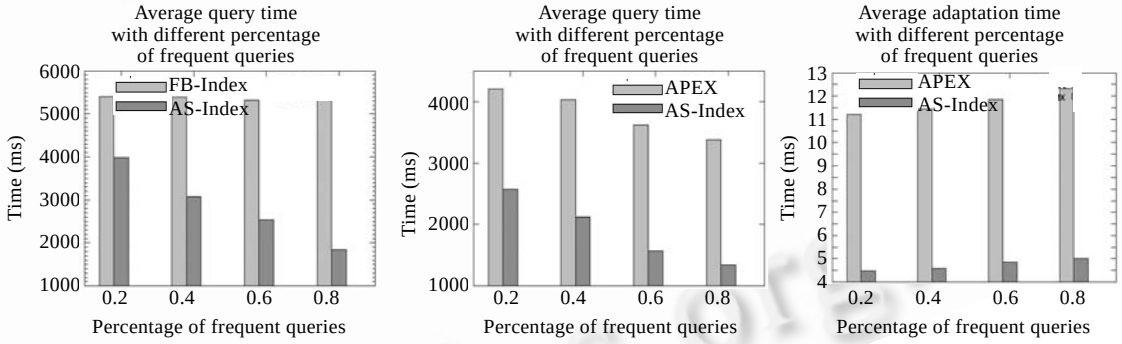
(b) AS-Index query
(b) AS-Index 查询性能

Fig.10 Scalability of evaluation

图 10 查询操作的可扩展性

6.5 不同程度的频繁查询测试

图 11 中,分别在 100M 的 XMark 文档上测试了 4 组不同频繁程度的查询负载的查询和调整结果.每组包含 500 个查询,频繁查询的比例分别为 20%,40%,60%和 80%.在图 11(a)中,因为 F&B-Index 没有针对频繁查询的优化过程,因此不受频繁查询比例的影响.而 AS-Index 的查询时间随频繁查询比例的增加而减少.在图 11(b)中可以看到,APEX 和 AS-Index 的查询性能随着频繁查询比例的增加而减小,AS-Index 的减少程度要比 APEX 更明显.这是因为 AS-Index 不仅能够优化频繁查询,而且能够优化非频繁查询.为了方便图示,我们在图 11(c)中使用调整时间的对数值加以对比.可以看到,AS-Index 和 APEX 的调整性能随着频繁查询比例的增加而增大,这是因为频繁查询比例越高,需要调整的查询数目就越多.由于 AS-Index 的调整算法更加高效,故 AS-Index 调整时间的增长幅度要小于 APEX.



(a) F&B-Index and AS-Index query
(a) F&B-Index 和 AS-Index 查询性能

(b) APEX and AS-Index query
(b) APEX 和 AS-Index 查询性能

(c) APEX and AS-Index adaptation
(c) APEX 和 AS-Index 调整性能

Fig.11 Evaluation result with different percentage of frequent queries

图 11 不同频繁程度查询的测试结果

从上面的实验可以看出,由于针对频繁的查询和查询部件进行了统计处理,AS-Index 的查询和调整性能明显优于已有的几种结构索引,并且即使针对较大的 XML,也具有非常良好的可扩展性.

7 结 论

XML 查询处理是近年来数据库领域的研究热点,如何设计高效的索引结构是研究者长期以来一直关注的主题.但是,现有的 XML 结构索引存在冗余的查询操作和低效的调整过程等问题,影响了 XML 结构索引的性能

和广泛应用.

本文针对以上问题提出了一种新型的自适应结构索引:AS-Index.AS-Index 可以反映用户的频繁查询,避免冗余的查询操作,并针对祖先后裔边的情况进行了查询优化.而且,针对低效调整的问题,本文为 AS-Index 设计了高效的调整算法.实验结果表明,AS-Index 在查询性能和调整性能上都优于现有的静态索引和自适应结构索引,并具有很好的可扩展性.

References:

[1] Goldman R, Widom J. DataGuides: Enabling query formulation and optimization in semistructured databases. In: Proc. of the VLDB. 1997. 436−445.

[2] Milo T, Suciu D. Index structures for path expressions. In: Proc. of the ICDT. 1999. 277−295.

[3] Cooper BF, Sample N, Franklin MJ, Hjaltason GR, Shadmon M. A fast index for semistructured data. In: Proc. of the VLDB. 2001. 341−350.

[4] Kaushik R, Shenoy P, Bohannon P, Gudes E. Exploiting local similarity for indexing paths in graph-structured data. In: Proc. of the ICDE. 2002. 129−140.

[5] Wu HW, Wang Q, Yu JX, Zhou AY, Zhou SG. *UD*(*k*,*l*)-Index: An efficient approximate index for XML data. In: Proc. of the 4th Int'l Conf. on Web Age Information Management (WAIM 2003). LNCS, Chengdu: Spring-Verlag, 2003.

[6] Kaushik R, Bohannon P, Naughton JF, Korth HF. Covering indexes for branching path queries. In: Proc. of the SIGMOD. 2002. 133−144.

[7] 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: Proc. of the VLDB. 2005. 145−156.

[8] Chung CW, Min JK, Shim K. APEX: An adaptive path index for XML data. In: Proc. of the SIGMOD. 2002. 121−132.

[9] Chen Q, Lim A, Ong KW. *D*(*k*)-Index: An adaptive structural summary for graph-structured data. In: Proc. of the SIGMOD. 2003. 134−144.

[10] He H, Yang J. Multiresolution indexing of XML for frequent queries. In: Proc. of the ICDE. 2004. 683−694.

[11] Berglund A, Boag S, Chamberlin D, Fernandez MF, Kay M, Robie J, Simeon J. XML path language (XPath) 2.0. In: Proc. of the W3C Proposed Recommendation. 2006. http://www.w3.org/TR/xpath20

[12] Miklau G, Suciu D. Containment and equivalence for an XPath fragment. In: Proc. of the PODS. 2002. 65−76.

[13] Zhang C, Naughton JF, DeWitt DJ, Luo Q, Lohman GM. On supporting containment queries in relational database management systems. In: Proc. of the SIGMOD. 2001. 425−436.

[14] Al-Khalifa S, Jagadish HV, Patel JM, Wu Y, Koudas N, Srivastava D. Structural joins: A primitive for efficient XML query pattern matching. In: Proc. of the ICDE. 2002. 141−154.

[15] XMark. http://monetdb.cwi.nl/xml

[16] YFilter 1.0 release. http://yfilter.cs.berkeley.edu/code_release.htm

张博(1979－),男,天津人,博士生,主要研究领域为 Web/XML 数据管理.

耿志华(1984－),男,硕士生,主要研究领域为 Web/XML 数据管理.

周傲英(1965－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为海量数据管理,Web 搜索和挖掘,Web 服务,P2P 计算.