文章编号:1001 - 9081(2008)04 - 0869 - 03

Native XML 数据库的结构连接算法研究

张 鹏[1], 冯建华[1], 房志峰[2]
(1. 清华大学 计算机科学与技术系,北京 100084; 2. 山东政法学院 信息科学技术系,济南 250014)
(zhangp@tsinghua.org.cn)

摘 要:Native XML 数据库的快速查询,可以通过基于 XML 文档编码的结构连接算法实现。在对现有结构连接算法进行综述的前提下,提出一种新的 Native XML 数据库的结构连接算法——基于深度均匀划分的结构连接算法(DRIAM)。该算法不要求输入数据 AList 和 DList 有序或在其节点编码上建有索引,避免了排序和索引所增加的额外开销;不需要输入数据 AList 和 Dlist 全部加载到内存中,可以适应不同内存大小限制的情况,并且该算法时间复杂度非常低。

关键词:Native XML 数据库;XML 查询;结构连接
中图分类号: TP311.13 **文献标志码**:A

A new structural join algorithm in Native XML database

ZHANG Peng[1], FENG Jian-hua[1], FANG Zhi-feng[2]
(1. *Department of Computer Science and Technology, Tsinghua University, Beijing* 100084, *China*;
2. *Department of Information Science and Technology, Shandong Institute of Political Science and Law, Jinan Shandong* 250014, *China*)

Abstract: Structural join operation is the main solution to Native XML database query. Based on the survey of existing structural join algorithms, a new structural join algorithm, depth partition based structural join algorithm (DRIAM) was proposed. In DRIAM, input data sets AList and DList were not necessary to be ordered or indexed so that the additional cost was avoided. AList and DList were not necessary to be loaded in the memory. Besides, the time-complexity of DRIAM was very low.

Key words: Native XML database; XML query; structural join

0 引言

扩展标记语言(eXtensible Markup Language,XML)的重点是管理信息的数据本身,数据的显示交给其他技术解决。随着近些年 Web Service 的蓬勃发展,XML 越来越多地活跃在数据交换和存储领域。

Native XML 数据库是以自然的方式处理 XML 数据,以 XML 文档作为基本的逻辑存储单位,针对 XML 的数据存储和查询特点专门设计适用的数据模型和处理方法[1]。由于查询是数据库最为频繁的操作,因此在 Native XML 数据库的研究中,XML 查询处理器是研究的热点。XML 查询处理器的主要功能在于查询分解、查询优化和查询执行等,主要目的在于查询求解。目前 Native XML 数据库的查询求解方法包括:基于 XML 文档索引或者结构索引的导航遍历算法、基于 XML 文档编码的结构连接算法、基于 XML 文档序列表示的序列匹配算法等。在上述方法中,基于 XML 文档编码的结构连接算法是主流技术之一。Native XML 数据库的结构连接算法是实现 Native XML 数据库高效查询的重要条件,对 Native XML 数据库结构连接算法的研究具有非常重要的意义。

1 相关研究

为了能高效地查询求解,已经提出多种 Native XML 数据库的结构连接算法。所谓结构连接算法,是指给定一个潜在的 XML 祖先节点的集合 *AList* 和一个潜在的 XML 后裔节点的集合 *Dlist* 的情况下,找到所有的节点对(A,D)的集合的算法,其中节点 $A \in Alist$,节点 $D \in Dlist$,且节点 A 在 XML 文档中是节点 D 的祖先。根据基于结构连接算法是否要求 *Alist* 和 *Dlist* 有序或者有索引,可以将 Native XML 数据库的结构连接算法分为有序或有索引 XML 文档的结构连接算法和无序且无索引 XML 文档的结构连接算法。

1.1 有序或有索引 XML 文档的结构连接算法

大部分结构连接算法都要求输入数据 *AList* 和 *DList* 有序或在其节点编码上建有索引,其目的是为了跳过那些不存在祖先 - 后裔关系的节点。但排序和索引增加了算法处理的额外开销,尤其当 *AList* 和 *DList* 中的节点很多,以至于不能一次加载到内存时,已有的结构连接算法需要较高的 I/O 代价。

文献[2]提出了一种多谓词归并连接(Multi-Predicate Merge-Join,MPMJ)算法,其主要思想在于使用所有的连接条件指导归并,它的主要缺点在于多次扫描数据集。针对该算法存在的缺陷,文献[3]提出一种 Stack-Tree-Desc 算法,其主要思想是基于合并连接的算法,在内存中设置祖先节点栈,从而保证只扫描一次输入的数据集。文献[4]提出了一种 Anc_Desc_B + 算法,该算法利用索引技术全面改善连接的性能,在元素的编码上建立特定的索引结构,从而跳过不可能发生连接的节点。文献[5]针对上述方法不能有效跳过祖先节点的缺陷,提出一种 XR-Tree 算法,该算法基于新的索引结构,跳过不可能发生连接的祖先节点。XR-Tree 算法是目前公认比较好的结构连接算法之一。

1.2 无序且无索引 XML 文档的结构连接算法

XR-Tree 算法在输入数据有序或者在其节点编码上有索引的情况下,具有较好的性能。但是,在 Native XML 数据库中,结构连接算法的输入数据可能来自元素、路径或者中间结果,因此数据有序或者存在索引的前提并不是总能够得到满足。当数据有序或者存在索引的条件不成立时,这些算法需要

收稿日期:2007 - 09 - 03;**修回日期**:2007 - 12 - 12。
基金项目:浙江省自然科学基金资助项目(Y105230);清华大学基础研究基金资助项目(JCqn2005022)。
作者简介:张鹏(1981 -),男,山东莘县人,硕士研究生,主要研究方向:XML 数据库、移动 Ad Hoc 网络、移动 mesh 网络、移动通信; 冯建华(1967 -),男,山西万荣人,副教授,博士,主要研究方向:数据库、信息处理; 房志峰(1977 -),男,山东聊城人,讲师,硕士,主要研究方向:网络数据库。

对数据临时排序或者建立索引,降低了结构连接算法的效率。因此,有学者提出无序且无索引XML文档的结构连接算法。

文献[6]提出一种基于区域划分的结构连接算法,该算法利用区域编码的特点对输入数据进行划分,使得在输入数据无序并且无索引的情况下能够进行有效的结构连接操作。但是,该算法基于区间对输入数据进行划分,从而导致数据划分结果不均匀,影响效率。文献[7]提出基于PBiTree编码的结构连接算法,该算法使用节点的水平划分和垂直划分进行节点子集划分,其中水平划分根据节点在文档树的高度进行,垂直划分根据节点所在的分支进行。该方法仅仅适用于PBiTree编码,对于区间编码的文档不适用。文献[8]提出一种基于均匀划分策略的结构连接算法PRIAM,该算法基于BBT编码机制利用一定规则将输入数据均匀地划分到不同的桶中,以满足内存大小限制的需要,从而高效地实现了结构连接操作。但是,该算法的时间复杂度较高,至少为$O((|AList|+|DList|)\times\log 3N)$。

综合上述分析,现有的结构连接算法多为有序或有索引XML文档的结构连接算法,要求输入数据*AList*和*DList*有序或在其节点编码上建有索引,实用性和有效性较差。

2 基于深度均匀划分的结构连接算法

鉴于现有Native XML数据库结构连接算法存在的上述缺陷,本文提出一种基于深度均匀划分的结构连接算法(DRIAM),该机制具有较低的时间复杂度,效率较高。

2.1 XML文档节点的存储结构

为了实现本文所提出的结构连接算法,XML文档节点的存储采用下述存储结构实现。

2.1.1 存储实现

Native XML数据库中的XML节点的存储包括下述域:祖先域parents,数据域data,深度域depth和孩子域child。

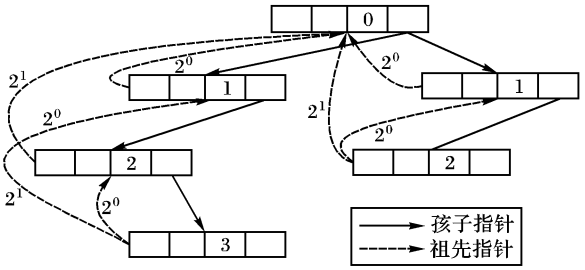

图1 存储结构示例

其中,祖先域parents是祖先指针数组$p[0]$,$p[1]$,…,$p[i]$,…,$p[n-1]$。其中,$p[i]$表示这个节点的第$2i$代祖先节点的地址,n是一个相当小的数。如果n为1010,那么p取10即可。如果整个XML树有m个节点,那么树的高度约为$\lg m$的级别,从而n的大小大约是$\lg m$。数据域data用于存放该XML节点存储的数据,如果该域变长,可以采用指针的形式。深度域depth用于标识该节点在XML文档树中的深度。孩子域child是孩子指针数组$c[0]$,$c[1]$,$c[2]$,…,用于存放孩子的地址。孩子的数目不确定,因此如果只用一个块无法存储孩子域child,则使用扩展块。图1给出了存储结构的一个示例,其中实线表示孩子指针,用于标识该节点的孩子;虚线表示祖先指针,用于标识该节点的第$2i$代祖先节点。

2.1.2 更新操作

和传统的关系数据库的更新操作相同,XML文档树节点的更新操作主要包括修改、删除和插入。

对XML文档树节点的修改主要是指,在不改变文档树结构的前提下,修改文档树中节点的内容、标签名称等。因为上述修改并不影响文档树的结构,因此也不会影响节点的存储结构。对XML文档树节点的删除需要递归地删除整棵子树。对XML文档树节点的插入采用下述操作实现:假设将节点newNode插入到父节点parents的第n个孩子前面,将节点parents当前第n个以及以后的孩子节点的指针向后移动,如果第n个孩子之后还有扩展块,则该扩展块也要做相应的后移,即将每个扩展块的最后一条记录都将移到下一个扩展块的开始位置。根据newNode.parents[i] = newNode.parents[$i-1$].parents[$i-1$]这条性质,依次计算newNode.parents[0],newNode.parents[1],…,newNode.parents[$n-1$],从而完成XML文档树节点的插入。

2.2 DRIAM算法

针对现有结构连接算法存在的缺陷,本文提出一种基于深度均匀划分的结构连接算法。该算法具有以下优点:1)该算法不要求输入数据*AList*和*DList*有序或在其节点编码上建有索引,避免了排序和索引所增加的额外开销;2)该算法不需要输入数据*AList*和*DList*全部加载到内存中,可以适应不同内存大小限制的情况;3)该算法时间复杂度非常低,I/O访问次数较少。

在进行结构连接操作之前,该结构连接算法基于*AList*分桶策略,将*AList*中的元素按深度(depth域)均匀地划分到不同的存储桶里,然后再对*DList*中的所有元素进行高效判断。上述划分需要保证只有存储桶$AList_i$与对应的存储桶$DList_i$之间的结构连接才会对最终的结构连接结果有所贡献,存储桶$AList_i$与存储桶$Dlist_j(i\neq j)$之间的结构连接不会对最终的结构连接结果产生任何影响。因此,数据划分需要满足式(1)和式(2):

$$Alist\infty Dlist=\bigcup_{i=1}^{n_b}Alist_i\infty Dlist_i \tag{1}$$

$$Alist_i\cap Dlist_j=\varnothing;i\neq j \tag{2}$$

为了使基于深度均匀划分的结构连接算法DRIAM得以实现,对输入数据的划分必须满足下述两个条件,其中n_b表示存储桶的数目:

1) $Alist=\bigcup_{i=1}^{n_b}Alist_i$,而且$Alist_i\cap Dlist_j=\varnothing, i\neq j$。

2) $\forall M,N\in Alist_i, M.depth=N.depth$。

该划分方法保证了式(1)(2)的成立。对于*AList*的划分方法,这里用b_s表示内存中最多可以容纳的*AList*的节点个数。假设*AList*中深度为i的节点有C_i个,把*AList*中深度为i的节点放入$\lceil C_i/b_s\rceil$个存储桶中,这样总的存储桶数目$n_s\leqslant\lceil|AList|/b_s\rceil+\log N$。划分算法如下:

算法1 *AList*的划分算法

计算C_i;

计算$U_i=\lceil C_i/b_s\rceil$,表示深度为i的节点需要的桶的数目;

计算$S_i=\sum_{k=1}^{i}U_k$,用于选择桶的编号;

$n_b=S_{MaxD}$,其中*MaxD*表示*AList*中节点的最大深度;

for i = 1 to |*AList*|

{

$d=AList[i].depth$;

把$AList[i]$放入第S_d个桶;

如果第S_d个桶已满,即达到b_s的节点,则$S_d=S_d-1$;

}

对于一个节点M,可以采用下述算法求节点M的第D层祖先。

算法2 求祖先的算法

ancesor(*node*,*n*)

输入:节点*node*,层次*n*

输出:节点*node*的第*n*层祖先

```
{
  P ← node;
  for(i = 0;i < lb n;i ++)
    if (n 的二进制第 i 位为 1)
      P ← P. F(i);
  Return P;
}
```

上述求祖先的算法 ancesor(*node*,*n*) 的时间复杂度为 $O(\text{lb } n) = O(\text{lb}(\text{lb } N))$,其中 N 为节点个数。

采用下述算法实现祖先后裔关系的结构连接操作。

算法 3　结构连接算法

SJR Join(*AList*, *DList*, *limit*, n_b)

输入:祖先节点集 *AList*,后裔节点集 *DList*,内存最多容纳的节点个数 *limit*,存储桶的数目 n_b

输出:$SJR = \{(A,D) \mid A \in AList, D \in DList, \text{且 } A \text{ 是 } D \text{ 的祖先}\}$

```
{
  根据算法 1 划分 AList;
  SJR = ∅;
  for(i = 0;i < n_b;i ++)
    {
    对桶 i 中的节点建立索引;
    D = 桶 i 中的节点的深度;
    for( j = 0; j <| DList |; j ++)
      {
        if (DList[j]. depth ≥ D)
          M = ancesor(DList[j],DList[j]. depth - d);
        if (在索引中找到 M)
          SJR ← SJR ∩ (M,DList[j]);
      }
    }
  return SJR;
}
```

图 2 是执行结构连接操作之前的 XML 文档树,其中实线框中的节点是 *AList* 所包含的节点,虚线框中的节点是 *DList* 所包含的节点。

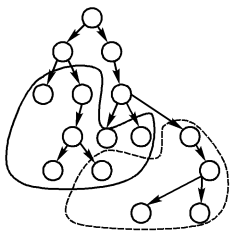

图 2　结构连接操作之前 XML 文档树

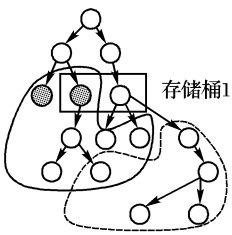

图 3　存储桶 1 的结构连接

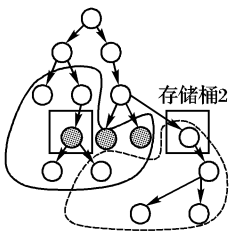

图 4　存储桶 2 的结构连接

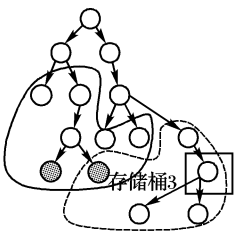

图 5　存储桶 3 的结构连接

如图 3 所示,根据算法 1,首先对于上述节点进行分层,并且得到存储桶 1。该存储桶 1 存储 XML 文档树中深度为 2 的节点。根据算法 3,在存储桶 1 中的 *AList* 的节点都用格状图案加以标识,并且被称为节点集 *AList′*。在存储桶 1 中的 *DList* 的节点的祖先都用框加以标识,并且被称为节点集 *AList″*。节点集 *AList′* 和 *AList″* 的交集就是符合结构连接条件的祖先,符合结构连接条件的后代可以在求符合结构连接条件的祖先的同时求出。从图 3 可以得知,有 1 个符合结构连接条件的祖先,它有 1 个符合结构连接条件的后代。

同理,如图 4 所示,根据算法 1 和算法 3,得到存储 XML 文档树中深度为 3 的节点的存储桶 2,从而得到 1 个符合结构连接条件的祖先,它有 1 个符合结构连接条件的后代。

最后,同理,如图 5 所示,根据算法 1 和算法 3,得到存储 XML 文档树中深度为 4 的节点的存储桶 3,从而得出没有符合结构连接条件的祖先。

完成上述操作之后,得出了所有结构连接的"祖先-后代"对。

2.3　算法分析

本文所提出的基于深度均匀划分的结构连接算法不需要输入数据有序或者建有索引结构,下面对该连接算法的时间复杂度和 I/O 访问代价进行理论分析。

2.3.1　时间复杂度分析

在基于深度均匀划分的结构连接算法中,根据对算法 3 的分析可以得知,由于内循环的执行次数约为 $\log N$,并且循环的时间复杂度为 $O(|AList| + |DList| \times (\log\log N + P))$,算法的时间复杂度约为 $O(|AList| \log N + |DList| \times (\log\log N + P) \times \log N)$。与现有的结构连接算法相比,该算法的时间复杂度最低。

2.3.2　I/O 访问次数分析

根据对算法 3 的分析可以得知,因为算法 3 第 1 行在读一遍 *Alist* 的同时还要把划分结果写到磁盘上,所以算法 3 第 1 行的 I/O 访问次数是 $2|Alist|$。在算法 3 的循环中,每次循环需要的 I/O 访问次数为 $|DList|$。在算法 3 将结构连接的最终结果写回磁盘的操作中,I/O 访问次数为 $|SJR|$。综合上述分析可以得知 DRIAM 算法的 I/O 访问次数约为:$2|Alist| + (\lceil |Alist| / b_s \rceil + \log N) \times |Dlist| + |SJR|$。与现有的结构连接算法相比,该算法的 I/O 访问次数比文献[8]提出的 PRIAM 算法略高,但是远低于其他结构连接算法。

3　性能分析

表 1　测试数据的大小

测试数据	$\lvert AList \rvert$	$\lvert DList \rvert$
XM1	9 750	35
XM2	21 750	43 500
XM3	21 750	48 250
XM4	25 500	12 823
XM5	10 830	59 486

本部分主要采用标准的 XMark 数据[9] 对 DRIAM 算法进行性能评估,与 PRIAM 算法,XR-Tree 算法的性能进行比较。执行实验程序的计算机的 CPU 是 Intel Core™ 2,内存为 512 MB,运行的操作系统是 Windows XP Professional,采用标准 C ++ 程序设计语言编写实验程序。

3.1　时间性能比较

在本实验中,生成标准测试数据集 XMark 的比例因子为 1,文件大小为 113 MB。实验选取了 5 组 XMark 数据(分别命名为 XM1,XM2,…,XM5)。测试数据的大小参见表 1。

从图 6 中可以看出,DRIAM 算法的时间性能较好,能够高速完成结构连接操作。并且输入数据非常大时,上述优势更为明显。

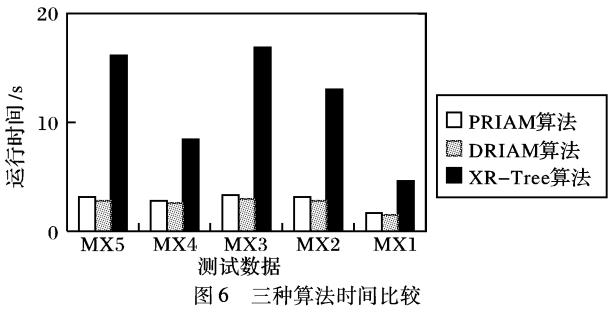

图 6　三种算法时间比较

(下转第 880 页)

$$Te(1) = \sum_{i=0}^{20} (T1(i) + T2(i)) = 3807\ ms$$

$$Te(5) = \sum_{i=0}^{20} max(T1(i), T2(i)) = 2039\ ms$$

所以，$\frac{Te(5)}{Te(1)} = \frac{2039}{3807} \approx 53.56\%$，接近于式(6)中50%的最理想状态。

测试分析：由表1中的实验测试结果可知，接收第1～9行时，Thread1比Thread3所需时间多，于是根据调度算法分配了更多的时间片，并且在Thread1和Thread3的接收和应用模式不发生大变化的情况下，*M*1和*M*3能够保持一个动态的相对稳定性；而在接收第10行开始，为了模拟负载的不确定性变化，人为地提高了数据处理的复杂度，于是Thread3需要获得更多的时间片，根据调度算法，*M*1和*M*3也动态发生变化，并逐渐稳定；同样地，在接收第18行数据开始，由于人为降低了数据处理的复杂度，于是*M*1和*M*3的分配再次恢复到给Thread1多分配时间片的状态。

由实验数据及其分析可以看出，在采用了主动等待多线程大数据传递策略后，能够通过调度保证负载不确定性下的性能。

4 结语

由于数据驱动中间层的出现，缓存中大数据传递的时间性能显得日益重要，但现存的策略无法解决负载不确定性上性能的保证。在此前提下，本文提出了主动等待多线程的大数据传递策略，基于负载不确定性的考虑，在此基础上，提出了通过动态反馈控制多线程时间片分配的方案。实验数据标明：该方案能够实现在负载不确定性下的性能稳定性，实验结果数据接近于算法分析的理想状态。

参考文献：

[1] O'NEIL P, O'NEIL E. Database-Principles, Programming, and Performance[M]. USA: Morgan Kaufmann, 2002.

[2] RIORDAN R M. Designing effective database system[M]. USA: Prentice Hall, 2006.

[3] SHASHA D, BONNET P. Database Tuning: Principles, Experiments, and Troubleshooting Techniques[M]. USA: Morgan Kaufmann, 2004.

[4] AKHTER S, ROBERTS J. Multi-Core Programming: Increasing performance through software multi-threading[M]. USA: Intel Press, 2007.

[5] CULLER D E. Parallel Computer Architecture[M]. USA: Morgan Kaufmann, 2005.

[6] BIC L, GAO G R, GAUDIOT J-L. Advanced Topics in Dataflow Computing and Multithreading[M]. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995.

[7] LU CHENYANG. Feedback control real-time scheduling: Framework, Modeling, and algorithms[M]. USA: University of Virginia, 2002.

[8] SKOGESTAD S, POSTLETHWAITE I. Multivariable Feedback Control: Analysis and Design[M]. New York, USA: John Wiley & Sons, 2001.

(上接第871页)

3.2 内存限制影响的比较

在本实验中，|*AList*|和|*DList*|的大小分别为21750和69969。图7描述了内存大小/数据集大小的值与结构连接算法的运行时间之间的关系，从而体现出DRIAM算法和XR-Tree算法在内存受限情况下的性能。

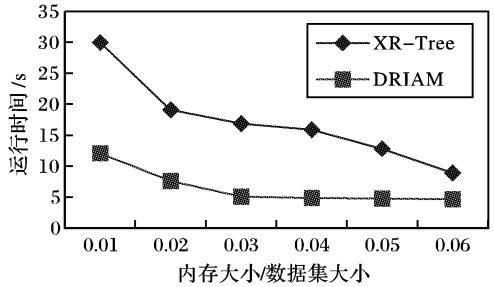

图7 内存限制对DRIAM算法和XR-Tree算法的影响比较

从图7可以看出，当内存大小受到限制时，XR-Tree算法的性能急剧下降；而DRIAM算法相对而言保持良好的性能。因此，DRIAM算法可以高效地应用在不同内存限制的环境中。

4 结语

本文集中讨论了Native XML数据库的结构连接算法，提出了一种新的结构连接算法：基于深度均匀划分的结构连接算法DRIAM。理论分析和实验结果表明，该算法不要求输入数据*AList*和*DList*有序或在其节点编码上建有索引，避免了排序和索引所增加的额外开销；该算法不需要输入数据*AList*和*DList*全部加载到内存中，可以适应不同内存大小限制的情况，并且该算法时间复杂度非常低。这是在Native XML数据库结构连接算法的研究上进行的有效尝试。

本文提出的结构连接算法I/O访问次数较多，如何降低由此带来的代价，如何降低偏斜数据对该算法的影响，等等，都是值得进一步研究的问题。

参考文献：

[1] 冯建华，钱乾，廖雨果，等. 纯XML数据库研究综述[J]. 计算机应用研究，2006，23(6)：1－7.

[2] ZHANG CHUN, NAAGHTON J, DeWITT D, *et al*. On supporting containment queries in relational database management systems [C]// ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2001: 426－437.

[3] AL-KHALIFA S, JAGADISH H V, KOUDAS N, *et al*. Structural joins: a primitive for efficient XML query pattern matching[C]// 18th International Conference on Data Engineering. San Jose: IEEE Computer Society, 2002: 141－152.

[4] CHIEN S Y, VAGENA Z, ZHANG DONG-HUI, *et al*. Efficient structural joins on indexed XML documents[C]// 28th International Conference on Very Large Data Bases. Hong Kong: Morgan Kaufmann Publishers, 2002: 263－274.

[5] JIANG HAI-FENG, LU HONG-JUN, WANG WEI, *et al*. XR-Tree: indexing XML data for efficient structural joins[C]// 19th International Conference on Data Engineering. Los Alamitos: IEEE Press, 2003: 253－264.

[6] 王静，孟小峰，王珊. 基于区域划分的XML结构连接[J]. 软件学报，2004，15(5)：720－729.

[7] WIRTH N. Type extentions[J]. ACM Transaction on Programming Languages and systems, 1988, 10(2): 204－214.

[8] 冯建华. 纯XML数据库的查询求解关键问题研究[D]. 北京：清华大学，2006.

[9] Xmark: An XML Benchmark Project[EB/OL]. [2007－08－01]. http://monetdb.cwi.nl/xml.