维普资讯 http://www.cqvip.com

计算机科学 2005Vol. 32№. 11

DVBB:基于 Dewey 向量的 B^+ 树索引结构连接算法*)

张恩德　王国仁　宁　博　王　斌
(东北大学信息科学与工程学院计算机系统研究所　沈阳 110004)

摘　要　为解决 XML 数据库中的结构关系查询问题,本文以 Dewey 向量为基础,提出了基于 Dewey 向量的 B^+ 树(Dewey Vector Based B^+ Tree,简称为 DVBB)双栈结构连接算法。该算法利用了两个栈,PublicStack 和 PrivateStack,在这两个栈的基础上,利用 DVBB 索引,能够最大限度地避免那些不能产生连接结果的元素参加连接运算。一系列的实验结果表明,基于 DVBB 的双栈结构连接算法,无论是对于有效的跳过"祖先"还是"后代"节点,都具有很高的性能。

关键词　XML,结构连接,Dewey 向量编码,B^+ 树索引

DVBB: A Structural Joins Algorithm on Dewey Vector Based B^+ Tree Index

ZHANG En-De　WANG Guo-Ren　NING Bo　WANG Bin
(College of Information Science & Engineering, Northeastern University, Shenyang 110004)

Abstract　To evaluate the primitive structural relationships of XML data, a new algorithm named DVBB (Dewey Vector Based B^+ tree)is proposed in this paper. Unlike the traditional way, DVBB employs the Dewey Vector to encode XML elements, and performs structural joins on the DVBB index. This algorithm can effectively skip the elements which cannot produce the join results with two stacks, PublicStack and PrivateStack. An extensive of experiments show that the DVBB algorithm can gain the best efficiency in structural joins.

Keywords　XML, Structural joins, Dewey code, B^+ tree index

1　引言

Internet 的迅猛发展,越来越多的数据采用 XML[1] 作为数据表示和数据交换的标准。有效地对 XML 数据进行存储和查询日益受到人们的关注。作为 XML 查询语言的一种标准,XPath 以及 XQuery 渐渐地成为了标准。这些查询语言都将路径表达式作为核心内容,而其中对于"/"(表示"父子"关系)和"//"(表示"祖先后代"关系)的查询操作,即结构连接操作,又是核心中的核心。

解决结构连接一种直接而简单的办法就是遍历 XML 文档树,匹配路径查询表达式。但这种方法在处理大规模的 XML 文档数据的时候,无论是在空间性能还是时间性能上,都不尽如人意。因此,在 XML 范围编码(DocId, StartPos, EndPos, Level)[2] 的基础上,提出了很多结构连接算法[2~5],很好地解决了结构连接问题。其中 MPMGJN[2] 和 Stack-Tree Desc/Anc[3] 是各种算法的基础,文[4]提出的基于 B^+ Tree 的 Anc_Des_B^+ 算法,有效地提高了连接效率,特别是随着 XR-Tree[5] 被引入,提出的基于 XR-Tree 的结构连接算法 XR-stack,使结构连接的性能有了很大的提高。被认为是当前解决结构连接最先进的(state-of-the-art)办法。

上文所述结构连接算法,均是以 XML 范围编码为基础。该编码对 XML 文档进行更新的代价特别昂贵,很多时候,为了只插入一个节点元素,不得不对整个 XML 文档的编码进行更新。在对 XML 文档的编码中,另一种编码技术 Dewey[6] 向量,能够很好地解决这个问题。在文[6,7]中,提出了基于 Dewey 向量的 XML 编码更新解决办法。通过进一步的研究我们发现,有效地利用 Dewey 向量的性质,我们建立基于 Dewey 向量的 B^+ 树索引,利用双栈模型,可以最大限度地减少参加连接运算的祖先和后代元素个数,尤其对于减少参加连接运算的祖先元素,不需任何辅助索引,因此大大提高了连接效率。

2　背景知识和相关工作

为了高效地确定"父子"和"祖先后代"关系,以 Stack-Tree[3] 为基础的一组结构连接算法利用了范围编码方法[2]。这里,我们先介绍一下对 XML 节点元素进行编码的范围编码的编码技术,然后介绍我们所采用的 Dewey 向量编码技术,最后讨论了结构连接的相关问题。

2.1　XML 编码

XML 数据对象通常被形象地表示为 DOM 树,树中的边表现了 XML 数据之间的嵌套关系,树中的节点表示元素、属性及值。为了表现 XML 数据中的各种关系,需要对 XML 文档的元素进行编码,文[2]提出的(DocId, StartPos, EndPos, Level)范围编码技术,文[6]的 Dewey 向量编码,都是为了这个目的。

2.1.1　*范围编码*　图 1 给出了一个 XML 范围编码的例子。在范围编码中,DocId 代表的是不同的 XML 文档的文档编号,对于多个文档和对单个文档的结构连接问题来说,它们本质上是一样的,因此图中省略了 DocId。StartPos 表示先序遍历 XML 文档中某节点的编号,EndPos 表示在先序遍历文档中回溯到该节点的编号,对于叶子节点来说,它们的 StartPos 和 EndPos 是一样的。Level 用来表示该节点所在

*)基金项目:教育部高等学校优秀青年教师教学科研奖励计划基金资助项目;国家自然科学基金(60473074,60273079)资助。

维普资讯 http://www.cqvip.com

XML 树状模型中所在的层数，它用来帮助判断某两个节点之间是否具有“父子”关系。

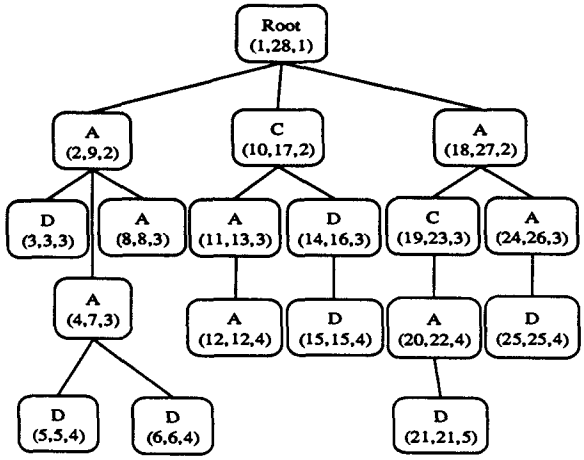


图 1　XML 范围编码实例

由于 XML 文档的严格嵌套关系，因此，各个元素节点的 StartPos 和 Endpos 之间不会有交叠的情况产生，也就说，如果 a 元素是 d 元素的祖先，那么必然是 a. StartPos < d. StartPos 并且 a. EndPos > d. EndPos，反之亦然，它们之间是充分必要条件。更进一步，如果 a 元素是 d 元素的父亲，那么它们的编码一定满足 a. StartPos < d. StartPos 且 a. EndPos > d. EndPos，并且 d. Level = a. Level + 1。

2.1.2　Dewey 向量编码　Dewey 编码[4]（Dewey Decimal Classification，点分十进制编码）的提出已有超过一百年的历史，原来主要用在图书馆分类系统上。本文称 Dewey 编码为 Dewey 向量，并把每个被小数点分隔开的部分称为向量的一个元素。注意，在本文中，多次提到了“元素”这个概念，它在不同的地方有不同的含义，当我们说元素节点的时候，我们指 XML 文档中的某一个节点（node），但我们说 Dewey 向量中的元素时，是指在 Dewey 向量中，被小数点隔开的每个部分。

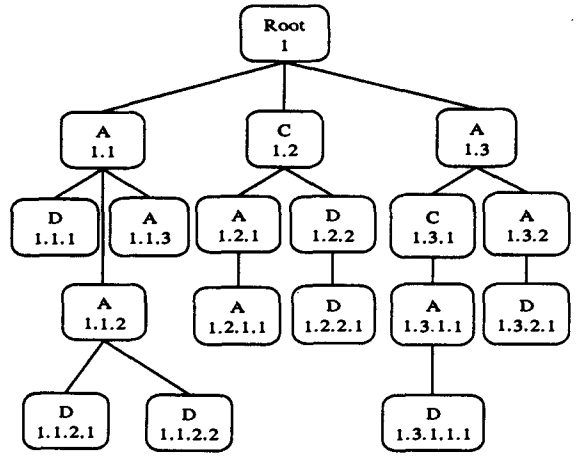


图 2　XML Dewey 编码实例

由于 XML 文档的树型模型，Dewey 向量所体现的层次思想能够很好地表现 XML 元素节点之间的各种关系，因此，用它对 XML 文档进行编码十分适合。XML 文档中每个节点被赋予一个向量 v，该向量的值唯一标识了一条从根到该节点的路径。在 Dewey 向量中，每个元素的值表示该向量在哪一条路径上（第几个孩子），向量元素的个数表示节点所在 XML 文档 DOM 树中的层数。图 2 给出了一个 Dewey 向量编码的实例。

表 1　Dewey 向量在计算机内部的存储实例

Bitstring	Li	Oi value range
0001	2	[−5,−2]
001	1	[−1,0]
01	0	1
10	1	[2,3]
110	2	[4,7]
1110	4	[8,23]
11110	8	[24,279]
111110	12	[280,4375]
1111110	16	[4376,69911]
11111110	20	[69912,1118487]

为判断任意两个节点是否具有“祖先后代”关系，只需要判断其中的一个编码是否是另一个编码的“前缀”即可；对于“父子”关系，只需要在“祖先后代”关系的基础上进一步判断它们之间向量元素的个数是否相差为一。Dewey 向量对于判断是否是“兄弟”等其它关系也十分方便，它几乎包括了 XML 文档元素关于位置的所有信息。

由于计算机对于任何数据的存储就是 0、1 比特串，因此，怎样把 Dewey 这个点分十进制存在计算机内部，是一个问题。文[6,7]提出了一个很好的解决办法，利用 UTF-8 的思想，每一个 Dewey 向量用前缀（Li）和值（Oi）来表示，如表 1[7]。按照表 1，1.5.3.11 在计算机内部实际 bit 串为 01(1)11001(5)101(3)11100011(11)。

2.2　结构连接

在引言中，我们已经介绍了结构连接的定义。结构连接的目的就是要找到在两个节点集合之间出现的全部结构关系，包括“祖先后代”关系和“父子”关系。在 XQuery 或者 XPath 中，具体的表现为“//”（祖先后代关系）和“/”（父子关系）。以“//”为例，如查找“a//d”，假设我们已经得到 a 和 d 节点的候选元素节点集合 A-List 和 D-List。要匹配这个查询，找到所有满足祖先后代关系的“a-d”对，就是计算祖先队列 A-List 和后代队列 D-List 之间的结构连接。文[3]提出的 Stack-Tree 算法是目前广为接受的结构连接算法。Stack-Tree 算法中的 A-List 和 D-List 是由 StartPos 属性排序的有序的输入队列。算法利用一个堆栈来管理 A-List，从而保证对 A-List 和 D-List 只遍历一次，提高了效率。但是它对 A-List 和 D-List 的所有元素都进行了比较，显然这里有一些元素不会产生连接结果，存在多余的访问操作。

在文[4]中，利用 B^+ 树索引技术，提出了一种有效地跳过（skip）不会产生连接结果的元素的算法。文[4]利用范围编码的特点，索引了每个元素节点编码中的 StartPos 编码，由于 XML 文档中元素节点的严格嵌套关系，可以保证当某一祖先元素不是当前所处理后代元素的祖先时，所有在 A-List（已按照 StartPos 排序）的元素中，位于当前祖先节点和当前后代节点之间的元素都不会与该后代节点产生连接结果，因此，可以跳过这些元素。同理，对于后代元素，经过 B^+ 树索引，也可以减少不必要的操作。

文[5]中，为了进一步地跳过祖先节点，提出了一种更为有效的索引结构，XR-Tree 索引。在文[4]的 B^+ 树算法中，该算法只能部分地跳祖先，对于非嵌套的（less nested）祖先元

维普资讯 http://www.cqvip.com

素，该算法无能为力。XR-Tree 增加了一个辅助索引 StabList，通过一些 Key 把各个元素“串”(stab)起来。XR-Tree 相当于同时索引了 StartPos 和 EndPos，因此它在 B^+ 树的基础上进一步提高了连接算法的性能。

3 DVBB：B^+ 树索引双栈结构连接算法

为了利用 DVBB 算法解决结构连接问题，我们建立了 DVBB 索引，其结构如图 3。

在 Dewey 向量的基础上，我们可以定义几个基本的原语(Primitive)操作，利用 DVBB 索引，很容易实现它们。这些基本操作不仅是我们结构连接算法的基础，在解决 XML 的其它问题中，也有很多应用，因此单独把它们列出，逐一加以说明。

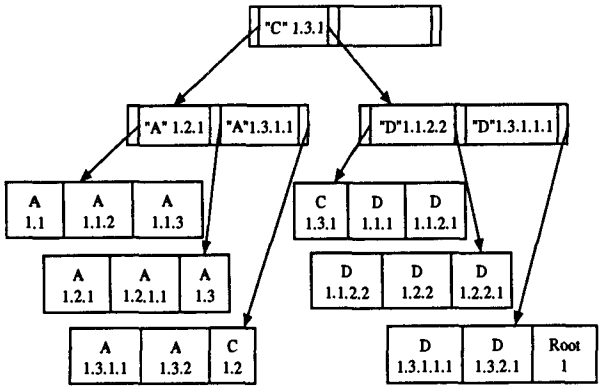


图 3 一个 DVBB 索引实例

(1)CmpDocOrder(E1, E2) 该原语操作用来比较任意两个节点的文档序。在结构连接以及很多的 XML 的应用当中，文档序是一个很重要的问题。在范围编码当中，由编码的 StartPos，很自然地就可以确定文档序，而在 Dewey 向量编码中，按照表 1 的存储方式，可以按字节来比较任意两个 Dewey 向量的大小关系，其中每个字节都符合字典序(lexicographic order)。在下文中，为叙述方便，我们把 E1、E2 的文档序分别记为 E1<E2，E1>E2 和 E1=E2。

(2)IsAncestor(Ea, Ed) 该原语操作返回一个布尔值，判断任意两个元素是否具有祖先后代关系。判断祖先后代关系，也是结构连接算法的一个基础，同上面一样，我们也是按字节来比较 Ea 在计算机内部存储的 Dewey 向量是否是 Ed 的 Dewey 向量前缀。实际上，也就是比较任意两个比特串是否具有前缀关系，而在计算机内部，对于这个问题，我们可以用位操作来实现。因此，对于判断祖先后代关系，效率是很高的。

(3)GetParent(E) 该原语操作得到某一节点的父亲节点。在 XML 中，给定某个节点，要得到其的父亲的 Dewey 向量，是十分容易的事情。我们只需要把该节点的 Dewey 向量的最后一个元素去掉，即得到其父亲节点的 Dewey 向量，然后，我们利用 DVBB 中值查询的特性，很容易得到该元素的父亲节点。在我们后面的结构连接算法中，为了实现有效的跳过(skip)没有连接结果的祖先元素节点，GetPrivateAncestors 就利用了这个原语操作。

3.1 DVBB 双栈结构连接算法

双栈结构连接算法如下所示。该算法利用了两个有序队列及两个栈来解决结构连接问题。两个队列为祖先队列(A-List)和后代队列(D-List)，分别存储了祖先元素和后代元素候选集，算法依次扫描两个队列；两个栈(PublicStack 和 PrivateStack)用来存储中间结果。其中 PublicStack 保证能够对两个队列只进行一遍扫描，因为在顺序扫描祖先队列时候，某些已处理过的祖先元素很可能是后面 D-List 中元素的祖先，为了保留这些元素，我们把它们存放到 PublicStack 中；PrivateStack 的用途是为了有效地跳过无用元素节点，因为在扫描两个队列的时候，有一些元素是不能够产生连接结果的，为了避免这样的元素节点参加连接运算，引入了 PrivateStack，这个栈用来存储祖先节点元素，对于不能够进入这个栈的那些元素节点，即表示不会产生连接结果，我们的算法利用 DVBB 索引，可以有效地跳过它们。

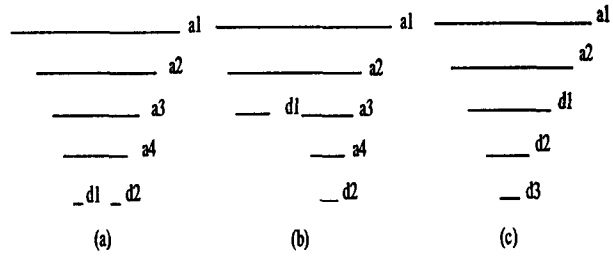


图 4 不同的祖先后代嵌套情况

以图 4 为例，我们可以具体说明这两个栈的作用。当算法处理到 d2 的时候，此时，d2 的祖先节点可能全部(如图 4a)或者部分(如图 4b)处理过，处理过的元素放到 PublicStack 中，表示它们以后还可能会和后面的后代节点进行匹配。在如图 4a，因为 d2 的祖先节点全部被处理过了，因此，此时的 PrivateStack 为空，表示 d2 没有自己的“私有”节点，在图 4b 中，d2 的祖先元素被部分地处理过，a1、a2 已经被压入到 PublicStack 中，我们只需要处理 d2 的“私有”祖先节点 a3 和 a4，把它们相应地压入到 PrivateStack 中，然后把它们按顺序压入到 PublicStack 中，它们也变成了“公共”节点。

算法：结构连接 Structural_Joins_DVBB(A-list, D-list)
输入：祖先元素队列 A-list，后代元素队列 D-list；
输出：连接运算结果；
算法描述：

```
1: Initialization();
2: While (A-list 和 D-list 没有结束)do
3:     PopStack(PublicStack, Topop, CurrentD );
4:     GetLastProcessedA(PublicSatck);
5:     If (CurrentA < CurrentD)
6:         PrivateStack ← GetPrivateAncestors (CurrentD, AncTag,
                          LastProcessedA);
7:       PrivateStack 中的元素压到 PublicStack 中，清空 Privat-
                      eStack 清空;
8:       输出连接结果(PublicStack 中元素 A,CurrentD);
9:       CurrentA ←A-List 中满足 CurrentA>CurrentD 的第一个
                    元素;
10:      如果 CurrentD 是 CurrentD. next()的祖先，ToPop 置为假，
                    否则 ToPop 为真;
11:      CurrentD ← CurrentD. next();
12:    Else If PublicStack 栈空
13:        CurrentD←D-List 中满足 CurrentD >CurrentA 的第一
                    个元素;
14:        Else 输出连接结果(PublicStack 中元素 A,CurrentD);
15:        CurrentD← CurrenD. next();
16:      End If
17:  End If
18: End While
Function PopStack(PublicStack, ToPop, CurrentD )
1: If ToPop 为真
2:    PublicStack 中所有非 CurrentD 祖先的元素弹出;
3:    把 ToPop 置为假;
Function Initialization()
1: PublicSatck 和 PrivateStack 分别置空;
2:    CurrentA 和 CurrentD 指向 A-List 和 D-List 首元素;
3: ToPop 置为假;
4: End If
Function SetLastProcessedA(PublicSatck)
```

维普资讯 http://www.cqvip.com

```
1:If PublicSatck 为空
2:    LastProcessedA ←文档的根元素节点;
3:Else LastProcessedA ← PublicSatck. top();
4:End If
Function GetPrivateAncestors (CurrentD,AncTag,LastProcessedA)
1:ParentElement ← CurrentD. GetParent();
2:While(ParentElement > LastProcessedA)
3:    If (ParentElement. tagName == AncTag)
4:        把 ParentElement 压到 PrivateStack 中;
5:    End If
6:        ParentElement ← ParentElement. GetParent();
7:End While
```

算法主要执行过程如下,首先是初始化,将两个栈清空,将待处理的祖先和后代节点分别指向两个队列的第一个元素。当两个队列没有处理完的时候,执行循环部分。为了有效控制压入 PrivateStack 中的元素,我们需要得到当前 PublicStack 中的栈顶元素,得到最后一个处理的祖先节点元素 LastProcessedA,由于 XML 文档的严格嵌套关系,我们可以保证在 LastProcessedA 前面的元素都被处理过。如果当前待处理的祖先节点(CurrentA)小于后代节点(CurrentD),此时有可能会跳过祖先元素,在算法的第 9 行,根据我们的 DVBB 索引,跳过不会产生连接结果的祖先节点元素。如果当前待处理的祖先节点大于后代节点,可能会跳过后代节点元素。注意,在跳过后代元素之前,需要检查 PublicStack 是否为空,当 PublicStack 为空的时候,可以利用 DVBB 索引跳过不必要的后代元素,否则,把待处理的后代节点指向 D-List 中的下一个元素。

在算法中,引入了一个布尔变量 ToPop,该变量是为了处理如图 4c 的情况。我们考虑当后代元素存在嵌套时,如图 4c,因为每次循环开始都要把不是当前后代节点的祖先的 PublicStack 中元素弹出,如果没有 ToPop,处理 d1、d2、d3 的时候,都要进行入栈和弹栈的操作,势必大幅度地降低算法性能,实际上 XR-Tree 在处理后代节点嵌套的时候,就存在这个问题。我们引入布尔变量 ToPop 可以保证,如果后代节点有嵌套,不需要冗余的入栈和弹栈的操作,连接算法的性能有很大的提高。

3.2 匹配父子关系

对于结构连接中的父子关系,DVBB 算法更显示了其优越性。因为通过 Dewey 向量很容易找到某节点的父亲节点。我们只需要在原语操作(3)的基础上作简单的扩充,判断通过原语(3)找到的祖先节点是否等于 A-List 的 TagName,并把上述算法中的第 6 行和第 7 行改为

```
6:ParentElement ← GetParent(CurrentD);
7:    If (ParentElement. TagName == AList. Tag)把 ParentElement 压到栈 PublicStack 中;
```

由于匹配父子关系只涉及到两层,因此省略了双栈模型中的 PrivateStack 栈,只保留了 PublicStack。但我们仍然利用了结构连接算法,是因为如果有多余的节点元素不能产生连接结果的话,我们的算法可以有效地跳过它们,减少不必要的操作。

对于匹配父子关系的结构连接来说,现有的算法并没有很好地解决。因为基于范围编码的结构连接算法,并不能快速地找到父亲节点元素,因此现存的大部分结构连接算法,Stack-Tree、B^+ Tree Index 和 XR-Tree 等,对于处理父子关系,都是在判断祖先后代关系的基础上进一步对 Level 关系的判断。但是事实上,对于结构关系来说,对"/"关系的处理要比对"//"关系的处理简单很多,因为它只涉及到一层嵌套,不必保留祖先元素节点,我们的算法很好地解决了这个问题。

4 实验结果与性能测试

为了测试 DVBB 双栈结构连接算法的性能,我们做了一系列的实验,并根据它们不同的祖先选择率、后代选择率以及不同文档大小,对它的性能分别进行了测试。

所有的测试都是在一台 2.5GHz,512M 内存,80G 硬盘的 PC 机上进行的。测试时的操作系统为 Windows XP,使用 Berkeley DB 数据库管理系统来存储数据,使用 Apache's Xerces-C XML parser 对 XML 文档进行解析,算法用标准 C^{++} 语言实现,使用 Microsoft Visual C^{++} 6.0 编译器编译。

实验主要在 XR-Tree 索引结构连接算法(以下简称 XR-Tree)与基于 Dewey 向量 B^+ 树双栈结构连接算法(以下简称 DVBB)上进行。实验并没有与 Stack-Tree[3] 和 B^+ 树索引算法[4]这两种算法进行比较,因为在文[5]中,XR-Tree 已经与 Stack-Tree 和 B^+ 树算法进行过全面详细的比较,说明了 XR-Tree 在不同的祖先选择率和不同的后代选择率上,性能均优于这两种算法。

为公平,我们采用了和 XR-Tree 中完全一样的 DTD,其中用到的 A-List 和 D-List 中的 tagname,也与 XR-Tree 一致。

实验 1 不同的祖先选择率对性能的影响

图 5 显示了不同的祖先选择率对实验结果的影响。该实验主要为了比较在后代选择率一定的情况下,对于不同的祖先选择率,DVBB 与 XR-Tree 在性能上的表现,关于祖先或者后代选择率,我们是这样定义的:

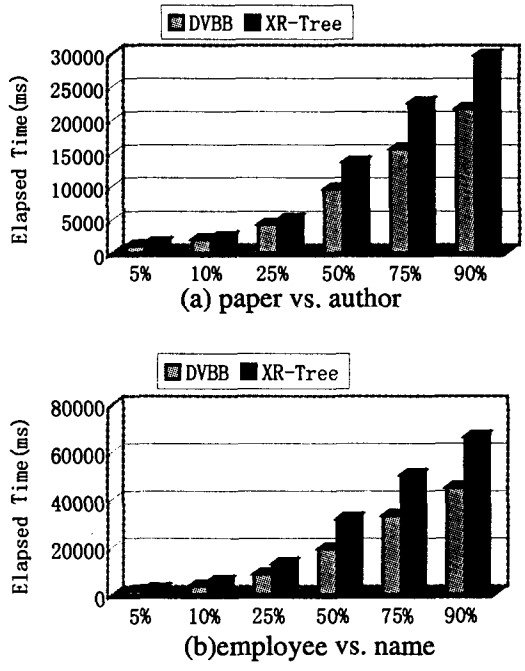


图 5 不同祖先选择率的实验结果

$$\text{祖先(后代)选择率}=\frac{\text{能够产生连接结果的祖先(后代)节点数}}{\text{祖先(后代)节点总个数}}$$

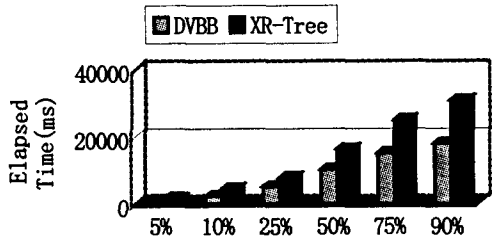
图 5 实验中保持后代的选择率很高(99%),也就是说,几乎大部分后代节点都能都产生连接结果,每次实验,控制能产生连接结果的祖先元素的个数。

实验 2 不同的后代选择率对性能的影响

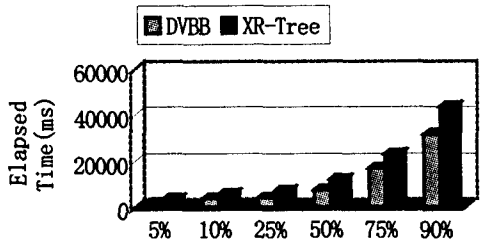
图 6 是我们在保持祖先选择率很高(99%),控制后代选择率变化的实验结果。在这次实验中,通过变化后代选择率,

维普资讯 http://www.cqvip.com

在不同的后代选择率的情况下，对算法性能进行了测试。



(a) paper vs. author

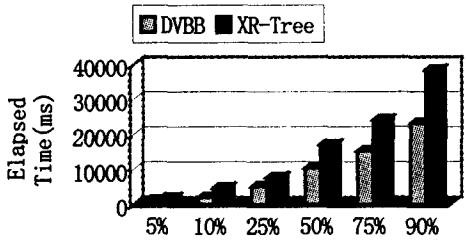


(b)employee vs. name

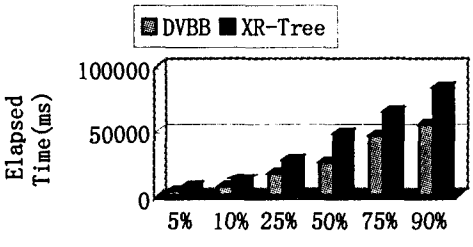
图 6　不同的后代选择率结果

实验 3　祖先后代选择率均变化对性能的影响

对于祖先后代的选择率均变化的时候，也进行一系列的测试。图 7 是我们这次实验的结果。我们对祖先和后代的选择率变化时均进行了测试，实验结果表明，在祖先后代的选择率均变化时，我们的算法也具有很好性能。



(a) paper vs. author



(b)employee vs. name

图 7　祖先后代选择率均变化的实验结果

实验 4　XML 文档大小变化对性能的影响

采用 Dewey 向量，随着 XML 文档的增大，某些节点的编码在计算机内部的存储空间会增大。为了检测文档大小对我们算法的影响，我们在一系列大小不同的文档上进行了实验。

实验所用数据由 IBM XML data generator 产生。DTD 与前面的多层嵌套一致，数据集的嵌套层数最多为 10 层，在文[7]中已经说过，99% 的 XML 文档，其嵌套层数不超过 8 层。因此，这样的嵌套深度足以验证我们算法的性能。实验结果表明，DVBB 算法在文档的大小变化时，也具有很好的性能。

表 2　不同大小文档的一些参数

parameter / Average size (byte)	Max length (bit)	Number of elements	Matched results (pair)
80M	5	71	1688315 6634279
120M	5	71	2260325 8984952
160M	5	71	3005168 11937419
200M	5	71	3615006 14492231
300M	5	71	5616712 22520229

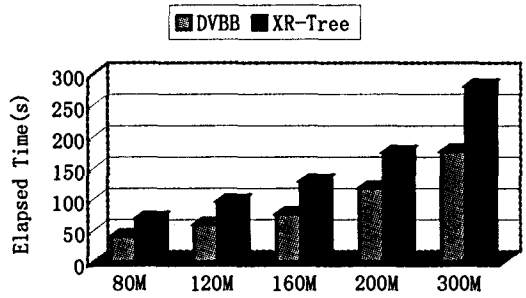


图 8　文档大小变化对性能的影响

结论　本文在总结前人工作成果的基础上，提出了一种基于 Dewey 向量的 B^+ 树索引结构，并在此基础上提出了基于该索引结构的连接算法——DVBB 双栈结构连接算法。并对算法进行了详细的讨论。最后针对不同的祖先或者后代选择率以及不同的文档大小进行了一系列实验，实验的结果表明该算法对于 XML 文档的结构连接操作，性能确实具有大幅提高。在以后的工作中，我们会对复杂的结构连接，Twig Join 进行进一步的研究，通过扩展 DVBB 算法，解决 Twig 查询问题。

参 考 文 献

1　Extensible Markup Language (XML)1.0 (Second Edition). October 2000. W3C Recommendation available at http://www.w3.org/TR/2000/REC-xml-20001006

2　Zhang Chun, Naughton J, DeWitt D, et al. On Supporting Containment Queries in Relational Database Management Systems. In: Proc. of the ACM SIGMOD Conference. New York, ACM Press, 2001. 425～436

3　Al Khalifa S, Jagadish H V, Koudas N, et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In: Proc. of ICDE, San Jose, California, USA, 2002. 141～152

4　Chien S Y, Vagena Z, Zhang D, et al. Efficient structural joins on indexed XML documents. In: Proc. of the 28th Int'l Conf. on VLDB. San Francisco, Morgan Kaufmann Publishers, 2002. 263～274

5　Jiang H, Lu H, Wang W, Ooi B C. XR-Tree: Indexing XML data for efficient structural joins. In: Proc. of the 19th ICDE. Los Alamitos, IEEE Press, 2003. 253～263

6　Tatarinov I, Viglas E, Beyer K, Shanmugasundaram J, Shekita E. Storing and Querying Ordered XML Using a Relational Database System. In: Proc. of the ACM SIGMOD Conference. Madison, Wisconsin, USA, 2002. 310～321

7　O'Neil P, O'Neil E, Pal S, et al. ORDPATHs: Inter-Friendly XML Node Labels. In: Proc. of the ACM SIGMOD Conf. Paris, France, 2004. 903～908