

# 基于分片的 XML 快速结构连接算法

王国仁 乔百友 韩东红 王 斌

(东北大学信息科学与工程学院 沈阳 110004)

**摘 要** 结构连接作为 XML 查询的重要部分,对查询性能来说起着非常重要的作用.目前有几种结构连接算法已经被提出,例如 Stack-Tree、XR-tree. 这些算法主要集中在节点之间关系的确定上.与之不同,作者从分片的角度去解决结构连接问题,首先把节点间的关系引申到分片之间的关系,从而得出各分片之间的一些性质,再利用分片间的性质来提高结构连接操作的性能.文中提出了一种基于分片的结构连接算法和两种优化方法,实验表明该算法在性能上要优于 Stack-Tree 算法和 XR-tree 算法.设计了一个简单而又高效的索引结构来存储分片结果,实验结果表明该索引结构的维护代价要小于 XR-tree 的维护代价.

**关键词** XML; 结构连接; 分片

**中图法分类号** TP311

## A Fast Structural Join Algorithm Based on Partitioning

WANG Guo-Ren QIAO Bai-You HAN Dong-Hong WANG Bin

(College of Information Science and Engineering, Northeastern University, Shenyang 110004)

**Abstract** Structural join is the core part of XML queries and has a significant impact on the performance of XML queries. Several structural join algorithms have been proposed such as Stack-Tree and XR-tree. This paper studies to solve the problem of structural join by partitioning. It first extends the relationships between nodes to the relationships between partitions and get some observations and properties about the relationships between partitions that can be exploited for improving the performance of the structural join algorithms. This paper then proposes a new partition-based structural join method and two optimized methods based on the properties derived from these observations. Extensive experiments show that the performance of the proposed algorithms is better than that of Stack-Tree and XR-tree algorithms. In order to store the partitioning results, the authors design a simple but efficient index structure. The experimental result shows that the index structure has less maintenance overhead than XR-tree.

**Keywords** XML; structural join; partitioning

## 1 引 言

随着 XML 作为 Internet 一个新的数据交换和表示的国际标准,面向 XML 的查询处理受到了越

来越多的关注.在处理 XML 查询时,一条查询往往被分解为一系列包含“/”或“//”的子查询,然后对所有的子查询的结果进行合并,最后得到最终的查询结果,在这里我们称“/”和“//”为基本的结构连接,基本的结构连接问题解决得如何将直接影响着

XML 查询性能的优劣,所以基本的结构连接问题一直是 XML 查询问题的核心部分。

在以前的文献中有许多结构连接算法被提出,文献[1]提出了基于合并的连接算法称为 MPMGJN (Multi-Predicate Merge Join); 文献[2]提出了 Stack-Tree-Desc/Anc 算法,该算法通过使用堆栈的机制极大地改进了基于合并的结构连接算法,它只需对要连接的两个元素链表进行一次遍历;文献[3]使用 B+ 树作索引来实现 Stack-Tree 算法,目的是能跳过一些不参加连接的后代元素;文献[4]提出了 XR-tree(XML Region tree)来过滤一些不参加连接的祖先后代元素。同时,前人也提出了许多编码方法用于快速定位树中节点之间的关系,以前解决结构连接问题主要是利用编码的基本性质,换句话说,他们大部分关注的是结点之间的关系,而忽略了把不同节点进行分组后,各组之间的位置关系,在本文中把“组”称为“片断”;文献[5]中提出了两种通过分析 XML 文档模式信息对路径表达式的优化策略:路径缩短策略和补路径策略,从而来提高 XML 路径查询效率。这就增加了一部分额外的工作量,而且对于优化后或不能优化的查询,该文献研究得不够。

在这里,为了提高 XML 查询效率,我们把节点间的关系引申到片断之间的关系,从而提出一种基于分片的结构连接方法。为了进一步改进该方法,我们又提出一种基于分片的高效结构连接算法和两种优化算法。

本文第 2 节介绍本文所选择的编码模式和一些有关分片的概念;第 3 节提出一种新的基于分片的结构化连接算法;第 4 节提出一种高效的基于分片的结构化连接算法和两种优化方法;第 5 节给出不同指标的实验曲线和性能分析;第 6 节结束语。

## 2 编码模式及一些分片概念

XML 数据对象通常被形象地表示为 DOM 树,树中的节点表示元素、属性及其值,树中的边表示 XML 数据之间的嵌套关系。通过对 XML 数据树编码能够有效地判断树中节点间的位置关系,因此编码被广泛应用于 XML 研究领域。这里首先介绍本文采用的一种编码模式,接着将刻画出片断和元素节点的空间特性。

在文献[2,6-10]有许多编码技术被提出,他们都能够有效地判断树中节点间的位置关系,在本文

中,我们试图利用编码模式来处理要划分的数据,从而提高结构连接的性能,因此,对于编码模式的选择是至关重要的。我们使用文献[6]提出的 Dietz 编码模式对 XML 文档进行编码,每一个元素节点都有一对属性值 (*preorder*, *postorder*),其中 *preorder* 表示在先序遍历 XML 文档中产生的元素节点序号, *postorder* 表示在后序遍历 XML 文档中产生的元素节点序号, Dietz 编码模式表示元素节点之间的位置关系如下所示:

(i) 任意两个元素节点  $e_i$  和  $e_j$ , 如果  $e_i$  是  $e_j$  的祖先, 那么一定满足  $Pre(e_i) < Pre(e_j)$  并且  $Post(e_i) > Post(e_j)$ 。

(ii) 如果元素  $e_i$  和  $e_j$  是兄弟节点, 在先序遍历中先遇到  $e_i$ , 那么一定有  $PRE(e_i) < PRE(e_j)$  并且  $POST(e_i) < POST(e_j)$ 。

在这里,  $Pre(e)$  代表节点  $e$  在先序遍历 XML 文中产生的节点序号,  $Post(e)$  代表节点  $e$  在后序遍历 XML 文中产生的节点序号, 图 1 给出了一棵利用 Dietz 编码模式进行编码得到的 XML 数据树。

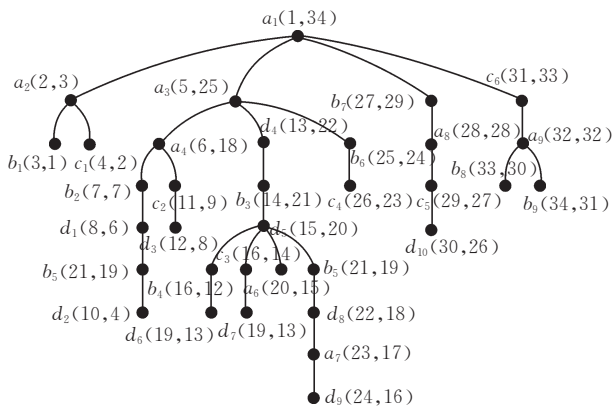


图 1 一个 XML 文档例子

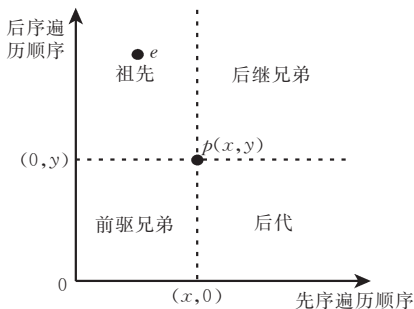


图 2 Dietz 编码模式应用示意图

正如图 2 所示, 利用 Dietz 编码模式, XML 文档树中结点之间的结构关系就被清晰地显示在 2 维平面中了。X 和 Y 坐标轴分别代表 *preorder* 和 *pos-*

torder 值,对于任何一棵 XML 文档数据树而言,任何两个节点的 *preorder* 和 *postorder* 值都不相同,由此得知,在 2 维平面中的每一个空间点对应唯一的元素节点.以图 2 中的节点 *p* 为例,在节点 *p* 左上方任取一个节点 *e*,该节点 *e* 肯定为节点 *p* 的祖先元素,因为  $Pre(e) < Pre(p)$  并且  $Post(e) > Post(p)$ ;同理,位于节点 *p* 右下方的任一元素都为节点 *p* 的后代元素;位于节点 *p* 左下方的任一元素都为节点 *p* 的前驱节点;位于节点 *p* 右上方的任一元素都为节点 *p* 的后继节点.

下面分析了 Dietz 编码模式和范围编码模式的不同,从而体现出本文选择 Dietz 编码模式的必要性.范围编码中 *starting* 指在先序遍历 XML 文档时产生的节点序号,*ending* 指在先序遍历过程中回溯

到该节点时的序号,任意给定两个元素 *a* 和 *b*,如果元素 *a* 是元素 *b* 的祖先,那么一定满足条件:  $a.starting < b.starting \ \&\& \ a.ending > b.ending$ .以图 1 中的文档为例,图 3(a)显示了采用 Dietz 编码模式对文档进行编码后元素节点在 2 维空间中的分布情况,图 3(b)显示了采用范围编码模式对文档进行编码后元素节点在 2 维空间中的分布情况.为了简单扼要的说明问题,在图 3 中我们仅仅给出了节点 *a* 和节点 *d* 的分布情况.由图 3 很明显的可以看出采用 Dietz 编码模式编码的元素节点比较均匀的分布在对角线附近,然而,采用范围编码模式编码的元素节点一般会集中分布在对角线的一侧,因为本文主要目的是想通过划分区域来尽量平均的对元素进行划分,所以选择了能使元素分布比较均匀的 Dietz 编码模式.

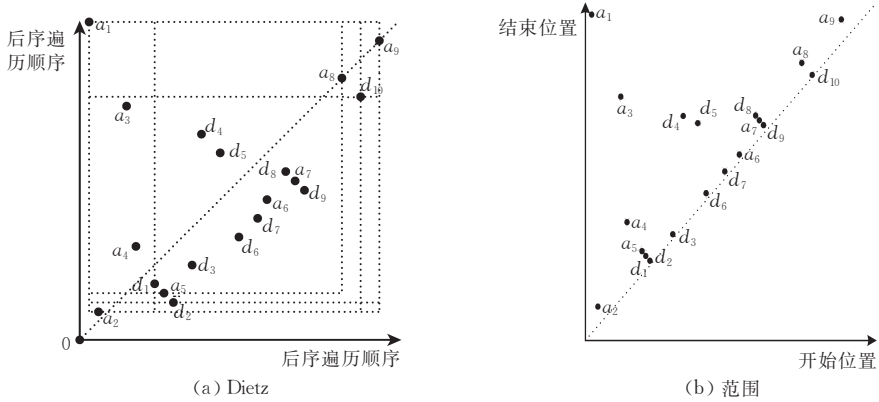


图 3 Dietz 编码和范围编码

在本文中,我们把图 2 中节点之间的关系扩展到 2 维空间中各个片断之间的关系,如图 4 所示,在这里我们把该 2 维空间划分成 9 个矩形区域即  $R_1, R_2, \dots, R_9$ . 每个  $R_i$  (其中  $i=1, 2, \dots, 9$ ) 都包含 XML 文档中元素节点的子集.通过分析我们可以得出以下 9 个特征:

- (1)  $R_1$  中任意的元素节点是  $R_5$  中任意元素节点的祖先元素;
- (2)  $R_3$  中任意的元素节点是  $R_5$  中任意元素节点的后继节点;
- (3)  $R_7$  中任意的元素节点是  $R_5$  中任意元素节点的前驱节点;
- (4)  $R_9$  中任意的元素节点是  $R_5$  中任意元素节点的后代元素;
- (5)  $R_2$  中一些元素节点可能是  $R_5$  中某一元素节点的祖先元素也可能是后继节点;
- (6)  $R_4$  中一些元素节点可能是  $R_5$  中某一元素节

点的祖先元素也可能是前驱节点;

(7)  $R_6$  中一些元素节点可能是  $R_5$  中某一元素节点的后代元素也可能是后继节点;

(8)  $R_8$  中一些元素节点可能是  $R_5$  中某一元素节点的后代元素也可能是前驱节点;

(9) 在  $R_5$  内部一些元素节点可能与某一元素节点存在连接关系.

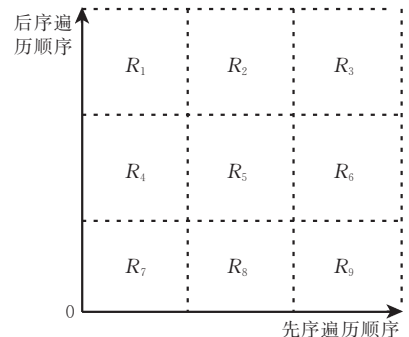


图 4 分片

值得注意的是,特征(1)~(4)发现的结构关系可以适用于两个区域中所有的元素节点;特征(5)~(8)发现的结构关系仅适用于两个区域中的一部分元素节点;特征(9)说明了一个问题就是在一个区域内部利用片段的位置关系不能确定节点之间的关系.通过对上面特征的分析,我们可以用基于分片的方法来处理结构连接问题.在这里我们通过查询区域  $R_5$  中某一元素节点的祖先元素为例进行分析:

(1) 区域  $R_5$  和  $R_3, R_6, R_7, R_8, R_9$  都不用做连接,因为在这 5 个区域中任意节点都不是  $R_5$  中某个元素的祖先;

(2) 区域  $R_5$  和  $R_1$  中的元素节点不用做结构连接可以直接输出连接结果,因为区域  $R_1$  中的所有元素节点都是区域  $R_5$  中的元素节点的祖先;

(3) 区域  $R_5$  和  $R_2, R_4$  中的元素需要做结构连接,因为区域  $R_2, R_4$  中只有一部分元素节点是区域  $R_5$  中某些元素节点的祖先;

(4) 区域  $R_5$  内部的元素只能作结构连接才能得到祖先后代关系.

我们在此定义两个判定谓词,即  $all(S, R_i, R_j)$  和  $some(S, R_i, R_j)$ , 其中  $R_i, R_j$  是 2 维空间中的两个区域,在这里我们称为片断,  $S$  定义为类型指针,它包括 *ancestor*, *descendant*, *following* 和 *preceding*.  $all(S, R_i, R_j)$  所代表的含义是  $R_i$  中所有节点与  $R_j$  中所有节点都存在关系  $S$ ,  $some(S, R_i, R_j)$  所代表的含义是  $R_i$  中部分节点与  $R_j$  中部分节点存在关系  $S$ , 为了进一步说明问题以图 4 为例,  $all(ancestor, R_1, R_5)$  为真,  $all(ancestor, R_5, R_1)$  为假,  $all(ancestor, R_4, R_5)$  为假. 所以说, 只有当片断  $R_i$  位于  $R_j$  片断的“左上”时  $all(ancestor, R_i, R_j)$  才为真, 同理, 只有当片断  $R_i$  位于  $R_j$  片断的“右下”时  $all(descendant, R_j, R_i)$  才为真.

对于一个片断  $R_i$ , 可能会存在两种片断  $R_j$  使得  $some(ancestor, R_j, R_i)$  为真,  $R_j$  位于片断  $R_i$  这一区域的上面或左面. 例如  $R_2$  位于片断  $R_5$  区域的上面, 记作  $up(R_2, R_5)$ , 所以  $some(ancestor, R_2, R_5)$  为真, 而  $some(ancestor, R_4, R_5)$  也为真, 因为  $R_4$  位于片断  $R_5$  区域的左侧, 记作  $left(R_4, R_5)$ . 同理, 如果要满足  $some(descendant, R_j, R_i)$  为真, 那么必须保证片断  $R_j$  位于片断  $R_i$  这一区域的下面或右侧, 记作  $down(R_j, R_i)$  和  $right(R_j, R_i)$ . 基于上面分析, 我们总结出 4 个性质.

**性质 1.** 给定两个片断  $R_i$  和  $R_j$ , 存在 3 个节点  $x \in R_i, y \in R_j, z \in R_j$ , 如果  $up(R_j, R_i)$  为真, 节点  $y$  是节点  $x$  的祖先, 并且有  $Pre(z) < Pre(y)$ , 那么节点  $z$  也一定是节点  $x$  的祖先.

**性质 2.** 给定两个片断  $R_i$  和  $R_j$ , 存在 3 个节点  $x \in R_i, y \in R_j, z \in R_j$ , 如果  $left(R_j, R_i)$  为真, 节点  $y$  是节点  $x$  的祖先, 并且有  $Post(z) > Post(y)$ , 那么节点  $z$  也一定是节点  $x$  的祖先.

**性质 3.** 给定两个片断  $R_i$  和  $R_j$ , 存在 3 个节点  $x \in R_i, y \in R_j, z \in R_j$ , 如果  $down(R_j, R_i)$  为真, 节点  $y$  是节点  $x$  的后代, 并且有  $Pre(z) > Pre(y)$ , 那么节点  $z$  也一定是节点  $x$  的后代.

**性质 4.** 给定两个片断  $R_i$  和  $R_j$ , 存在 3 个节点  $x \in R_i, y \in R_j, z \in R_j$ , 如果  $right(R_j, R_i)$  为真, 节点  $y$  是节点  $x$  的后代, 并且有  $Post(z) < Post(y)$ , 那么节点  $z$  也一定是节点  $x$  的后代.

对片断内部的节点进行排序后, 通过这些性质的应用可以极大地减小结构连接的工作量. 文献[9]中, Grust 利用编码属性来判断节点之间的位置关系, 然而本文主要关注的是通过对编码属性的分析来判断片断之间的位置关系, 从而提高结构连接的性能.

### 3 一种新的基于分片的结构连接方法

在本节中, 我们提出一种新的基于分片的结构连接方法来解决祖先后代查询问题. 我们的方法可以很容易地扩展到其它的 XPath 轴, 比如前驱、后继和兄弟关系等等. 为了更加清晰地阐述基于分片的结构连接方法, 考虑查询 `section//title` 作为例子.

首先, 我们对该连接方法涉及到的几种情况进行分析, 接着我们将给出基于分片的结构连接算法. 图 5(a) 给出了一般情况下的示意图, 片断  $D$  是指包括所有后代元素在内的最小矩形区域, 在片断  $D$  所在的区域可能存在一些祖先元素, 那么这一部分祖先元素所在的区域就构成了片断  $A_1$ . 这样, 所有祖先元素构成的一个 2 维空间经过  $A_1$  的分割将被划分成 4 个片断(区域)为  $A_1, A_2, A_3$  和  $A_4$ . 考例子查询 `section//title`, 所有 title 点所构成的区域形成片断  $D$ , 所有落在区域  $D$  中的 section 点构成片断  $A_4$ ; 落在区域  $A_1, A_2, A_3$  中的 section 点分别构成了 section 的分片片断  $A_1, A_2, A_3$ .

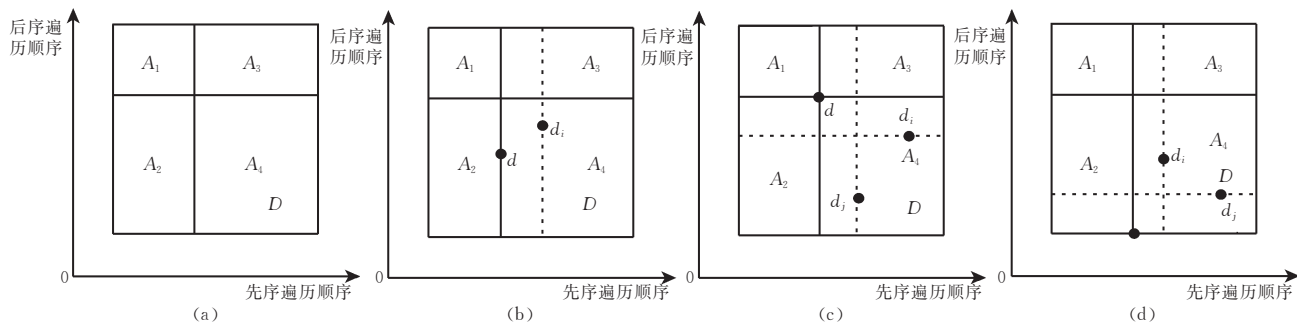


图 5 片断调整的几种情况

在这里我们采用逐步调整的策略来进行连接操作,也就是说,遍历一次后代元素,在遍历的同时对可能存在祖先元素的片断与每个后代元素进行连接,随着后代元素的减少,片断  $D$  和片段  $A_4$  也会相应地变小,这样,片断  $A_1, A_2, A_3$  和  $A_4$  都会进行相应的变化,直到遍历完后代元素.在遍历后代元素的过程中,针对一个后代元素  $d$  具体有以下 3 种调整情况.

(1)  $d$  位于片断  $A_2$  和片断  $D$  的边界线的中部(除了边界线的两个端点外的部分),如图 5(b)所示.

(2)  $d$  位于片断  $A_2$  和片断  $D$  的边界线上面的端点部分,如图 5(c)所示.

(3)  $d$  位于片断  $A_2$  和片断  $D$  的边界线下面的端点部分,如图 5(d)所示.

这里假定每个片断  $A$  中的祖先元素是按 *pre-order* 升序排列的,对于情况(1)来说,我们直接输出后代元素  $d$  和  $A_1$  中的所有祖先元素  $a$  作为部分查询结果,接着反向遍历  $A_2$  中的祖先元素  $a_i$  与后代元素  $d$  进行比较判断,如果  $a_i$  不是  $d$  的祖先,那么  $d$  后面也不存在  $a_i$  的后代元素,即  $a_i$  与  $d_i$  不用做连接.否则,如果  $a_i$  是  $d$  的祖先,那么根据性质 2 可知,  $a_i$  之前的  $a_j$  (属于  $A_2$ ) 都是  $d$  的祖先元素直接输出而不用进行比较连接.处理完  $d$  以后,  $A$  的 4 个片段会做相应的变化,  $A_3$  中的部分元素会划分到  $A_1$  中,  $A_4$  中的部分祖先元素会划分到  $A_2$  中,这样,  $A_1, A_2$  片段的区域会增大,而  $A_3, A_4$  片段的区域会减小.图 5(b)给出了调整后的示意图.其它两种情况如图 5(c)、图 5(d)中虚线所示进行调整重新划分片段区域.这里给出基于分片的结构连接算法 P-Join.

#### 算法 1. 基于分片的结构连接算法.

输入: 祖先元素  $A$ , 后代元素  $D$

输出: 连接结果

方法:

1.  $A_1, A_2, A_3, A_4 \leftarrow \text{Partition}(A, D)$
2. for  $d \in D$  (*sorted*)

3. do switch
4. case 1:  $\text{posFlag} \leftarrow \text{MIDDLE}$
5. case 2:  $\text{posFlag} \leftarrow \text{TOP}$
6. case 3:  $\text{posFlag} \leftarrow \text{BOTTOM}$
7.  $\text{Output}(A_1, d)$
8. switch
9. case  $\text{posFlag} = \text{MIDDLE}$ :
10.  $\text{Locate}(A_2, d)$
11.  $\text{Output}(A_2, d)$
12.  $\text{AdjustMiddle}(A, D)$
13. case  $\text{posFlag} = \text{TOP}$ :
14.  $\text{AdjustTopLeft}(A, D)$
15. case  $\text{posFlag} = \text{BOTTOM}$ :
16.  $\text{Output}(A_2, d)$
17.  $\text{AdjustBottomLeft}(A, D)$

P-Join 算法的优势在于,通过使用动态边界调整技术来过滤一些不产生连接结果的元素,这样就减小了结构连接的工作量.过滤技术在数据库系统中应用十分广泛,通过减少 I/O 代价来改善系统性能.在关系数据库中,传统的过滤技术<sup>[11-12]</sup>主要通过判断两个关系的属性值是否相等来进行过滤.然而,对于 XML 半结构化数据来说,传统的过滤技术不再适用,所以我们提出一种新的过滤技术,利用空间位置特性来过滤元素节点.

通过图 3(a)说明本文是如何利用空间位置特性来进行过滤的.首先,我们通过后代元素  $d$  的分布形成一个包括所有后代元素的面积最小的矩形  $D$ ,利用该矩形我们可以过滤  $A$  中的部分元素,凡是出现在区域  $D$  的右侧和下面  $A$  中的点都可以被区域  $D$  所过滤掉,因为这些点肯定不是区域  $D$  的祖先.这样,肯定没有后代元素的祖先元素  $a_2$  和  $a_9$  就被过滤掉了,我们就从  $A$  中移除  $a_2$  和  $a_9$  两个元素.接着,我们把剩余的所有祖先元素根据其分布也形成一个面积最小的矩形  $A'$ ,通过矩形  $A'$  来过滤后代元素  $d$ ,落在区域  $A'$  上面和左侧  $D$  中的点都可以被区域  $A'$  过滤掉,因为他们不可能是区域  $A'$  中任



何点的后代, 在这里, 对于过滤  $A$  和  $D$  的优先顺序对结果没有影响。

## 4 空间划分方法及优化技术

在本节中, 我们将进一步深入研究当一个查询  $a/d$  到来时, 如何有效划分  $A$  和  $D$  的问题, 这里我们将提出一种空间划分方法和两种优化技术, 并且讨论一种新的数据结构是如何对分片结果进行索引的。最后, 基于空间划分方法的基础上, 本文提出两种新的结构连接算法。

### 4.1 空间划分

对  $A$  和  $D$  的划分本身就是一个复杂和费时的过程, 但是划分合理能极大地降低结构连接操作的工作量, 所以在这里我们提出一种高效的划分方法。尽管对  $A$  和  $D$  划分的优先顺序对结果数没有影响, 但是可能会影响查询效率。当后代元素的数量很大时, 嵌套层数可能很大, 分布也将不均匀, 这时候更多地考虑对  $D$  的划分将能使查询性能得到很好的改善。相反, 如果祖先元素的数量很大时, 那么, 这时候更多地考虑对  $A$  的划分将能使查询性能得到很好的改善。因为在这里对  $A$  和  $D$  的划分过程本身是对称的, 所以我们主要是集中讨论对  $D$  的划分方法, 该方法对  $A$  也适用。

理论上, 元素节点可以分布在 2 维空间的任何地方, 但是, 经过对 XML 文档的分析, 一般情况下, 元素节点大部分都分布在 2 维空间的从左下到右上对角线部分。基于 XML 文档的这种特性, 我们提出一种高效的划分方法, 记作 Spatial-Partitioning。

由于元素分布的特性, 可知在整个大矩形的边界上最多有 4 个点分布, 从这 4 个点中选出足够远的两个点是很容易的, 所以每次选择出两个点作为入口点。如果要把后代元素划分成  $N$  个片断, 那么可以利用  $Nearest()$  方法找到距离选择的入口点最近的  $|D|/N$  个后代元素。 $Nearest()$  的实现过程如算法 2 所示。其中,  $Furthermost(x, nX)$  表示从元素集合  $nX$  中选择距离  $x$  元素最远的元素节点,  $Distance(e, x)$  表示元素  $e$  和元素  $x$  在 2 维平面中的距离,  $X-Distance(p, x)$  表示  $p$  和  $x$  两节点的水平距离。

#### 算法 2. 元素的空间划分算法。

输入: 入口元素节点  $x$ , 片断应包含元素数量  $y$ , 后代元素集合  $D$

输出: 找到距离  $x$  最近的  $y$  个元素

方法:

1.  $nX \leftarrow x$ 's nearest  $n$  points in  $x$ -axis
2.  $e \leftarrow Furthermost(x, nX)$
3.  $r \leftarrow Distance(e, x)$
4.  $p \leftarrow \min Preorder node in the left nodes$
5. while  $X-Distance(p, x) < r$
6.     do if  $Distance(p, x) < r$
7.         then  $nX \leftarrow nX - e + p$
8.          $e \leftarrow Furthermost(x, nX)$
9.          $r \leftarrow Distance(e, x)$
10.         $p \leftarrow p.next$
11. return  $nX$

现在结合图 6 来说明  $Nearest()$  是如何划分后代元素的。在这里, 后代元素是按  $preorder$  有序排列的。我们假设要找到距离元素  $x$  最近的 5 个元素节点, 首先从有序链表  $D-List$  中选择前 5 个元素放入  $nX$  集合中, 也就是  $\{x, e_1, e_2, e_3, e_4\}$ 。在这个集合中距离  $x$  最远的元素是  $e_1$ ,  $r_1$  为元素  $x$  和元素  $e_1$  的距离。又因为如果一个元素和  $x$  的水平距离小于  $r_1$ , 则这个元素就有可能是我们所要查找的结果元素, 所以继续从  $D-List$  中取后代元素  $e_5$ , 经过计算  $x$  和  $e_5$  距离要小于  $r_1$ , 所以把  $e_1$  从集合中移出并且放回到  $D-List$  中, 接着插入  $e_5$ , 这时的集合  $nX$  为  $\{x, e_2, e_3, e_4, e_5\}$ 。同理,  $e_6$  替代  $e_3$ , 因为  $X-Distance(e_7, x) > r_3$ , 所以对一个片断的划分结束, 最终集合为  $\{x, e_2, e_4, e_5, e_6\}$ 。

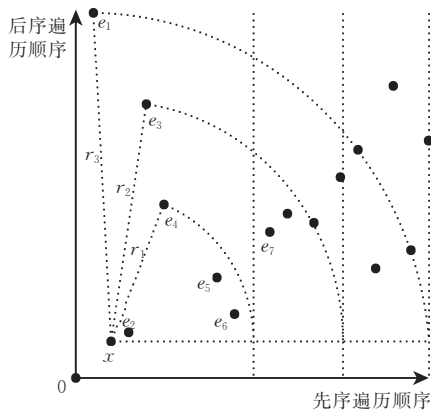


图 6 寻找  $n$  个距离最近的节点

由上面的例子可以看出, 当需要把后代元素  $D$  划分成  $N$  个片断时, 即每个片断应该分到  $|D|/N$  个元素,  $Nearest()$  函数只是从部分后代元素而不是所有的后代元素中选择满足条件的  $|D|/N$  个元素, 也就是说, 只遍历部分后代元素, 因此这种分片方法在分片效率方面有很大的优越性。算法 3 给出了 Spatial-Partitioning 算法的实现过程。

**算法 3.** Spatial-Partitioning 算法.

输入：后代元素集合  $D$ , 要划分的片断数量  $N$

输出：后代元素被划分成  $N$  个片断

方法：

```

1. switch
2.   case  $N=1$ :
3.     return  $D_i$ 
4.   case  $N=2$ :
5.     choose  $p_1$  and  $p_2$  in  $D$ 
6.      $D_i \leftarrow \text{Nearest}(p_1, |D|/N)$ 
7.      $D_{i+1} \leftarrow \text{Nearest}(p_2, |D|/N)$ 
8.     return  $D_i, D_{i+1}$ 
9.   case  $N>2$ 
10.    choose  $p_1$  and  $p_2$  in  $D$ 
11.     $D_i \leftarrow \text{Nearest}(p_1, |D|/N)$ 
12.     $D_{i+1} \leftarrow \text{Nearest}(p_2, |D|/N)$ 
13.     $D \leftarrow D - D_i - D_{i+1}$ 
14.     $N \leftarrow N - 2$ 
15.     $i \leftarrow i + 2$ 
16.    Spatial-Partitioning( $D, N, i$ )
17.    return  $D_1, D_2, \dots, D_N$ 

```

该算法是采用递归方法来实现的, 当  $N=1$  时, 我们只需要把所有的后代元素  $D$  划为一个片断; 当  $N \neq 1$  时, 因为矩形的边界上最多存在 4 个点, 所以每次需要从边界上的点中选择两个作为入口点, 再从后代元素  $D$  中选择距离每个入口点最近的  $|D|/N$  个元素; 递归调用算法 *Spatial-Partitioning()* 直到后代元素划分完为止. 为了说明该算法, 这里给出一个简单的例子. 假设  $N=2$ , 后代元素在 2 维空间中的分布如图 3(a) 所示, 首先我们选择两个足够远的点  $d_1$  和  $d_{10}$ , 并且知道  $|D|/N=5$ , 所以距离  $d_1$  最近的 5 个后代元素为  $d_1, d_2, d_3, d_5$  和  $d_6$  被划分给  $D_1$ , 这样其余的元素  $d_4, d_7, d_8, d_9$  和  $d_{10}$  就被划分给  $D_2$ . 最终的划分结果如图 7 所示.

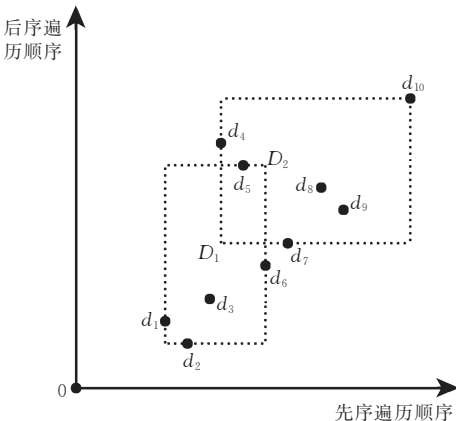


图7 划分后代元素片断

**4.2 优化划分方法**

尽管后代元素按个数被平均划分到各个片断中, 但是我们的目的是要做祖先后代查询, 所以说, 后代元素在各个片段的划分情况在很大程度上会影响祖先元素的划分, 从而也会影响查询性能. 所以在这里我们提出两种优化划分后代元素的方法.

第一种方法是最大面积最小化, 也就是在 2 维空间区域中选择矩形面积最大的一个片断, 利用 *MinMax()* 对其所在的后代元素进行重分布调整优化, 使得有共同祖先的后代元素尽量落在同一片断中, 这样就减少了祖先元素在不同片断中的复制, 从而降低了结构连接操作的工作量. 算法 4 给出了 *MinMax()* 实现算法.

**算法 4.** 最大面积最小化算法.

输入:  $N$  个后代元素片断

输出: 使面积最大的矩形最小化

方法:

```

1.  $D_i \leftarrow \text{MaxArea}(D_i) (i=1 \dots N)$ 
2.  $D_j = \text{NearestArea}(\text{Center}(D_i))$ 
3.  $d = \text{NearestPoint}(D_i, D_j)$ 
4. if  $\text{Area}(D_i + d) < \text{Area}(D_i)$ 
5.   then Merge( $D_j, d$ )
6.   Cut( $D_i, d$ )
7.   return MinMaxD( $D_1, D_2, \dots, D_N$ )
8. else return  $D_1, D_2, \dots, D_N$ 

```

该算法首先从  $D_1$  到  $D_N$  中找到面积最大的矩形  $D_i$  和任意一个矩形  $D_j (i \neq j)$ , 但是必须满足矩形  $D_j$  的中心点距离矩形  $D_i$  的中心点最近, 接着从矩形  $D_i$  的边界上找一个满足条件的元素节点  $p$ , 它必须保证节点  $p$  从矩形  $D_i$  移走后使得新矩形  $D_i$  的面积减小, 同时它还必须保证得到节点  $p$  的矩形  $D_j$  的面积要小于矩形  $D_i$  变化前的面积. 只有满足了这两个条件节点  $p$  才能从矩形  $D_i$  中调整到矩形  $D_j$  中, 反复调用该算法直到结束为止.

下面结合图 7 来说明 *MinMax()* 算法的具体执行过程, 这里只存在两个矩形  $D_1$  和  $D_2$ , 其中矩形  $D_2$  的面积最大, 所以需要对矩形  $D_2$  进行调整. 这时需要从矩形  $D_2$  边界上存在的元素节点中选择距离矩形  $D_1$  中心点最近的一个元素节点  $d_7$ , 因为从  $D_2$  中移除  $d_7$  后矩形  $D_2$  的面积变小了, 同时矩形  $D_1$  得到  $d_7$  后的面积比矩形  $D_2$  变化前的面积仍小, 所以  $d_7$  就被划分到  $D_1$  中了. 这时比较变化后的两矩形  $D'_1$  和  $D'_2$  的面积,  $D'_1$  的面积要大于  $D'_2$  的面积, 所以要调整  $D'_1$ , 方法同上, 经过计算比较, 将  $d_5$  从  $D'_1$  中划分到  $D'_2$  中. 直到不存在需要调整的矩形为止. 最终调整结果如图 8 所示.

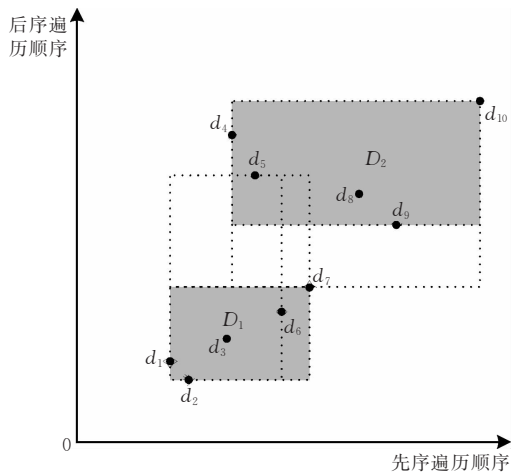


图 8 最小化最大面积

第二种优化方法是减小矩形间的相交面积. 也就是使得落在相交区域内的后代元素尽量少, 这样可以减少后代元素在各片断中的复制, 因为有相同祖先的兄弟节点能尽量集中划分到一个片断中, 这就使得祖先元素复制减少, 从而降低了结构连接的工作量. 优化算法  $MinOverlay()$  如算法 5 所示. 该算法首先判断在  $D_1$  到  $D_N$  中是否存在两个片断相交, 如果任意两个片断都不相交那么跳过该算法, 否则, 存在 3 种情况来处理相交区域  $O$ :

#### 算法 5. 减小相交面积算法.

输入:  $N$  个后代元素片断

输出: 减小了相交面积的矩形

方法:

1. if  $(D_i \cap D_j) = \text{NIL} (i \neq j)$
2. then return  $D_1, \dots, D_N$
3.  $SelectUndisposed(O) = D_i \cap D_j$
4. switch
5. case no point in  $O$ :
6. case points are not on the edge of  $O$ :
7. return  $D_1, D_2, \dots, D_N$
8. case points are on the edge of  $O$ :
9. for each  $p \in Edge(O)$
10. do if  $p \in D_i$
11. then  $D_i \leftarrow D_i - p$
12.  $D_j \leftarrow D_j + p$
13. else  $D_j \leftarrow D_j - p$
14.  $D_i \leftarrow D_i + p$
15. return  $MinOverlay(D_1, D_2, \dots, D_N)$

(i) 在相交区域  $O$  中不存在元素节点, 所以不用处理;

(ii) 在相交区域  $O$  中存在元素节点, 但是区域  $O$  的边界上不存在元素节点, 在这种情况下, 调整节点不会影响相交区域  $O$  的面积, 所以不用处理;

(iii) 在相交区域  $O$  中存在元素节点, 而且区域  $O$  的边界上至少存在一个元素节点, 这时需要把边界上的元素节点划分给某个片断, 该片断的中心点要距离调整的元素节点最近, 从而保证相交区域  $O$  的面积减小.

下面仍然以图 7 为例来说明该算法的执行过程. 矩形  $D_1$  和  $D_2$  有相交区域, 而且有节点  $d_5$  落在相交区域边界上了, 因此  $d_5$  从  $D_1$  调整到  $D_2$  中, 结果调整后的矩形  $D_1$  和  $D_2$  不再存在相交区域, 而且使得矩形  $D_1$  的面积减小了. 如果还有其它的矩形需要进行优化, 那么继续执行该算法, 在这里不存在其它的相交区域了, 所以该算法结束, 利用优化后的结果如图 9 所示.

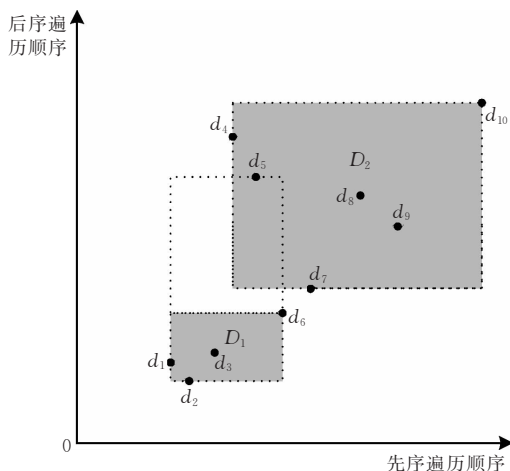


图 9 优化相交面积

### 4.3 片段的索引结构

过去在对空间数据的索引和查询上有大量的研究成果, R-tree<sup>[13]</sup>、R\*-tree<sup>[14]</sup>和 M-tree<sup>[15]</sup>是处理空间数据最常见的索引方法, 而且 R-tree 和 M-tree 在查找操作上更快捷, 所以在本文中也可以使用这两种数据结构, 但是 XML 数据不同于其它的空间数据, 因为它有自己内在的结构特性, 其中包括本文前面分析到的一些节点和片段的特征和性质, 因此我们采用了一种更加有效的方法作索引来处理数据片断.

下面给出了本文采用的索引结构的基本构造过程. 第一步利用 Spatial-Partitioning 算法把后代元素  $D$  划分成  $N$  个片断, 分别为  $D_1, D_2, \dots, D_N$ ; 第二步利用 B+ 树做数据结构来存储划分后的片断结果, 对于每一个  $D_i$  都会对应着一个祖先元素的片断  $A_i$ , 而且每一个片断  $A_i$  被划分成 4 个小片断  $A_{i1}, A_{i2}, A_{i3}$  和  $A_{i4}$ . 这样构建索引结构的过程就结束了. 我们称这种索引结构为 Partitioned Spatial Structural Index tree (PSS-tree). PSS-tree 结构如图 10 所示.



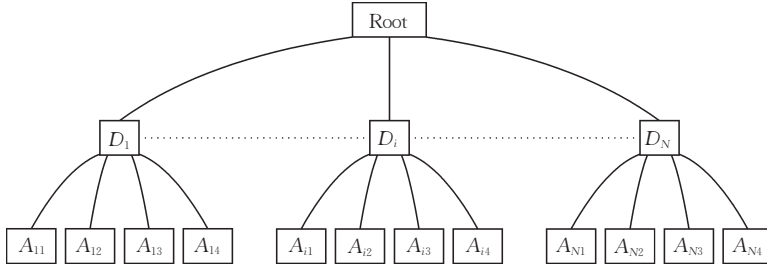


图 10 PPS-tree 的数据结构

#### 4.4 基于分片的空间结构连接算法

在 Spatial-Partitioning 方法的基础上本节提出了两种基于分片的空间结构连接算法：Partitioned Spatial Structural Join (PSSJ) 和 Optimized Partitioned Spatial Structural Join (OPSSJ)。PSSJ 算法首先过滤肯定不产生连接结果的祖先和后代元素；接着利用 Spatial-Partitioning 方法来划分后代元素  $D$ ；再利用 PSS-tree 作存储结构来索引每个后代元素片断和对应的祖先元素片断；最后，调用  $P\text{-Join}()$  方法进行连接操作。算法 6 给出了 PSSJ 算法的实现过程。

##### 算法 6. 基于分片的结构连接算法 (PSSJ)。

输入：祖先元素  $A$ , 后代元素  $D$

输出：连接结果

方法：

1.  $Filter(A, D)$
2.  $Spatial\text{-}Partitioning(D, N)$
3. for each  $D_i \in \{D_1 \cdots D_N\}$
4.     do  $PPSTree.Insert(D_i, A_i)$
5. for each  $i \in 1 \cdots N$
- do  $P\text{-Join}(A_i, D_i)$

OPSSJ 算法的处理过程与 PSSJ 算法类似。唯一不同的是当后代元素  $D$  划分完以后，OPSSJ 算法要调用两个优化方法来对分片结果进行优化。OPSSJ 算法如算法 7 所示。

##### 算法 7. 基于分片的空间结构连接优化算法 (OPSSJ)。

输入：祖先元素  $A$ , 后代元素  $D$

输出：连接结果

方法：

1.  $Filter(A, D)$
2.  $Spatial\text{-}Partitioning(D, N)$
3.  $MinMaxD(D_1 \cdots D_N)$
4.  $MinOverlay(D_1 \cdots D_N)$
5. for each  $D_i \in \{D_1 \cdots D_N\}$
6.     do  $PPSTree.Insert(D_i, A_i)$
7. for each  $i \in 1 \cdots N$
- do  $P\text{-Join}(A_i, D_i)$

#### 4.5 算法复杂性分析

在本小节，我们主要是分析本文提出的两种算法 PSSJ 和 OPSSJ 算法的复杂性。考虑结构连接  $A//D, |A|, |D|$  分别表示集合  $A$  和  $D$  的基数。假定  $N$  为片断的个数。

过程  $Filter(A, D)$  需要使用  $A$  来过滤  $D$ ，且使用  $D$  来过滤  $A$ ，因此该过程的复杂度为  $O(|A| + |D|)$ 。 $Spatial\text{-}Partitioning(D, N)$  实际上是一个递归的过程，递归的次数为  $N/2$ ，而每次将当前的集合  $D$  分为两个片断，因此该过程的复杂度为

$$O\left(\frac{|D|}{2^0} + \frac{|D|}{2^1} + \cdots + \frac{|D|}{2^{\frac{N}{2}-1}}\right) \approx O(|D|).$$

由于索引 PPSTree 的特殊性，PPSTree.Insert 操作的复杂度可以看作是一个常数，因此算法 PSSJ 的 3~4 行和算法 OPSSJ 的 5~6 行的复杂度为  $O(N/2)$ 。而过程  $P\text{-Join}(A_i, D_i)$  为  $O(|A|/N + |D|/N)$ ，这样算法 PSSJ 的 5~6 行和算法 OPSSJ 的 7~8 行的复杂度为  $O(|A| + |D|)$ 。如果假设  $N \ll \min\{|A|, |D|\}$  的话，则 PSSJ 算法的总的复杂度为  $O(|A| + |D|)$ 。对于 OPSSJ 算法来讲，与 PSSJ 算法相比有两个优化的过程： $MinMaxD$  和  $MinOverlay$ 。通过算法 4 和算法 5 可以看出这两个优化过程的复杂度为  $O(|D|/N)$  和  $O(|D|^2/N^2)$ 。当  $N \ll \min\{|A|, |D|\}$  时，算法 OPSSJ 的复杂度可以被估算为  $O(|A| + |D|^2)$ 。

## 5 实验评估

这里将给出实验平台和一些测试集，通过这些来考察本文提出的基于分片的结构连接算法的性能及应用范围。

### 5.1 测试集

PC 机的硬件配置为处理器 Intel Pentium 2.60MHz，内存 512MB，硬盘 40GB。

测试软件环境为操作系统为 Microsoft Windows XP，采用自己开发的面向对象数据库管理系统 Fish 来存储 XML 数据信息，测试程序为 Forte C++ 执行代码。

在以上环境下,分别对本文提出的 PSSJ 算法,及它的改进算法 OPSSJ,还有文献[2]提出的 Stack Tree Join(STJ)算法和文献[4]提出的 XR-tree 算法分别进行了测试.到目前为止,有很多有关 XML 连接方面的连接算法,包括 PathStack 连接、Twig-Stack 连接等,它们是为简单路径表达式和 Twig 查询而设计的 Holistic 连接算法,而本文主要是研究基于分片思想的结构连接(A-D)算法. Stack Tree Join 算法是第一个提出解决结构连接的方法,而 XR-tree 算法则是目前性能最好的一种结构连接算法,因此本文选择这两种结构连接算法作为性能分析比较的对象.为了通过控制 XML 文档的结构以及连接特性来体现不同算法查询性能的差异,本文所采用的测试集是自己合成的.我们用 IBM XML 数据生成器<sup>[15]</sup>来生成数据文档,采用的 DTD 如图 11 所示.由每个 DTD 分别生成了从 20MB 到 100MB 大小不同的测试文档.

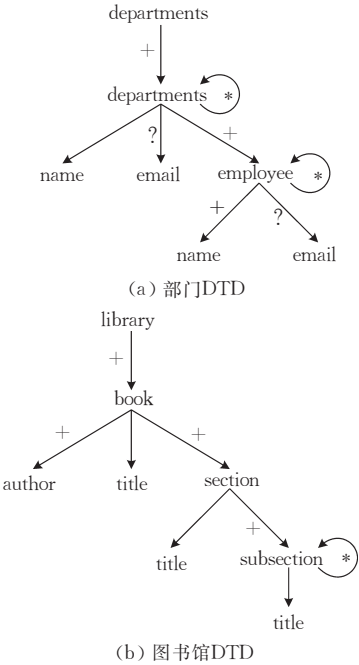


图 11 用于合成数据的 DTDs

每个 DTD 给出了两个查询语句,这是为了更加全面地考察本文提出的新算法的适用范围,表 1 给出了测试查询性能的测试集.因为在查询语句中祖先后代的嵌套对查询效率有很大的影响,所以表 2 给出了不同查询语句的嵌套情况,这也是考察一个算法优劣的指标.

表 1 查询集		
查询	描述	DTD
Q <sub>1</sub>	department//employee	Department
Q <sub>2</sub>	employee//name	Department
Q <sub>3</sub>	section//subsection	Library
Q <sub>4</sub>	section//title	Library

表 2 查询语句的嵌套情况

查询	Anc Nested	Desc Nested
Q <sub>1</sub>	highly	highly
Q <sub>2</sub>	highly	less
Q <sub>3</sub>	less	highly
Q <sub>4</sub>	less	less

5.2 划分的片断数量对查询性能的影响分析

本小节实验的目的是为了测试在查询和文档一定的情况下,划分的片断数量不同对 PSSJ 算法和 OPSSJ 算法性能的影响,图 12 给出了 4 条查询语句 Q<sub>1</sub>, Q<sub>2</sub>, Q<sub>3</sub> 和 Q<sub>4</sub> 随着片断数量划分的变化对响应时间的影响.其中,图 12(a)、图 12(b)分别给出了后代元素被划分为 10~60 个不同片断时查询语句 Q<sub>1</sub> 和 Q<sub>2</sub> 随着划分数量的不同响应时间的变化曲线图,图 12(c) 给出了后代元素被划分为 20~120 个片断时查询语句 Q<sub>3</sub> 随着划分数量的不同响应时间的变化曲线图,图 12(d) 给出了后代元素被划分为 30~180 个片断时查询语句 Q<sub>4</sub> 随着划分数量的不同响应时间的变化曲线图.

从图 12 中可以看出,随着划分数量的增大 PSSJ 算法和 OPSSJ 算法的性能都得到了很大的改善,从而证明了本文提出的划分片断的方法在改善查询性能上是很有效的,同时也证明了在相同条件下优化算法 OPSSJ 要优于 PSSJ 算法.通过对图 12 中各种情况的观察,我们还会发现当划分数量达到一定程度时查询性能提高得越来越慢,这与祖先后代元素的数量以及在文档中的嵌套情况有关,祖先后代元素数量越多,分片就越能改善查询性能.

5.3 测试连接选择率的变化对不同算法的影响

在第二组实验中,我们研究不同的选择率对不同算法的影响,从而来说明各种算法的适用范围.为了说明问题,在测试过程中文档大小和元素的划分数量保持不变,但是祖先和后代元素的连接选择率是变化的.

图 13 给出了当连接选择率变化时 4 个不同算法(PSSJ、OPSSJ、StackTree 和 XR-Tree)的性能变化曲线.其中图 13(a),(b)分别测试了 Q<sub>1</sub> 和 Q<sub>2</sub> 查询,这两条查询中祖先元素的嵌套很深,在这样的条件下的结果曲线上可以看出,连接选择率对 XR-Tree 算法的影响很大,但是对其它 3 种算法影响不大,而且在不同的选择率情况下,其它 3 种算法的性能都要优于 XR-Tree 算法的性能.图 13(c),(d)分别测试了 Q<sub>3</sub> 和 Q<sub>4</sub> 查询语句,在两条查询中祖先元素的嵌套少,在这样的条件下从结果曲线上可以看出,连接选择率对这 4 种算法都有很大的影响,而且在这种情况下,XR-Tree 算法的性能要优于 Stack-

Tree 算法的性能,与算法 PSSJ 和 OPSSJ 的性能差不多.从上面的分析可以看出本文提出的两个结构

连接算法在选择率变化的情况下仍能保持很好的性能.

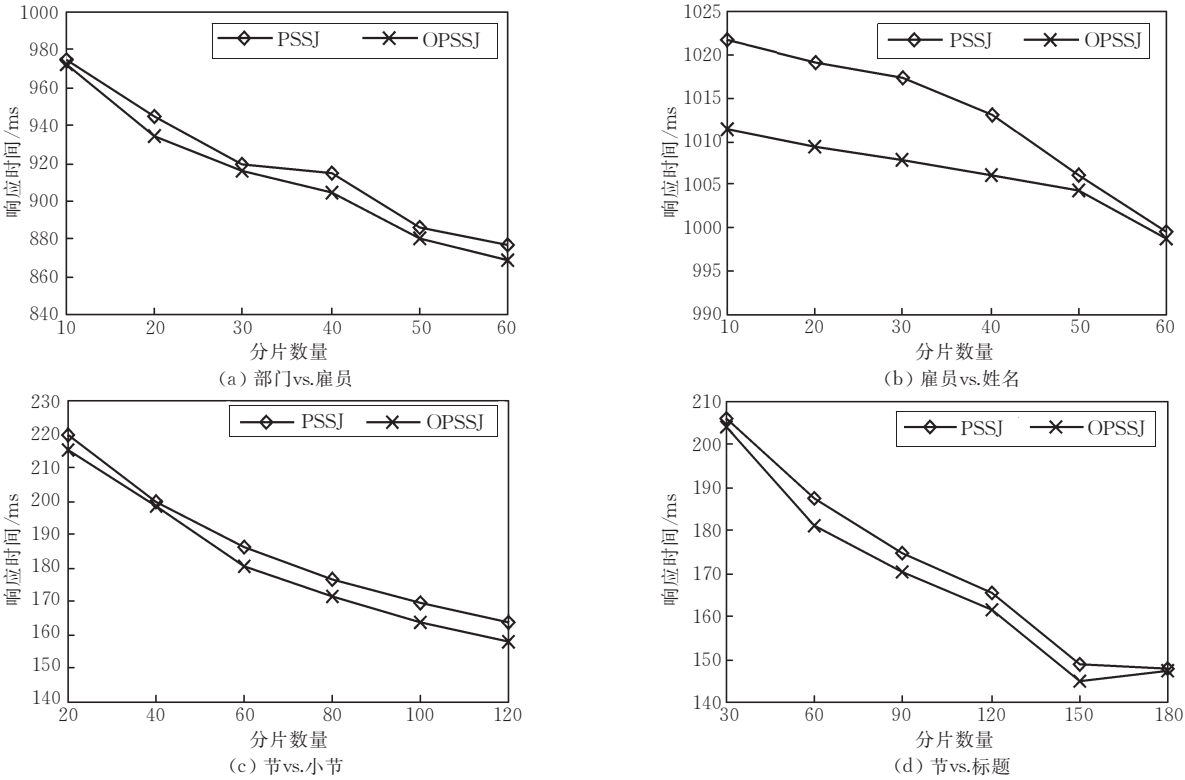


图 12 对片断数量指标的测试

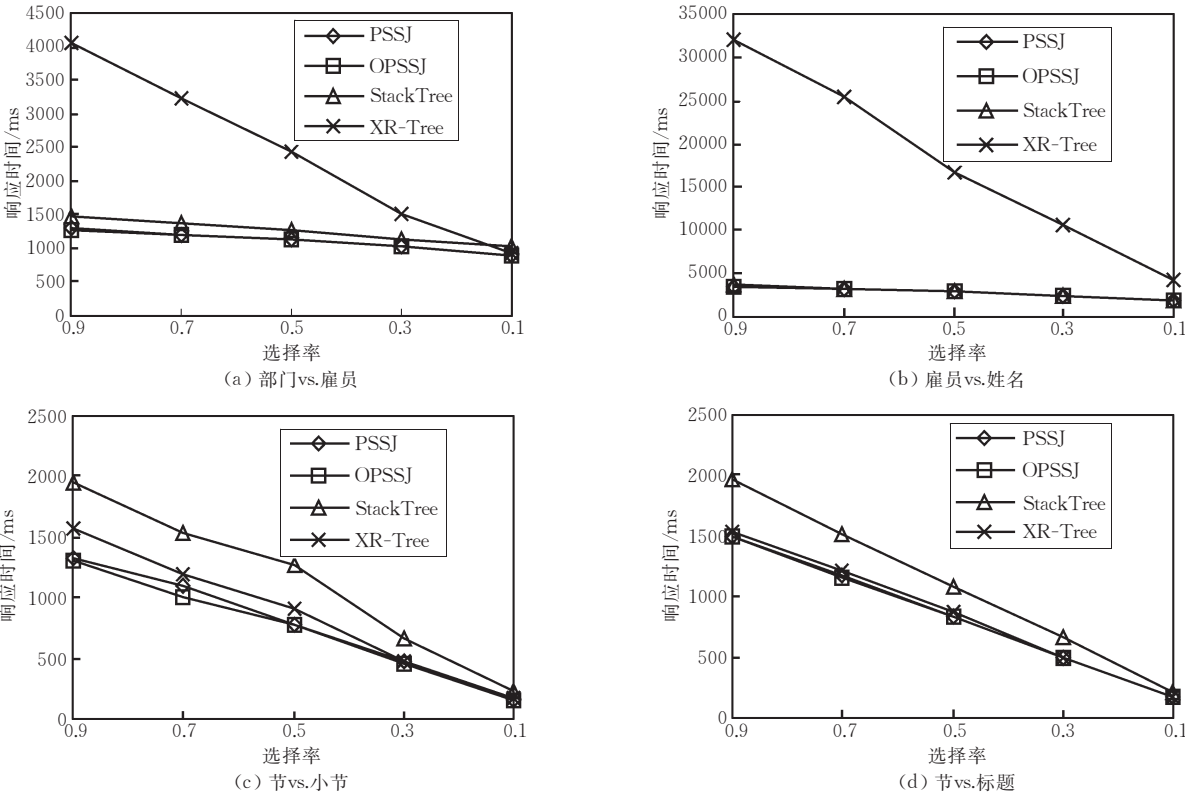


图 13 对连接选择率指标的测试

5.4 文档大小变化

在第三组实验中,我们保持连接选择率为 30% 和划分元素片断数量一定的情况下,通过改变测试

文档的大小来观察对各种不同算法性能的影响. 其中文档大小从 20MB~100MB 变化. 图 14 给出了测试性能曲线.

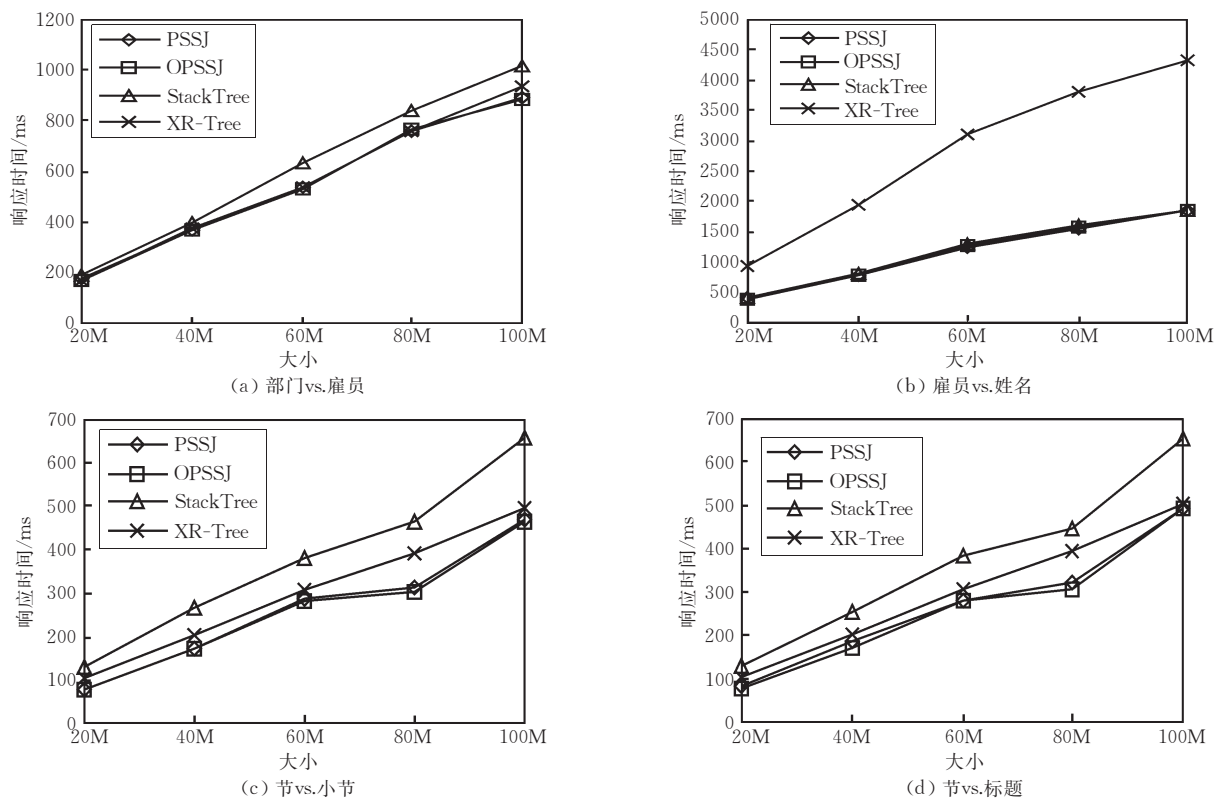


图 14 不同大小文档的测试

通过观察我们可以看出,在所有情况下,本文提出的 PSSJ 和 OPSSJ 算法的性能都要优于 XR-Tree 算法和 StackTree 算法的性能,但是图 14(b)是个例外,因为当祖先元素嵌套深,后代元素嵌套不深时,XR-Tree 算法的性能比 StackTree 算法的性能要差很多. 通过分析可以知道,这 4 种结构连接算法随着测试文档的变大都有比较好的性能,而且每种算法的响应时间也呈现出有序变化.

5.5 分片与工作量

在本节中,我们分析分片对结构连接工作量的影响. 表 3 显示了图 1 中文档在划分片断数量不同时工作量的变化.  $A_{output}$  是指不用做结构连接而直接输出连接结果的祖先元素个数,这样在做结构连接操作时我们就省略了这部分工作量.  $A_{join}$  是指需要结构连接的祖先元素个数. 在这里我们通过计算  $n_a \times n_d$  来衡量工作量的变化,其中,  $n_a$  是指参加结构连接操作的祖先元素个数,  $n_d$  是指参加结构连接操作的后代元素个数. 如表 3 所示,随着片断数量的增大,落在  $A_{output}$  中的祖先元素个数越来越多,这样相应参加连接操作的祖先元素就减少了,从而使总的工作量变小了. 例如,当片断数量为 1 时,连接操作

的工作量为 60;当片断数量为 2 时,连接操作的工作量为 25;当片断数量为 3 时,连接操作的工作量减小为 12;但是当片断数量为 4 时,连接操作的工作量仍为 12,因为这时候参加连接的祖先后代元素数量不再改变,所以在这种情况下,就没必要继续增大片断划分的数量了.

表 3 XML 例子文档中查询 A//D 的工作量			
划分数	$A_{output}$	$A_{join}$	工作负载
1	1	$6 \times 10$	60
2	2	$2 \times 5$	25
	1	$3 \times 5$	
3	3	$1 \times 3$	12
	2	$0 \times 4$	
	1	$3 \times 3$	
4	3	$1 \times 3$	12
	2	$0 \times 2$	
	2	$0 \times 2$	
	1	$3 \times 3$	

表 4 和表 5 显示了随着片断划分数量的变化,两条查询语句 section//title 和 department//employee 的结构连接工作量的变化情况. 从这两个表中可以看出,随着片断划分数量的增加,结构连接工作量越来越小,但是当片断数量达到某个值时,工作

量变化很小甚至不再变化.

表 4 查询 section//title 的工作量

划分数	$A_{output}$	工作负载
10	0	637009
20	0	376814
30	0	283415
40	2	231936
50	6	197166
60	13	167544
70	21	147767
80	29	131672

表 5 查询 department//employee 的工作量

划分数	$A_{output}$	工作负载
10	10	1406674614
20	29	704345956
30	51	470172574
40	77	352900632
50	107	282578829
60	140	235757887
70	171	202118390
80	202	177131250

5.6 代价维护的测试

在最后一组实验中,目的是测试 XR-tree 和本文提出的基于分片的空间结构连接算(PSSJ)的维护代价,本节主要通过测试两个算法在构建索引结构时所花费的时间来比较它们的维护代价的.图 15 给出了两个算法在文档大小变化时的维护代价.

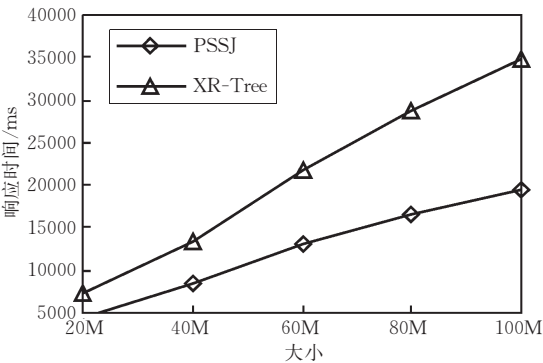


图 15 维护代价的测试

在这里我们仅仅给出了在执行查询语句  $Q_1$  时维护代价的曲线来说明问题,执行其它查询语句时,维护代价曲线有类似的趋势.从图 15 中的性能曲线走势可以看出,PSSJ 的维护代价要小于对 XR-tree 的维护代价,而且随着文档的增大,这种趋势越来越明显.

6 结束语

本文的主要贡献如下:

(1)我们把节点间的关系引申到片断之间的关

系,通过研究片断之间的关系我们得到 9 个特征,通过对这些特征的分析,最终得到 4 条性质.

(2)我们提出了一种基于分片的结构连接方法,即 P-JOIN().在以前的工作中,文献[3-4]仅仅把  $B^+$ -tree 和 XR-tree 作为索引,然后再利用 Stack-Tree 算法在连接过程中跳过一些不参加连接的祖先或后代元素.文献[17]虽然提出了基于区域划分的概念,但是主要是解决当输入元素集合不存在索引或者无序时,基于任务分解的思想来解决结构连接问题的,该文并没有进一步研究各区域之间的特性.然而在本文中,祖先和后代元素通过它们之间的位置关系被划分到不同的区域,根据区域间的特性,可以对一些不用做连接操作的区域就可以直接输出连接结果,其它部分利用本文发现的空间特性也能很快产生连接结果.

(3)我们提出了一种基于分片的结构连接方法和两种优化方法,另外,我们设计了一种新的数据结构作索引来快速查询分片结果.

(4)通过实验对各项指标进行测试,最终结果表明本文提出的基于分片的结构连接算法要优于以前提出的结构连接算法.

参 考 文 献

[1] Zhang C, Naughton J, DeWitt D, Luo Q, Lohman G. On supporting containment queries in relational database management systems//Timos S. Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2001: 425-436

[2] Al-Khalifa S, Jagadish H V, Koudas N, Patel J M, Srivastava D, Wu Y Q. Structural joins: A primitive for efficient XML query pattern matching//Agrawal R, Dittrich K, Ngu A H H. Proceedings of the 18th International Conference on Data Engineering. Los Alamitos: IEEE Press, 2002: 141-152

[3] Chien S Y, Vagena Z, Zhang D H, Tsotras V J, Zaniolo C. Efficient structural joins on indexed XML documents//Bernstein P A et al. Proceedings of the 28th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers, 2002: 263-274

[4] Jiang H F, Lu H J, Wang W, Ooi B C. XR-Tree: Indexing XML data for efficient structural joins//Dayal U, Ramaritham K, Vijayaraman T M. Proceedings of the 19th International Conference on Data Engineering. Los Alamitos: IEEE Press, 2003: 253-264

[5] Lu Jian-Hua, Wang Guo-Ren, Yu Ge. Optimizing path expression queries of XML data. Journal of Software, 2003, 14 (9): 1615-1620(in Chinese)



(吕建华, 王国仁, 于戈. XML 数据的路径表达式查询优化技术. 软件学报, 2003, 14(9):1615~1620)

- [6] Dietz P F. Maintaining order in a linked list//Proceedings of the 14th Annual ACM Symposium on Theory of Computing. New York: ACM Press, 1982: 122-127
- [7] Li Quan-Zhong, Moon Bongki. Partition based path join algorithms for XML data//Proceedings of the 14th International Conference DEXA. Czech Republic: Springer-Verlag Heidelberg Publishers, 2003: 160-170
- [8] Wang W, Jiang H F, Lu H J, Jeffrey X Y. PbiTree coding and efficient processing of containment join//Dayal U, Ramamritham K, Vijayaraman T M eds. Proceedings of the 19th International Conference on Data Engineering. Los Alamitos: IEEE Press, 2003: 391-402
- [9] Min Jun-Ki, Park Myung-Jae, Chung Chin-Wan. XPRESS: A queriable compression for XML data//Proceedings of the ACM SIGMOD Conference on Management of Data. New York: ACM Press, 2003: 122-133
- [10] Grust T. Accelerating XPath location steps//Proceedings of the ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2002: 109-120
- [11] Babb E. Implementing a relational database by means of specialized hardware//Proceedings of the ACM Transactions on Database System (TODS). New York: ACM Press, 1979: 1-29
- [12] Valduriez P, Gardarin G. Join and semi-join algorithms for a multiprocessor database machine//Proceedings of the ACM Transactions on Database System (TODS). New York: ACM Press, 1984: 133-161
- [13] Guttman A. R-Trees: A dynamic index structure for spatial searching//SIGMOD' 84 Proceedings of Annual Meeting. New York: ACM Press, 1984: 47-57
- [14] Beckmann N, Kriegel H-P, Schneider R, Seeger B. The R\*-tree: An efficient and robust access method for points and rectangles//Proceedings of the ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 1990: 322-331
- [15] Ciaccia P, Patella M, Zezula P. M-tree: An efficient access method for similarity search in metric spaces//Proceedings of the 23rd International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers, 1997: 426-435
- [16] Diaz A, Lovell D. XML generator. <http://www.alphaworks.ibm.com/tech/xmlgenerator>, Sept. 1999
- [17] Wang Jing, Meng Xiao-Feng, Wang Shan. Structural join of XML based on range partitioning. Journal of Software, 2004, 15(5): 720-729(in Chinese)  
(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)



**WANG Guo-Ren**, born in 1966, professor, Ph.D. supervisor. His research interests include XML data management, query processing and optimization, bioinformatics, high-dimensional indexing, parallel database systems, and P2P data management.

**QIAO Bai-You**, born in 1972, Ph.D.. His research interests include P2P data management.

**HAN Dong-Hong**, born in 1968, Ph.D. Her research interests include data management techniques over data streams.

**WANG Bin**, born in 1971, Ph.D. His research interests include P2P data management.

## Background

XML is emerging as the dominant standard for representing information and exchanging data over the Internet, and a lot of research results have been published recently, including query languages, storage management, indexing, and query processing and optimization. Since structural join is the core part of XML queries, it has a significant impact on the performance of XML queries and several structural join algorithms have been proposed such as Stack-Tree and XR-tree.

These algorithms mainly consider the relationships between two element codes to adjust the structural relationship between these two elements. In this paper, the authors first partition the element encoding space into several partitions, and extend the element structural relationship to the partition structural relationship, then exploit the partition structural relationships to speed up the performance of the structural join operations.