基于区间编码的 XML 索引结构的有效结构连接

万常选[1),2)] 刘云生[2)] 徐升华[1)] 刘喜平[1)] 林大海[1)]

[1)](江西财经大学信息管理学院 南昌 330013)

[2)](华中科技大学计算机科学与技术学院 武汉 430074)

摘 要 该文给出了一个 XML 树数据模型的形式化定义.将编码方案、逆序列表和路径索引的思想相结合,提出了一种改进的 XML 数据的索引结构;给出了两个实现双亲/孩子关系和拥有关系的结构连接算法,它们最多只需要对参与连接的两个列表分别进行一次扫描,并且能够根据双亲结构信息等利用 B^+-树索引尽可能多地跳过不需要参与连接的元素结点.实验结果表明,该文给出的基于 XML 索引结构实现双亲/孩子关系和拥有关系的结构连接算法是高效的、健壮的.

关键词 XML 数据模型;XML 索引结构;区间编码;结构连接;拥有关系

中图法分类号 TP311

Indexing XML Data Based on Region Coding for Efficient Processing of Structural Joins

WAN Chang-Xuan[1),2)] LIU Yun-Sheng[2)] XU Sheng-Hua[1)] LIU Xi-Ping[1)] LIN Da-Hai[1)]

[1)](*School of Information Technology, Jiangxi University of Finance & Economics, Nanchang* 330013)

[2)](*School of Computer Science & Technology, Huazhong University of Science & Technology, Wuhan* 430074)

Abstract Firstly, this paper gives a formal definition of XML tree data model in context environment. Secondly, it proposes an improved index structure to retrieve XML data based on the idea of the numbering scheme, the inverted list and the path index. This index structure can process XQuery path expressions via two methods: Stepwise structural join method and path method. Furthermore, it enables us to quickly determine ancestor/descendant relationships and parent/child relationships between any pair of nodes in the XML trees, thus can efficiently process containment joins in XQuery path expression queries and keyword searches. Finally, this paper presents two algorithms for processing structural joins of parent/child relationship and holding relationship, which lead to optimal join performance by avoiding scanning on each list joined repeatedly and omitting the examination of elements, which do not participate in the join, via B^+-tree index based on parent's information and etc. Experimental results have showed that the structural join algorithms for processing parent/child relationships and holding relationships based on the proposed XML index structure are effective, efficient and robust.

Keywords XML data model; XML index structure; region coding; structural join; holding relationship

---

收稿日期:2002-10-20;修改稿收到日期:2004-01-29.本课题得到江西省自然科学基金项目(0411009)和江西省教育厅科技项目(赣财教[2003]73 号)资助.**万常选**,男,1962 年生,博士,教授,主要研究方向为 XML 数据库、Web 信息管理、数据挖掘、电子商务. E-mail: wanchangxuan@263.net. **刘云生**,男,1940 年生,教授,博士生导师,主要研究方向为现代数据库理论与技术及其集成实现、数据库与信息系统开发. **徐升华**,女,1952 年生,教授,博士生导师,主要研究方向为信息管理与信息系统. **刘喜平**,男,1981 年生,硕士研究生,主要研究方向为 XML 数据库、Web 信息管理. **林大海**,男,1980 年生,助教,主要研究方向为 XML 数据库.

1 引 言

XML 是 Web 上进行信息表示与交换的一个层次数据格式[1]. 随着大量 XML 数据的出现,如何有效地索引、存储和查询这些 XML 数据就成为目前值得研究的一个重要课题.

XML 查询典型地包括:在元素内容上的选择,即通过限定在元素内容或属性值上的取值而进行的选择查询,称为值查询;通过路径表达式,对文档中标记的元素之间的结构关系进行查询,称为结构查询. 元素之间的结构关系包括祖先/后裔(ancestor/descendant)关系、双亲/孩子(parent/child)关系、之前/之后(preceding/following)关系、左兄弟/右兄弟(preceding-sibling/following-sibling)关系等. 其中,祖先/后裔、双亲/孩子关系统称为包含关系. 利用路径表达式,用户能够导航 XML 文档树中的任意长度的路径. 但是,一个单一路径步的查询,如 *book/author* 或 *section//figure*,将分别是仅仅查找那些作为 *book* 元素的孩子 *author* 元素(即 *book* 与 *author* 元素之间是双亲/孩子关系)或是 *section* 元素的后裔 *figure* 元素(即 *section* 与 *figure* 元素之间是祖先/后裔关系). 因此,XML 文档的结构查询将导致结构连接(structural join)[2~8]的计算. 在文献[7]中称为 EE-连接和 EA-连接,而在文献[8]中称为"包含查询". 与 XML 文档树的遍历操作相比较,这种按祖先/后裔关系的结构连接操作,对于搜索路径很长或具体路径不清楚的 XML 文档的查询和搜索来说是特别有效的.

为了有效地支持 XML 查询,特别是路径表达式查询,对 XML 数据的各种编码方案(encoding scheme)和索引技术被提出[7~15]. 对于 XML 路径表达式查询,一种实现方法是建立 XML 文档的路径索引,并通过路径索引来加速 XML 路径表达式查询的计算;另一种方法是对 XML 文档树中的结点进行编码,并将 XML 路径表达式的计算转化为结构连接的计算.

文献[7]对利用编码方案(numbering scheme)来有效地处理 XML 的正则路径表达式(Regular Path Expression,RPE)查询进行了研究,以实现快速确定在 XML 数据层次结构中任意结点对之间的祖先/后裔关系. 文献[8]利用关系数据库来存储 XML 文档的逆序列表(inverted list),以支持关键字搜索,主要是对 XML 的 RPE 查询和关键字搜索中遇到的一类包含连接操作在关系数据库系统中的实现性能进行了分析. 文献[13]提出了一个基于路径索引的 XML 文档的关系存储模式,在该关系模式下,一个 XQuery 简单绝对路径表达式首先被转化为在路径表 Path 的 *pathExp* 列上的串配匹操作,然后将表 Path 经串配匹的结果与元素表、属性表或值表按 *pathID* 列进行等值连接操作. 单纯的路径法有很大的局限性,它只能处理简单绝对路径表达式查询.

本文将编码方案[7]、逆序列表[8]和路径索引[13]的思想相结合,基于区间编码提出了一种改进的 XML 数据的索引结构. 它能够快速地判断 XML 文档树中任意结点对之间的祖先/后裔关系和双亲/孩子关系,有效地支持结构连接的计算,有效地支持 XQuery 路径表达式查询和关键字搜索. 然后,基于区间编码的 XML 索引结构,分别提出了实现双亲/孩子关系和拥有关系结构连接的高效算法,它们最多只需要对参与连接的两个列表分别进行一次扫描,并且能够根据双亲结构信息等利用 B^+-树索引尽可能多地跳过不需要参与连接的元素结点.

2 XML 数据模型

一个 XML 文档是由一些嵌套的元素结构所组成. 图 1 表示的是一个包含有关出版信息的 XML 文档. 其中,editor 元素有一个 ID 类型的属性 id(值为"205");article 元素有一个 IDREFs 类型的属性 editorID,该属性的值是"205 206",并通过它引用 id 属性的值为"205"和"206"的 editor 元素结点.

```
〈pub〉
  〈book year="2001"〉
    〈title〉Database System Concepts〈/title〉
    〈price〉38.50〈/price〉
    〈author id="102"〉
      〈name〉Kaily Jone〈/name〉
    〈/author〉
    〈author id="103"〉
      〈name〉Silen Smith Jone〈/name〉
    〈/author〉
  〈/book〉
  〈article editorID="205 206"〉
    〈title〉A Query Language for XML〈/title〉
    〈author id="104"〉
      〈name〉Kaily Jone〈/name〉
    〈/author〉
  〈/article〉
  〈editor id="205"〉
    〈name〉A. Deutsch〈/name〉
  〈/editor〉
〈/pub〉
```

图 1 一个 XML 文档实例

为节省篇幅，图 1 中给出的 XML 文档是不完整的.

一个 XML 文档能够被建模为一个有序的、边标记的树，称为查询数据模型[1]，这里我们简称为树模型. 图 2 表示的是图 1 所示 XML 文档的一个树模型，为了简化，该图中只标出了 pub 根元素结点所嵌套的一个 article 子元素结点的信息. 在树模型中，结点表示文档元素、属性或文本数据等，边表示元素-子元素(或双亲/孩子关系).

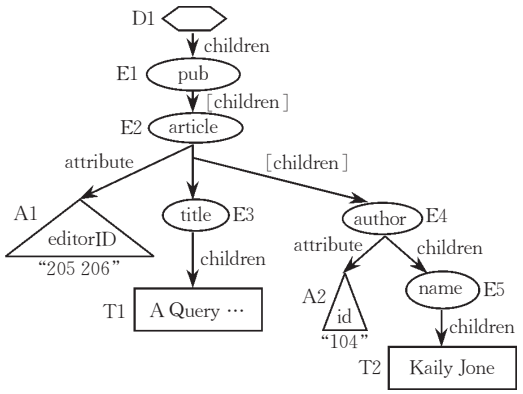

图 2　对于图 1 所示 XML 文档的树模型实例

结点是树模型中最基本的概念. 在树模型中共有七种类型的结点，分别是文档结点、元素结点、属性结点、文本结点、名字空间结点、处理指令结点和注释结点. 一个 XML 文档有一个唯一的文档结点，称为根结点. 对于一个格式规范的(well-formed) XML 文档而言，文档结点直接指向一个唯一的元素结点，称为根元素结点，但对于查询数据模型而言，并没有唯一性的要求. 元素结点依次指向其它的元素结点、属性结点、文本结点等.

下面，我们针对本文的上下文给出一个 XML 文档树模型的形式化定义. 这里，我们忽略了名字空间结点、处理指令结点和注释结点.

定义 1.　一个 XML 文档是一个有序的、边标记的树，记为 $T=\langle V,E,r,label,rank\rangle$. 其中，$V$ 是所有结点的集合，$E\subseteq V\times V$ 是边的集合，$r\in V$ 是文档根结点，并且

(1) $V=r\cup V_E\cup V_A\cup V_T$，其中 V_E，V_A 和 V_T 分别表示元素结点、属性结点和文本结点的集合.

(2) $E=E_E\cup E_A\cup E_T$，其中 $E_E\subseteq V_E\times V_E$ 是元素边的集合，$E_A\subseteq V_E\times V_A$ 是属性边的集合，$E_T\subseteq V_E\times V_T$ 是文本边的集合. 这里，元素边、属性边和文本边分别是指从元素结点指向子元素结点的边、指向属性结点的边和指向文本结点的边. 元素边与文本边又统称为孩子边.

(3) $label=label_E\cup label_A\cup label_T$，其中

① 函数 $label_E$：$V_E\rightarrow\langle string,string,string\rangle$，给每个元素结点赋予一个三元字符串组，分别表示该元素结点的对象标识、元素标记名和结点类型. 结点类型的值是“EE”，“ET”，“EM”或“EN”，它们分别表示的为内容是子元素、内容是文本、内容是子元素和文本混合或内容为空的元素结点.

② 函数 $label_A$：$V_A\rightarrow\langle string,string,string,string\rangle$，给每个属性结点赋予一个四元字符串组，分别表示该属性结点的对象标识、属性名、属性值(字符串值)和属性值的类型.

③ 函数 $label_T$：$V_T\rightarrow\langle string,string,string\rangle$，给每个文本结点赋予一个三元字符串组，分别表示该文本结点的对象标识、文本结点的内容(字符串值)和内容的类型.

(4) 函数 $rank$：$V\rightarrow integer$，给每一个结点赋予一个整数，表示该结点的文档序号. 结点的文档序号在一个文档内唯一，它反映在先序遍历文档树时该结点的出现次序. 对于一个元素结点而言，这里规定它所拥有的属性结点的文档序号必须小于它的内容结点(子元素结点和文本结点)的文档序号. 由于元素结点中所包含的各个属性结点之间不区分次序，因此，对于某元素结点所包含的各属性结点的文档序号的先后无关紧要.

3　XML 数据的索引结构

3.1　XML 数据的区间编码

各种文献中提出的编码方案主要是针对如下特点来设计的：① 被编码数据的结构，例如，树、图等；② 支持祖先/后裔、双亲/孩子等关系的结构查询；③ 编码算法的复杂度；④ 编码的最大长度或平均长度；⑤ 编码后的查询执行时间；⑥ 插入操作导致的重新编码代价.

对于区间编码方案，树 T 中的每一个结点被赋予一个区间编码 $[begin,end]$，满足：一个结点的区间编码包含它的后裔结点的区间编码.

第一个区间编码方案是在文献[16]中被提出的(称为 Dietz 编码)，树 T 中的每一个结点被赋予一个具有先序遍历序号和后序遍历序号的二元组 $\langle pre,post\rangle$. 由于树 T 中的一个祖先结点 u 在先序遍历(后序遍历)中必然出现在它的后裔结点 v 之前(之后)，因此，结点 u 和 v 是祖先/后裔关系，当且仅当 $pre(u)<pre(v)$ 且 $post(v)<post(u)$. 因此，树 T 中的任意两个结点之间的祖先/后裔关系的检测(即包含检测)能够在常数时间内被计算(即两次比较运算). 对于该编码方案，pre 或 $post$ 均可以作为

结点的唯一标识.

文献[7,8,13～15]中提出的 XML 数据的区间编码方案是 Dietz 编码的推广. 文献[7]提出的 XML 数据的区间编码方案(称为 Li-Moon 编码)是:XML 文档树 T 中的每一个结点被赋予一个二元组$\langle order, size\rangle$,其中,$order$ 为结点的扩展先序遍历序号,它的取值是非连续的,为结点的插入预留序号空间;$size$ 为结点的后裔范围. 因此,树 T 中的任意两个结点 u 和 v 是祖先/后裔关系,当且仅当$order(u)<order(v)$且 $order(v)+size(v)<=order(u)+size(u)$,即祖先结点 u 的区间编码$[order(u),order(u)+size(u)]$包含后裔结点 v 的区间编码$[order(v),order(v)+size(v)]$. 这种祖先/后裔关系的判别条件可以进一步改写为 $order(u)<order(v)$且 $order(v)<=order(u)+size(u)$. 另外,树 T 中的每一个结点再被赋予一个值 $depth$,表示该结点在树中所处的层数,这样,树 T 中的任意两个结点 u 和 v 是双亲/孩子关系,当且仅当 $order(u)<order(v)$且$order(v)<=order(u)+size(u)$ 且 $depth(u)=depth(v)-1$. Li-Moon 编码与 Dietz 编码相比,能够更好地支持文档的修改. 对于该编码方案,$order$ 作为结点的唯一标识.

文献[8]提出的 XML 数据的区间编码方案(称为 Zhang 编码)是:XML 文档树中的每一个结点被赋予一个二元组$\langle begin, end\rangle$. 对树 T 的所有结点进行先序遍历,每一个结点在遍历时分别被访问两次并产生两个序号,一次是在遍历该结点的所有后裔结点之前访问该结点,并产生该结点的序号 $begin$;另一次是在遍历完该结点的所有后裔结点后再一次访问该结点,并产生该结点的另一个序号 end. 因此,树 T 中的任意两个结点 u 和 v 是祖先/后裔关系,当且仅当 $begin(u)<begin(v)$且 $end(v)<end(u)$,即祖先结点 u 的区间编码$[begin(u),end(u)]$包含后裔结点 v 的区间编码$[begin(v),end(v)]$. 这种祖先/后裔关系的判别条件可以进一步改写为 $begin(u)<begin(v)$且 $begin(v)<end(u)$. 另外,树 T 中的每一个结点也被再赋予一个值 $depth$,表示该结点在树中所处的层数. 对于该编码方案,$begin$ 作为结点的唯一标识.

3.2 扩展先序列表

为了有效地实现对文档树中任意两个结点对之间的祖先/后裔关系和双亲/孩子关系的检测,以加速 RPE 的计算,需同时实现按关键字搜索 XML 文档. 我们将编码方案[7]和逆序列表[8]的思想相结合,提出了一种改进的 XML 数据的索引结构,称之为扩展先序列表(extended preorder list). 其基本思想是:对 XML 文档树中的所有元素结点、属性结点以及文本结点和属性值中的关键字进行先序遍历(即深度优先遍历),产生这些结点和关键字的先序遍历序号,并将这些结点和关键字的先序遍历序号按升序进行列表. 具体的处理方法是:

(1)元素索引列表. 对于所有具有相同元素标记名 Tag 的元素结点建立一个元素索引列表(即扩展先序列表)Elem_Tag,该索引列表中的每一个记录是标识该结点的一个六元组$\langle docID, order, maxOrder, depth, parentOrder, parentMax\rangle$. 其中:$docID$ 是该结点所在文档的文档标识;$order$ 是该结点的序号,它是一个文档树中所包含的所有元素结点、属性结点以及文本结点和属性值中的关键字按先序遍历所产生的先序遍历序号;$maxOrder$ 是以该结点为根的子树中所包含的所有元素结点、属性结点以及文本结点和属性值中关键字的最大 $order$;$depth$ 是该结点在文档树中所处的层数,以反映祖先/后裔关系中的双亲/孩子关系;$parentOrder$ 和 $parentMax$ 分别表示该结点的双亲结点的 $order$ 和 $maxOrder$,以便有效地实现双亲/孩子关系和拥有关系的结构连接、兄弟关系的判断以及双亲操作等.

(2)属性索引列表. 对于所有具有相同属性名 $Name$ 的属性结点建立一个属性索引列表$Attr_Name$,该索引列表中的每一个记录是标识该结点的一个六元组$\langle docID, order, maxOrder, depth, parentOrder, parentMax\rangle$.

对于文档森林中的一个元素或属性结点 n,记 $region(n)$表示区间$[order(n),maxOrder(n)]$. 因此,对于文档森林中的任意两个元素或属性结点 x 和 y,x 是 y 的一个祖先,当且仅当 $docID(x)=docID(y)$ and $order(x)<order(y)$ and $maxOrder(y)<=maxOrder(x)$,即区间 $region(x)$包含区间 $region(y)$. 由于对于任意一个结点 y,一定有 $order(y)<=maxOrder(y)$,因此,上述条件可以改写为 $docID(x)=docID(y)$ and $order(x)<order(y)$ and $order(y)<=maxOrder(x)$,记为 $region(x)$ contains $region(y)$. 如果 x 是 y 的双亲,则还必须同时满足 $depth(x)+1=depth(y)$,此时记为 $region(x)$ directly contains $region(y)$.

有了 $parentOrder$,在判断结点 x 是否是结点 y 的双亲(即结点 x 与结点 y 之间是否满足双亲/孩子关系)时,还可以利用 $order(x)=parentOrder(y)$ 条件,因此,$depth$ 信息将成为冗余;此外,$parentOrder$ 也可以用来判断元素结点之间的兄弟关

系，例如，x 和 y 是两个兄弟元素结点，当且仅当 $parentOrder(x) = parentOrder(y)$. 增加 *parentMax* 是为了加快结构连接的计算，第 4 节将对它进行详细阐述. 为了能够实现在 XML 文档树的指定层次范围内的包含查询，这里仍然保留 *depth* 信息.

为了便于文档的修改，结点和关键字的 *order* 取值并不是连续的，要为可能的插入和修改留有序号的空间. 因此，我们称之为“扩展”先序列表. 同时，*maxOrder* 的取值也不是以该结点为根的子树中所包含的所有元素结点、属性结点以及文本结点和属性值中关键字最大 *order*，而是取值为该结点之后(即满足 *following* 轴的结点域中)的第一个结点的 *order* 值减 1. 这里的 *order* 既可以被看作是定义 1 中的文档序号，以反映 XML 文档中各结点之间的有序性；又可以被看作是定义 1 中的结点对象的唯一标识.

元素索引列表 Elem_*Tag* 和属性索引列表 Attr_*Name* 的作用是有效地实现对文档树中任意结点对之间的祖先/后裔关系(或双亲/孩子关系)的检测，以加速 RPE 的计算. 典型地，XML 文档森林中的所有元素结点和属性结点的索引信息可以集中组织在一个列表中. 但是，一方面这样的一个列表一般来说是巨大的，另一方面 RPE 查询的计算经常要对这些列表进行选择操作、按祖先/后裔关系的连接操作等，因此为了提高处理性能，这里我们按文档中出现的元素标记“*Tag*”或属性名“*Name*”分别组成多个元素或属性索引列表.

(3) 关键字索引列表. 对于所有属性值和文本结点内容中出现的每一个相同的关键字 *Keyword* 建立一个关键字索引列表 Word_*Keyword*，该列表中的每一个记录是标识该关键字的一个三元组 $\langle docID, order, depth\rangle$. 其中，*docID* 是该关键字所在文档的文档标识；*order* 是该关键字在文档中所处位置的先序遍历序号；*depth* 是该关键字在文档树中所处的层数，以反映它是直接属于哪一个属性结点或双亲元素结点中.

对于文档森林中的每一个关键字 w，它有一个扩展先序遍历序号 *order*，并没有 *maxOrder*，设 $[order(w), order(w)]$ 表示它所对应的区间，记为 $region(w)$. 因此，对于文档森林中任意一个元素或属性结点 x 以及一个关键字 w，在以结点 x 为根的子树中(或 x 的孩子元素结点的内容和属性值中)包含关键字 w，当且仅当 $region(x)$ contains (or directly contains) $region(w)$.

关键字索引列表 Word_*Keyword* 的作用是有效地实现按关键字搜索. 同样，为了提高处理性能，我们已经按相同的关键字 *Keyword* 分别组织一个关键字列表.

图 4 反映的是图 3 所示 XML 文档片断的扩展先序列表结果(仅列示了 *order*、*maxOrder* 和 *depth* 信息). 这里，已经假设(根据 XML 文档的模式信息和统计信息来确定)：每一个 *title* 元素的内容一般有 20 个关键字，每一个 *name* 元素的内容或 *editor* 属性的值一般有 4 个关键字，且实例文档中不足的关键字的序号空间需要预留；对于每个 *book* 元素，可选的 *press* 子元素和它的内容需要预留 11 个序号，1 个或多个 *author* 子元素结点需要预留 4 个；对于每个 *author* 元素，可选的 *contact* 子元素和它的内容需要预留 21 个序号.

```
〈book editor="Tom Smith"〉
  〈title〉Database System Concepts〈/title〉
  〈author id="102"〉
    〈name〉Kaily Jone〈/name〉
  〈/author〉
  〈author id="103"〉
    〈name〉Silen Smith Jone〈/name〉
  〈/author〉
〈/book〉
```

```
〈!ELEMENT book(title, press?, author+)〉
〈!ATTLIST book editor CDATA #IMPLIED〉
〈!ELEMENT author(name, contact?)〉
〈!ATTLIST author id ID #REQUIRED〉
〈!ELEMENT title (#PCDATA)〉
〈!ELEMENT press (#PCDATA)〉
〈!ELEMENT name (#PCDATA)〉
〈!ELEMENT contact (#PCDATA)〉
```

图 3　XML 文档片断和它的 DTD 片断

book	(1,150,0)
editor	(2,6,1)
title	(7,27,1)
author	(39,66,1)
id	(40,40,2)
name	(41,45,2)
author	(67,94,1)
id	(68,68,2)
name	(69,73,2)

Tom	(3,2)
Smith	(4,2)
Database	(8,2)
System	(9,2)
Concepts	(10,2)
Kaily	(42,3)
Jone	(43,3)
Silen	(70,3)
Smith	(71,3)
Jone	(72,3)

图 4　XML 文档片断的扩展先序列表结果

3.3　索引结构与关系存储模式

图 5 所示的是我们根据扩展先序列表的思想而设计的 XML 数据的索引结构及关系存储模式.

Document(***docID***, *URL*, …)
Name(***nameID***, *name*, …)
Path(***pathID***, *pathExp*, …)
Elem_*Tag*(***docID***, ***order***, *maxOrder*, *depth*, *attrID*, *nodeType*, *pathID*, *parentOrder*, *parentMax*, *ssIndex*)
Attr_*Name*(***docID***, ***order***, *maxOrder*, *depth*, *pathID*, *parentOrder*, *parentMax*)
Word_*Keyword*(***docID***, ***order***, *depth*)
Structure(***docID***, ***order***, *maxOrder*, *depth*, *nodeType*, *nameID*, *parentOrder*, *parentMax*)
Value(***docID***, ***order***, ***afterOrder***, *value*, *valueType*, *pathID*)

图 5 XML 数据的索引结构及关系存储模式

其中：

(1) 各个索引列表或值列表的码属性被用粗体表示. 假设所有列表都按码属性建立聚集索引(即主索引),以加快查找的执行速度.

(2) 对于元素索引列表 Elem_*Tag*,*attrID* 用来存储元素结点的 ID 类型属性值,这样安排的目的是为了有效地实现按 ID 属性的精确定位,如有效地实现解除参照(dereference)操作"=>"等;*nodeType* 用来存储元素结点的类型,它的取值是"EE","ET","EM"或"EN",分别表示内容是子元素、内容是文本、内容是子元素和文本混合或内容为空的元素结点;*ssIndex* 用来存储该元素结点在它的同名兄弟结点中所处的位置序号,以便有效地实现按位置查询同名兄弟的操作.

(3) 在 XML 数据的索引结构中,除了元素索引列表、属性索引列表和关键字索引列表之外,进一步还要建立下面列表：

① 路径索引列表 Path. 用来存储 XML 文档中每一条从根标记开始到另一个元素标记或属性名的路径表达式 *pathExp* 和它的标识 *pathID*. 这种路径表达式是形如"$/name_1/name_2/\cdots/name_k$"或"$/name_1/name_2/\cdots/@name_k$"的字符串. 为了实现对一个简单绝对路径表达式进行串配匹操作,它的存储形式用"#/"来代替"/",如"$\#/name_1\#/name_2\#/\cdots\#/name_k$". 并假设属性结点的值或文本结点的内容的路径表达式与对应的属性结点或双亲元素结点的路径表达式相同.

② 文档索引列表 Document. 每一个记录与一个 XML 文档相对应,用来存储 *docID*(文档标识)、*URL*(文档所在位置的 URL)以及其它一些与该文档相关的信息.

③ 标记名索引列表 Name. 每一个记录与 XML 文档中的一个元素标记或属性名相对应,用来存储 *nameID*(名字标识)、*name*(标记名或属性名)以及其它一些与该名字相关的信息.

④ 结构索引列表 Structure. 用来存储所有元素结点和属性结点的扩展先序列表,即文档的总索引. 其中,*nameID* 存储的是标记名或属性名的标识,通过它能够从 Name 列表中获得标记名或属性名;*nodeType* 存储的是结点类型,它除了上述已说明的元素结点类型的取值之外,再加上一个取值"A",表示结点的类型是属性结点. 建立该索引列表的目的是有效地实现 XML 文档的结构查询以及查询结果的文档片断重构. 例如,对于一个给定的元素,查找它的所有孩子、后裔或属性结点等.

⑤ 值列表 Value. 用来存储 XML 文档的内容,包括所有属性结点的值和文本结点的内容. 在我们的存储结构中,所有属性值和文本内容都以字符串的形式进行存储,但查询经常需要转换字符串到其它的带有更多语义信息的数据类型. 这里,*valueType* 就是用来说明属性或文本内容的数据类型. *order* 存储属性值或文本结点的内容所对应的属性结点或双亲元素结点的 *order*. 对于"EM"类型的元素结点而言,一个序号为 *order* 的元素结点可能包含多个文本内容串,但每一个文本内容串必然被安排在本元素结点或它的某个子元素结点之后出现,因此,*afterOrder* 用来存储这种用于标识文本内容串所在位置的元素结点序号.

4 结构连接算法

4.1 背景及相关工作

XQuery 的核心是 XPath,XQuery 查询的核心是路径表达式查询. 通常,一个复杂的 XQuery 路径表达式查询能够被分解为一些如下子表达式的组合：

(1) 由一个单一元素(或属性)构成的表达式,如 *author*,*@year*;

(2) 由二个元素或一个元素与一个属性(其中的元素或属性可以通过通配符来表示,下同)构成的满足包含关系的表达式,如 *chapter*//*title*,*book*/*child*::*,*book*/*@isbn*,*book*/*attribute*::*;

(3) 由二个元素或一个元素与一个属性构成的满足拥有关系的表达式,如 *book*[*chapter*],*book*[*@year*],*book*[*descendant*::*section*],*book*[*child*::*]. 拥有

关系是一种特殊的包含关系；

(4) 由一个元素(或属性)与一个搜索关键字构成的表达式，如 *contains*(*title*，“Web”)、*contains*(*descendant-or-self*::*，“XML”)，它实际上是元素(或属性)列表与关键字列表之间的满足拥有关系的结构连接；

(5) 由二个其它子表达式的并构成的表达式，如 *figure* union *table*.

对于由一个单一元素或单一属性组成的表达式(即第 1 种子表达式)，直接通过存取 Elem_*Tag* 或 Attr_*Name* 列表就能被处理；对于由两个子表达式的并构成的表达式(即第 5 种子表达式)，直接通过合并两个按码属性{*docID*，*order*}有序的基本子表达式的中间结果也能被处理. 对于其它三种类型的子表达式(即第 2～4 种子表达式)，它们需要对两个列表(或中间结果)按包含关系或拥有关系进行结构连接操作.

包含关系的返回结果通常是满足包含关系的后裔(或孩子)结点的有序序列；而拥有关系的返回结果却是满足拥有关系的祖先(或双亲)结点的有序序列.

因此，基于本文提出的 XML 数据的索引结构，一个路径表达式查询可以有两种计算策略，一种是路径法，另一种是逐步结构连接法. 路径法只能处理简单绝对路径表达式查询[13]，例如，查询//*book*/*chapter*//*title* 采用路径法进行计算时，首先在路径索引 Path 的 *pathExp* 属性上进行字符串匹配操作，然后将匹配结果与 Elem_*title* 列表按 *pathID* 属性进行等值连接操作. 而逐步结构连接法能够处理任意路径表达式查询，例如，查询//*book*[.//*name*＝“Kaily Jone”]/*title*，它的一个执行计划如图 6 所示. 对于该执行计划，虽然需要进行三次连接操作，但是，如果采用结构连接算法，整个计算都可以采用流水线技术，它的 I/O 代价除索引代价外，最多只需要对 Elem_*book*，Elem_*name*，Elem_*title* 和 Value 列表分别扫描一次(通过使用索引，实际执行代价一般会更低).

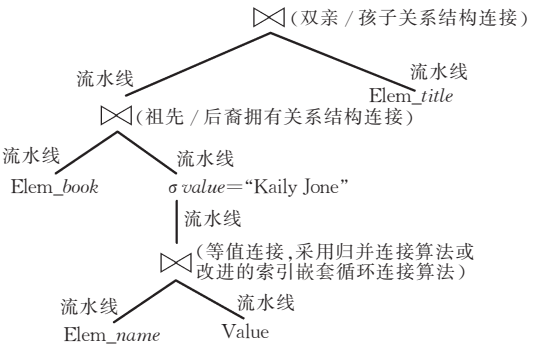

图 6　XML 路径表达式查询的执行计划

基于传统的关系数据库系统或原生的 XML 数据库系统，已经提出了各种结构连接的实现技术. 例如，文献[8]利用多谓词提出了一个改进的合并连接算法，称为 MPMGJN(Multi-Predicates Merge Join)算法. 然而，该算法在处理祖先列表中存在许多同名嵌套元素的祖先/后裔关系的结构连接，或双亲/孩子关系的结构连接时，会对被连接的后裔列表执行许多次重复的扫描，因此性能较差. 类似地，文献[7]中提出的 EE-连接算法也会产生同样的问题. 在 MPMGJN 算法的基础上，文献[6]给出了索引改进合并连接算法 IIMGJN. IIMGJN 算法采用了索引定位技术、短路技术和预侦技术来避免 MPMGJN 算法中不必要的扫描和搜索，从而减少连接代价.

在文献[2～5]中提出的结构连接算法代表了当前的研究水平. 其中，文献[2]中提出的 Stack-Tree-Desc 算法，假设祖先列表和后裔列表都是按 *order* 有序存储，则通过维护一个祖先元素结点栈(该栈用来存放后来可能需要连接的祖先元素结点)，仅仅需要对祖先列表和后裔列表分别扫描一次就可以实现祖先/后裔关系的结构连接. 如果在祖先列表和后裔列表上存在索引的话，则还可以通过索引跳过许多可以事先判断并不参与连接的元素结点，即并不需要对它们进行顺序扫描. 文献[3]中提出的 Anc-Des-B＋算法利用 B^+-树索引实现了这种技术. 文献[4]对通过选择谓词在多个元素上描述的小枝模式(twig pattern)包含关系的结构连接问题进行了研究，对一个 XML 路径表达式中包含的多个祖先/后裔关系结构连接的计算进行整体考虑，提出了一个整体小枝连接(holistic twig join)算法 TwigStack. 假设所有的元素列表都是按结点的先序遍历序号有序存储，通过维护多个祖先元素结点栈(该栈用来存放后来可能需要连接的祖先元素结点)，TwigStack 算法仅仅需要对所有参与连接的元素列表分别顺序扫描一次就可以实现整个 XML 路径表达式的计算. 与分步结构连接的计算策略相比，TwigStack 算法减少了对中间连接结果的多次存取，达到了最小的 I/O 代价. 文献[5]对小枝模式包含关系的结构连接问题进行了进一步的研究，提出了一个一般的整体小枝连接算法 TSGeneric＋，该算法的主要特点是通过索引跳过许多可以事先判断并不参与祖先/后裔连接的元素结点. 然而，正如下面将要分析的，这些算法在处理祖先/后裔关系中的特例，即双亲/孩子关系的结构连接时，一些在孩子列表中被扫描的元素结点，通过利用双亲结构信息

(即 $parentMax$),可以事先判断它们并不参与双亲/孩子关系的结构连接,因此,可以通过 B$^+$-树索引来跳过(即算法 1 中的第 8 步),即并不需要对它们进行顺序扫描.

为了简化描述,我们在本文中也仅仅考虑一个单一的、大的 XML 文档. 通过利用 B$^+$-树索引,将这些算法推广到处理多文档是非常容易的,只需要增加对 $docID$ 的判断和索引定位即可.

4.2 处理双亲/孩子关系的结构连接算法

为了反映在处理双亲/孩子关系的结构连接时,如何根据双亲结构信息利用索引跳过不需要参与连接的元素结点的情形,考虑图 7 所反映的样例. 在图 7 中,细线段表示双亲列表 P 中的元素结点的[$order$,$maxOrder$]区间,粗实线段表示孩子列表 C 中的元素结点的[$order$,$maxOrder$]区间,粗虚线段表示其它元素结点的[$order$,$maxOrder$]区间. 图 7 所反映的是在处理双亲/孩子关系的结构连接时,孩子列表 C 中的哪些元素结点能够被避免顺序扫描的情形.

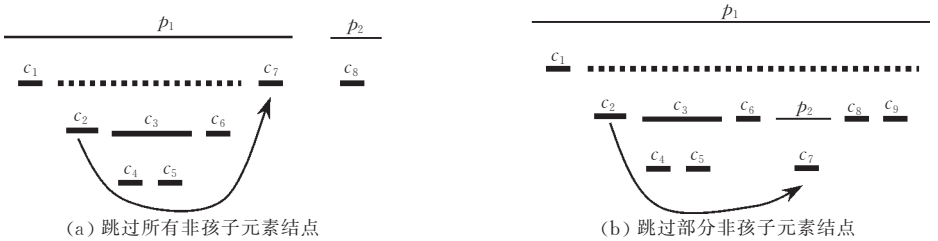

图 7 根据双亲结构信息来避免后裔元素结点被顺序扫描的情形

对于图 7(a)的情形,Stack-Tree-Desc 算法和 Anc-Des-B+算法都将处理如下:

1. 压 p_1 进栈,并将 p_1 与 c_1 进行连接(即 p_1 与 c_1 满足双亲/孩子关系);

2. 顺序扫描孩子列表 C 中的 c_2 到 c_6 元素结点,判断它们是否是 p_1 的孩子(事实上,它们并不满足双亲/孩子关系);

3. 连接 p_1 与 c_7;

4. p_1 出栈;

5. 压 p_2 进栈,并将 p_2 与 c_8 进行连接.

明显地,第 2 步是浪费的. 能够根据 c_2 的双亲结构信息利用 B$^+$-树索引避免 c_2 到 c_6 的顺序扫描. 在第 1 步之后,能够直接从 c_2 定位到 c_7. 这里,c_7 是孩子列表 C 中 $order$ 值大于 $c_2.parentMax$ 的第一个元素结点(即 c_7 是满足 $c.order>c_2.parentMax$ 条件中有最小 $order$ 值的元素结点).

对于图 7(b)的情形,在 p_1 与 c_1 进行连接之后,Stack-Tree-Desc 算法和 Anc-Des-B+算法都将顺序地扫描孩子列表 C 中的 c_2 到 c_6 元素结点. 事实上,利用 B$^+$-树索引能够直接从 c_2 跳到 c_7. 这里,c_7 是孩子列表 C 中 $order$ 取值大于 $p_2.order$ 的第一个元素结点. 此时 p_1 已经压进栈,p_2 表示双亲列表 P 中的当前元素结点.

算法 1 所示的是我们提出的处理双亲/孩子关系的结构连接算法,称为 PC-Join 算法. 首先,设变量 p 和 c 分别指向双亲列表 P 和孩子列表 C 的第一个记录(有最小的 $order$ 值);然后,算法依次扫描两个列表(即变量 p 和 c 依次指向两个列表的下一个记录),并对 p 和 c 所指向的记录进行判断,将符合双亲/孩子关系的元素结点对输出. 在算法执行过程中,需要维护一个祖先元素结点栈,该栈用来存放后来可能需要进行双亲/孩子关系连接的祖先元素结点. 具体过程如下:如果双亲列表中的 p_1,p_2,…,p_k(这里 $p_i.order<p_{i+1}.order$)都是孩子列表中的 c 的祖先,则将 p_1 到 p_k 都压入栈;判断栈顶元素 p_k 是否是 c 的双亲,如果是则输出结点对〈p_k,c〉,否则不输出;将 c 指向孩子列表的下一记录 c';将栈中不是 c' 的祖先元素结点全部出栈……这些处理过程类似于 Stack-Tree-Desc 算法和 Anc-Des-B+算法. 但是,在 PC-Join 算法中,第 8 步能够根据双亲结构信息利用 B$^+$-树索引跳过孩子列表中不参与连接的元素结点. 在实际应用中,为了避免不必要的 B$^+$-树索引的存取,可以首先检查下一个将要定位的孩子元素结点是否在当前页中(通过检查当前页中最后一个元素结点的 $order$ 值).

算法 1. PC-Join.

输入:P 和 C. P,C 分别是参与连接的双亲列表、孩子列表

输出:{〈p,c〉}. 列表 P 和 C 中满足双亲/孩子关系的元素结点对的序列

PC-Join(P,C)

1. 令 p 和 c 分别指向列表 P 和 C 的第一个记录. //p 和 c 分别表示 P 和 C 的当前记录

2. 令祖先栈 $Stack$ 为空. //用来存放后来可能需

```
        //要进行结构连接的祖先元素
3.  令 Sp 指向列表 P 的当前记录.    //如果栈非空,
        //Sp 表示栈顶元素；否则表示 P 的当前记录
4.  While (c 未指向表底,且 p 未指向表底或 Stack 非
    空)
5.    If (Sp 是 c 的祖先) then
6.      顺序扫描 P 表,并将 P 表中所有 c 的祖先压
        入 Stack 中. //设 Sp 指向栈顶元素
7.      If (Sp 是 c 的双亲) then 输出双亲/孩子对
        〈Sp,c〉,并令 c 指向列表C 中 order 值大于
        min(c.maxOrder,p.order)的第一个记录.
8.      Else 令 c 指向列表C 中 order 值大于
        min(c.parentMax,p.order)的第一个记录.
        //根据双亲结构信息在孩子表中跳过尽可能
        //多的不参与连接的元素结点
9.    Else if (c 位于 Sp 之后) then
10.     将Stack 中位于 c 之前的所有元素出栈,并设
        e 是最后一个出栈的元素；如果没有元素
        出栈,则令 e=p.
11.     令p 指向列表 P 中 order 值大于 e.maxOrder
        的第一个记录.
12.     If (Stack 为空) then 令 Sp 指向列表 P 的当
        前记录.
13.     Else 令 Sp 指向 Stack 的栈顶元素.
14.   Else 令 c 指向列表C 中 order 值大于Sp.order的
        第一个记录.
15. Endwhile
```

算法 PC-Join 利用 B^+-树索引在双亲列表和孩子列表中尽可能多地跳过不需要参与连接的元素结点(如算法 1 中的第 7,8,11 和 14 步所示),因此,它最多只需要对参与连接的两个列表分别进行一次扫描. 本算法的另一个优点是输出的双亲/孩子结点对序列{〈p,c〉}是按孩子结点 c 的文档位置有序的,这是 XML 查询所要求的,因此,它还可以节省对连接结果的排序代价.

4.3　处理拥有关系的结构连接算法

在 XML 的路径表达式查询中,还有一类特殊的祖先/后裔关系的结构连接,例如,查询 *book*[*chapter*]是返回所有至少拥有一个 *chapter* 孩子元素的 *book* 元素;查询 *book*[*descendant*::*email*]是返回所有至少拥有一个 *email* 后裔元素的 *book* 元素;而查询//*keyword*[contains(*descendant-or-self*::*,"king")]则是返回所有自身及后裔元素的内容中至少拥有一个关键字"king"的 *keyword* 元素,它是通过 Elem_*keyword* 列表与 Word_*king* 列表进行祖先/后裔关系的结构连接来实现关键字搜索操作.我们称这类特殊的包含关系为拥有关系,并记参与拥有关系结构连接的两个列表分别为 A 和 D.

如果利用 Stack-Tree-Desc 算法或 Anc-Des-B+算法来处理这类拥有关系的结构连接,其效率是较低的,它的处理过程如图 8 所示. 这里,假设 a_1 元素拥有 $d_1 \sim d_4$(即 d_i 是 a_1 的后裔、孩子或内容中包含的关键字),a_2 不拥有任何元素或关键字,a_3 拥有 $d_6 \sim d_8$. 此时,拥有关系结构连接的返回结果是由 a_1 和 a_3 组成的结点序列. 但是,结构连接算法 Stack-Tree-Desc 或 Anc-Des-B+却需要对两个列表 A 和 D 分别进行一次扫描和匹配,这就类似于在商用关系数据库系统中来实现这种连接操作,它将首先返回所有满足匹配条件的结点对(图中的实线和虚线所示的连接结果),然后再对返回的结果进行 distinct 操作(去除图中所有虚线所示的连接结果),因此它的执行效率是较低的. 本文提出的算法 2 则能更高效地处理这种拥有关系的结构连接操作.

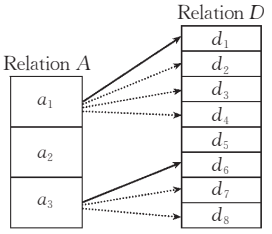

图 8　拥有关系结构连接示意图

拥有关系包括祖先/后裔拥有关系和双亲/孩子拥有关系. 如图 9 所示的是双亲/孩子拥有关系的结构连接情况,其中,参与连接的双亲列表和孩子列表中的元素结点分别用正方形和三角形表示,带阴影的正方形表示结构连接的返回结果.

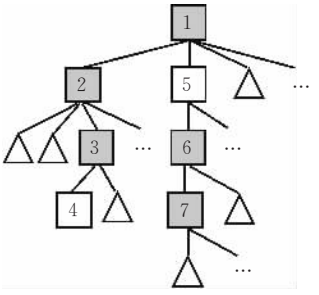

图 9　双亲-孩子拥有关系结构连接示意图

算法 2 中的栈和链表仅仅在处理双亲/孩子拥有关系时发挥作用. 栈用来存放可能在后面的双亲/孩子拥有关系结构连接中需要继续判断的祖先元素;栈中的每一个元素可以指向一个链表,且所指向链表中存放的是按文档顺序位于栈中该元素之后

的已满足双亲/孩子拥有关系结构连接的双亲元素.为了实现按双亲元素的文档顺序输出所有满足双亲/孩子拥有关系结构连接的双亲元素,这些链表中存放的双亲元素需要等到栈底元素出栈之后才能依次输出.

算法 2. Holding-Join.

输入:A 和 D.A,D 分别是参与连接的祖先列表、后裔列表

输出:{a}.列表 A 中满足拥有关系的元素结点的序列

Holding-Join(A,D)

1. 令 a 和 d 分别指向列表 A 和 D 的第一个记录.
   // a 和 d 分别表示 A 和 D 的当前记录
2. 令祖先栈 *Stack* 为空. //设 *Sa* 指向栈顶元素
3. While (a 和 d 均未指向表底,或 *Stack* 非空)
4. If (*Stack* 非空,且 *Sa* 拥有 d 或 d 位于 *Sa* 之后) then
5. If (d 位于 *Sa* 之后) then 将 *Stack* 中所有位于 d 之前的元素逐一出栈,并将出栈元素所指向的链表直接链接到栈中下一元素所指向链表的后面;如果遇栈底元素出栈,则输出栈底元素所指向链表中的所有元素.
6. Else if (栈顶元素位于栈底) then 输出栈顶元素以及它所指向链表中的所有元素,并将栈顶元素出栈.
7. Else 栈顶元素出栈,并将它追加到新栈顶元素所指向链表中,同时将已出栈元素所指向的链表直接链接到新栈顶元素所指向链表的后面.
8. If (a 拥有 d) then
9. If (*Stack* 为空) then 输出 a.
10. Else 将 a 追加到栈顶元素所指向链表中.
11. If (双亲/孩子拥有关系) then 令 a 指向列表 A 中 *order* 值不小于 d.*order* 的第一个记录.
12. Else 令 a 指向列表 A 的下一个记录.
13. Else if (a 位于 d 之前) then 令 a 指向列表 A 中 *order* 值大于 a.*maxOrder* 的第一个记录.
14. Else if (a 是 d 的祖先) then 将 a 压入栈 *Stack* 中,并令 a 指向列表 A 的下一个记录.
15. Else //d 是 a 的祖先或 d 位于 a 之前
16. If (双亲/孩子拥有关系且栈非空) then
17. 令 d 指向列表 D 中 *order* 值大于 min(d.*parentMax*,a.*order*)的第一个记录.
18. Else 令 d 指向列表 D 中 *order* 值大于 a.*order* 的第一个记录.
19. Endwhile

算法 Holding-Join 利用 B^+-树索引在祖先列表和后裔列表中尽可能多地跳过不需要参与连接的元素结点(如算法 2 中的第 11,13,17 和 18 步所示),也就是说,它最多只需要对参与连接的两个列表分别进行一次扫描;同时,它仅仅输出拥有关系查询语义所需要的不重复元素结点,因此,本算法在处理拥有关系结构连接时比现有算法要更加高效.本算法的另一个优点是输出的元素结点序列是按文档位置排序的,这是 XML 查询所要求的,因此,它还可以节省对连接结果的排序代价.

对于图 9 所示双亲/孩子拥有关系的结构连接实例,栈和链表的运行结果如图 10 所示,首先,a_1 结点进栈,随后 a_2 和 a_3 结点被追加到 a_1 所指向的链表中,状态如图 10(a)所示;其次,a_5,a_6 结点先后进栈,且 a_7 结点被追加到 a_6 结点所指向的链表中,状态如图 10(b)所示;第三,a_6 结点出栈,由于该结点满足双亲/孩子拥有关系,因此,将 a_6 追加到栈顶结点 a_5 所指向的链表中,同时将已出栈的 a_6 结点所指向的链表直接链接在栈顶结点 a_5 所指向的链表的后面,状态如图 10(c)所示;第四,a_5 结点出栈,由于该结点不满足双亲/孩子拥有关系,因此,将已出栈的 a_5 结点所指向的链表直接链接在栈顶结点 a_1 所指向的链表的后面,状态如图 10(d)所示;最后,a_1 结点出栈,输出 a_1(如果 a_1 结点不满足双亲/孩子拥有关系,则不输出),同时,输出结点 a_1 所指向链表中的所有结点.

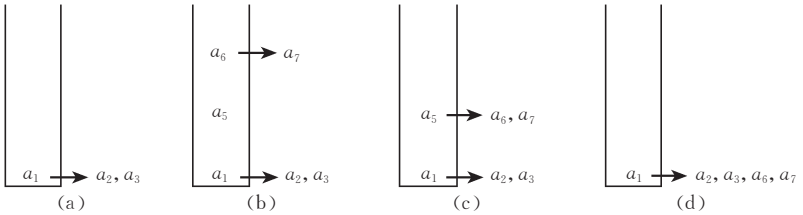

图 10 双亲/孩子拥有关系结构连接时栈和链表的执行示意图

5 实验结果

5.1 实验环境

我们对 PC-Join(简记为 PCJ)、Holding-Join(简记为 HJ)算法进行了性能测试,并且将它们与 Stack-Tree-Desc[2](简记为 STD)和 Anc-Des-B+[3](简记为 ADB)算法进行了性能比较,所有的测试程序均用 MS VC++6.0 编写.实验平台是 Pentium Ⅳ PC,主频 1.5GHz,内存 256M,30G 硬盘,操作系统为

Windows 2000 professional.

图 11 所示的是测试 XML 数据集的 DTD. 其中，标有“A”的边表示它所指向的是属性；标有“+”，“*”，“?”的边分别表示它所指向的元素在文档中可以重复出现一次或多次、零次或多次、零次或一次；*section* 元素可以嵌套出现零次或多次；*text* 是一个混合元素（即它的内容可以是文本与子元素的混合），它的孩子 *bold*，*keyword* 和 *emph* 也都是混合元素且它们都允许嵌套或相互嵌套零次或多次.

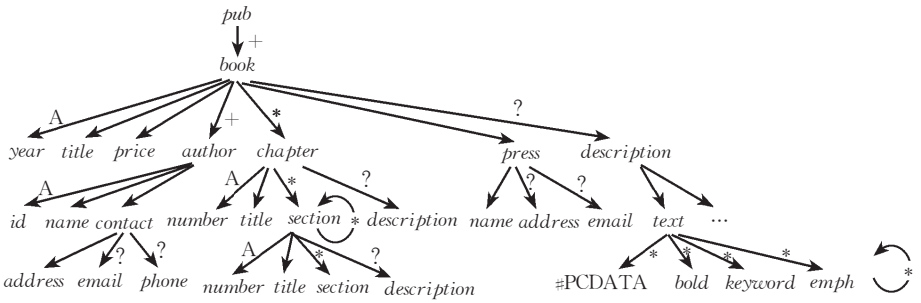

图 11　测试数据集的 DTD

由于测试需要在多种不同文档结构下进行，为了避免使用多个具有不同 DTD 的 XML 文档，我们设计了这样的复杂结构. 该 DTD 基本涵盖了常见的 XML 文档结构，具有一般性.

利用我们设计的 XML 文档生成器自动生成一个符合图 11 所示 DTD 的 XML 文档，文档大小为 80.8MB. 对于该 XML 文档，在每个 *book* 元素中，随机产生 5～10 个 *author* 元素结点，0～5 个 *chapter* 元素结点；在每个 *chapter* 元素中，随机产生 0～5 个 *section* 元素结点；在每个 *section* 元素中，随机产生 0～5 个 *section* 元素结点，且 *section* 元素结点随机嵌套 0～4 层；在每个 *text* 元素中，分别随机产生 0～8 个 *bold*，*keyword* 和 *emph* 元素结点；在每个 *bold*、*keyword* 或 *emph* 元素中，分别随机产生 0～8 个 *bold*，*keyword* 和 *emph* 元素结点，且它们随机嵌套（或相互嵌套）0～4 层. 在该 XML 文档中，共有 1979027 个元素和属性结点，其中，3000 个 *book* 元素结点，482775 个 *title* 元素结点，7496 个 *chapter* 元素结点，472279 个 *section* 元素结点，189264 个 *description* 元素结点，131576 个 *keyword* 元素结点，39010 个关键字“king”.

此外，利用我们设计的 XML 语法分析器和转储器，对 XML 文档进行语法分析并生成各种索引列表和值列表，各种索引列表和值列表均采用 B^+-树索引文件结构进行存储.

我们的实验是为了测试 4 种算法针对不同数据集、运行不同查询时的性能. 我们选取了 5 个测试查询实例，分别是 *book*/*title*，*chapter*/*section*，*section*/*title*，*section*/*description* 和 *book*/*child*::*. 其中，*book*/*title* 代表最简单的情况，即一个双亲只有一个孩子；*chapter*/*section* 则代表一个双亲对应多个孩子元素结点且孩子元素结点嵌套的情况；*section*/*title* 反映双亲元素结点嵌套且孩子元素结点位于最左边的情况；*section*/*description* 反映双亲元素结点嵌套且孩子元素结点位于最右边的情况；*book*/*child*::* 也是一个常见查询，它没有明确指定连接的孩子结点的标记名，反映了对整个 XML 文档索引（即结构索引）列表的查询. 对于 Holding-Join 算法，前 4 个查询实例分别对应为 *book*[*title*]，*chapter*[*section*]，*section*[*title*]和 *section*[*description*]；第 5 个查询实例则对应为 *book*[*child*::*]，由于该查询无太大意义，我们改用两个关键字搜索的实例进行测试，即双亲/孩子关系的关键字搜索 *keyword*[contains(.,“king”)]和祖先/后裔关系的关键字搜索 *keyword*[*contains*(*descendant-or-self*::*,“king”)]，它的含义是搜索其文本内容（或它的后裔元素或自身的文本内容）中包含关键字“king”的 *keyword* 元素，它将被转化为在 Elem_*keyword* 与 Word_*king* 列表之间按双亲/孩子或祖先/后裔拥有关系的结构连接操作. 这几个查询实例基本上概括了常见的双亲/孩子关系和拥有关系的结构连接.

与文献[12]的实验测试思路类似，我们也通过改变选择度来进行测试. 为了实现这个目的，我们在原有双亲、孩子元素列表的基础上，通过改变双亲结点的百分比（即从双亲列表中随机删除一定比例的元素结点）来对各种结构连接算法的性能进行测试比较，双亲结点的百分比按从大到小的顺序分别选取 100%，75%，50%，30%，10%和 1% 6 个值.

5.2　实验结果分析

与文献[12]相同，选取的性能指标也是两个：

① 扫描元素的数目. 即结构连接时读取元素结点(即记录)的数目,这个指标能够反映出算法跳过不相关元素结点的能力.

② 耗用时间. 连接耗用的时间,用于反映算法的综合性能.

当孩子结点保持不变,双亲结点百分比变化时,4 种算法扫描元素结点的数量如表 1 所示.

表 1 孩子结点不变并改变双亲结点百分比时 4 个结构连接算法实际扫描元素结点数量的对比(单位:千记录)

(a) *book* 与 *title*

Join-P(%)	STD	ADB	PCJ	HJ
100	486	486	26.0	6.0
75	485	369	20.0	4.0
50	484	240	14.0	3.0
30	484	141	8.0	2.0
10	483	52	3.0	0.6
1	480	6	0.4	0.1

(b) *chapter* 与 *section*

Join-P(%)	STD	ADB	PCJ	HJ
100	480	480	119	15.0
75	478	362	92	12.0
50	476	247	64	8.0
30	474	141	38	4.0
10	473	52	13	2.0
1	472	6	2	0.2

(c) *section* 与 *title*

Join-P(%)	STD	ADB	PCJ	HJ
100	955	955	955	945
75	837	837	837	709
50	719	717	715	472
30	624	611	608	284
10	530	402	397	95
1	487	368	354	11

(d) *section* 与 *description*

Join-P(%)	STD	ADB	PCJ	HJ
100	662	662	662	654
75	543	543	543	501
50	425	425	425	341
30	330	327	327	210
10	236	215	215	80
1	194	35	35	8

(e) *book* 与 *child* ::*

Join-P(%)	STD	ADB	PCJ
100	1982	1982	165
75	1981	1497	124
50	1981	982	84
30	1980	583	50
10	1978	208	17
1	1967	24	2

分析表 1 可知,从横向比较,在每种相同的百分比下,4 种算法扫描的结点数目是不一样的,其中以 PC-Join 和 Holding-Join 为优. 从纵向比较,随着百分比的降低,4 种算法扫描的结点数目都在减少,但减少的幅度不同,PC-Join 和 Holding-Join 减少的幅度明显大于其它两种. 这种结果是在我们预料之中的,首先,对于 PC-Join 算法而言,它在 Anc-Des-B+算法的基础上,还能够利用双亲结构信息进一步跳过可以事先判断并不参与连接的孩子元素结点(见算法 1 中的第 8 步),这对于在孩子列表中存在大量的双亲元素结点的后裔但不是它们的孩子的情况特别有效(这些是后裔但不是孩子的元素结点能够通过双亲结构信息来跳过),如表 1 的(a),(b),(e)三种情况所示. 其次,对于 Holding-Join 算法而言,由于对于每个祖先(或双亲)元素,它只要找到一个后裔(或孩子)元素即可,因此,它对于后裔(或孩子)重复较多且祖先(或双亲)不存在嵌套的情况特别有效,如表 1(a),(b)所示;如果祖先(或双亲)存在嵌套的话,当后裔(或孩子)元素位于被嵌套祖先(或双亲)元素的左边出现,这种情况也较有效,如表 1(c)所示.

算法运行时间如图 12 所示. 其中,横坐标表示双亲(或祖先)元素结点的百分比;纵坐标表示结构连接算法的执行时间,单位是 s.

从运行时间上来看,PC-Join 和 Holding-Join 算法总体上也要优于其它两种算法,它们的性能优势随着百分比的降低而愈发突出.

对于图 12(a),(b),(e)三种情况,由于 PC-Join 或 Holding-Join 算法都能够很大幅度地减少被扫描元素结点,因此它们的性能优势特别明显.

对于图 12(c),由于孩子元素 *title* 不能够根据双亲结构信息利用索引来加速结构连接的计算,因此 PC-Join 算法与 Anc-Des-B+算法的效果基本相同;当百分比较低时,由于能够较大幅度地减少被扫描元素结点,因此它们的性能略优于顺序扫描的 Stack-Tree-Desc 算法;对于 Holding-Join 算法,由于它能够较大幅度地减少被扫描元素结点,因此它比其它三种算法具有较大的性能优势.

对于图 12(d),由于双亲元素 *section* 和孩子元素 *description* 都不能利用索引来跳过太多不需要参与结构连接的元素,因此,4 种算法的性能相差不大;PC-Join 算法与 Anc-Des-B+算法的效果基本相同,当百分比较低时略优于顺序扫描的 Stack-Tree-Desc 算法.

单独考察处理拥有关系的结构连接时,我们可以发现 Holding-Join 算法不仅对一般的拥有关系支持较好,对于涉及到关键字搜索的拥有关系连接性能也很好.

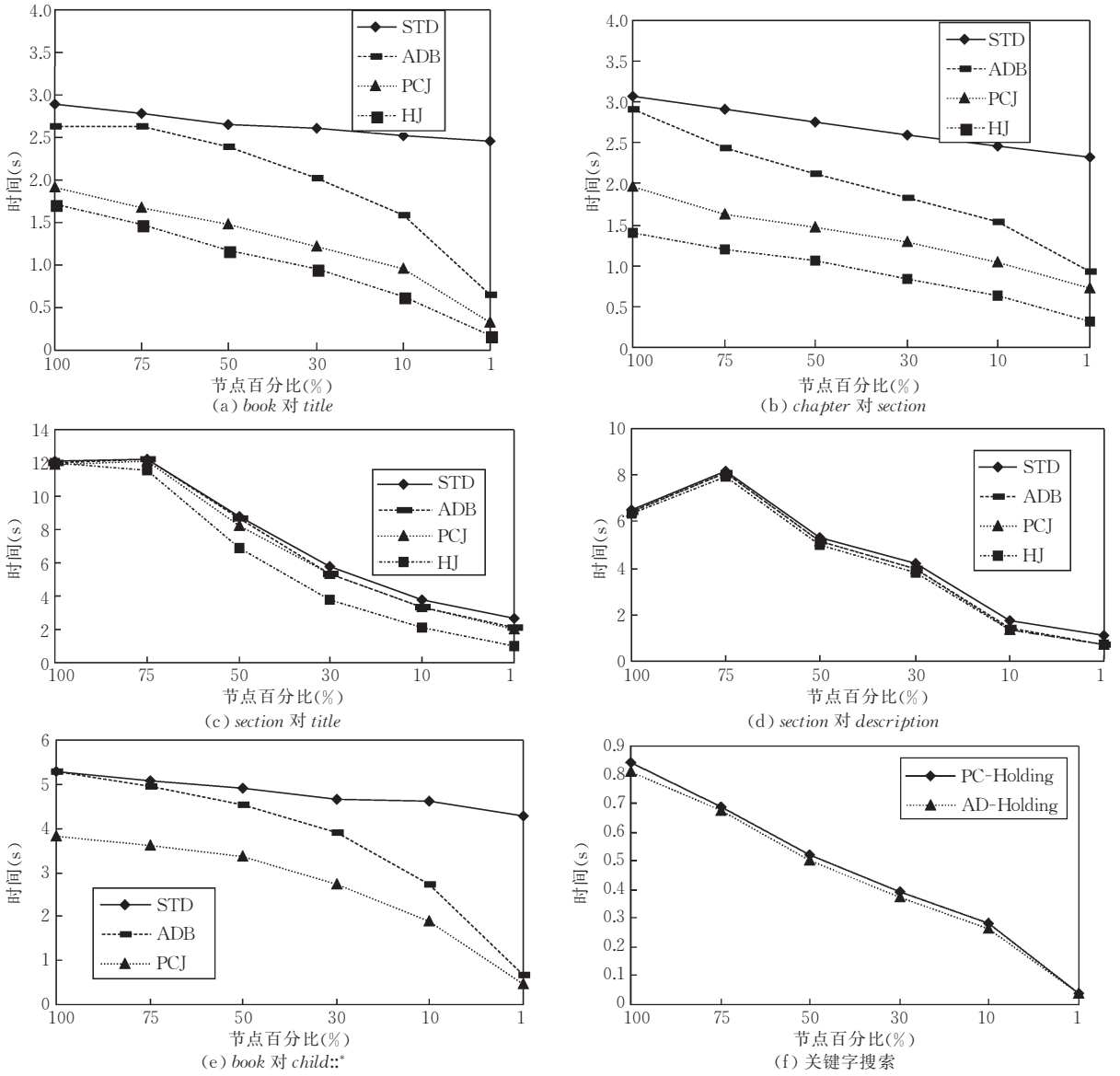

图 12　孩子(或后裔)结点不变并改变双亲(或祖先)结点百分比时 4 个结构连接算法实际运行耗用时间的对比

6　结　论

与文献[7]相比，本文提出的索引结构的主要改进及其意义如下：

(1) 在元素列表、属性列表和结构列表中增加了 *parentOrder*，*parentMax* 信息，它不仅可以用来有效地实现双亲操作和双亲/孩子关系结构连接，而且可以用来判断结点之间的兄弟关系(XPath 中的轴操作). 文献[7,8]中给出的祖先/后裔关系的结构连接算法都会导致对后裔元素列表的多次扫描，因此效率较低；文献[2,3]中提出的祖先/后裔关系的结构连接算法仅仅需要对参与连接的两个列表分别进行一次扫描. 在此基础上，本文提出的两个处理双亲/孩子关系和拥有关系的结构连接算法 PC-Join 和 Holding-Join，能够根据结点的双亲结构信息等利用 B^+-树索引尽可能多地跳过不需要参与连接的元素结点.

(2) 在元素列表中增加了 ID 属性值 *attrID*，以加速解除参照操作(它是 XML 查询中最常用的操作之一)以及按 ID 属性值选择元素结点的操作.

(3) 对文献[7]的结构列表进行了改造，取消了 *siblingOrder*(第一个兄弟的 *order*)，*childOrder*(第一个孩子的 *order*)和 *attributeOrder*(第一个属性的 *order*)等信息. 理由是：首先，在结构列表中，当前元素结点的下一记录表示的就是该元素结点的第一个属性结点，因此无须 *attributeOrder* 信息；其次，通过 *childOrder* 和 *siblingOrder* 信息来定位每一

个上下文元素结点(通常上下文元素结点是一个结点序列)的所有孩子结点会导致对结构列表的多次扫描,因此效率较低. 应该使用双亲/孩子关系的结构连接算法来实现. 我们的结构列表是用来支持结点测试中含种类测试(如 *book*/*child*::*node*())或名字测试中含通配符(如 *book*/*child*::*)的路径步的计算以及有效地实现 XML 文档片断的重构(由于给定元素结点的所有后裔信息被有序地聚集存储).

(4) 在元素列表和结构列表中增加了 *nodeType* 信息,并在值列表 Value 中增加了 *afterOrder* 信息,以支持混合元素结点和空元素结点. 而文献[7]仅考虑元素结点的内容要么是文本值要么是子元素的情况.

(5) 在元素列表中增加了 *ssIndex* 信息. 位置查询是一个代价较高的查询,需要临时对返回的结点序列产生位置序号. 其中,同名兄弟结点之间的位置查询又是 XML 查询中最常用的操作,因此,我们事先产生并存储 *ssIndex* 信息,以加快同名兄弟结点之间的位置查询的执行效率.

(6) 将编码方案[7]与路径索引[13]相结合,增加了路径列表 Path,相应地在元素列表、属性列表和值列表中增加了路径标识 *pathID*,使得对路径表达式的计算不仅可以使用逐步结构连接法,而且可以使用路径法(选择代价较小的方法),以加快路径表达式的计算.

(7) 将编码方案[7]与逆序列表[8]相结合,增加了关键字列表 Word_*Keyword*,以有效地支持关键字搜索. 由于 XML 数据本质上是半结构化数据,它不同于完全的结构化数据,它有模式,但用户可能并不完全了解这些模式;同时模式并不是必须的,且模式易变或不完全,甚至还可能没有模式;有时还表现出文本文件的特征. 因此,关键字搜索对于 XML 文档来说也是非常重要的.

本文的主要贡献有:

(1) 结合本文的上下文给出了一个 XML 树数据模型的形式化定义.

(2) 将编码方案、逆序列表和路径索引的思想相结合,提出了一种改进的 XML 数据的索引结构,它能够有效地支持 XQuery 路径表达式查询和关键字搜索.

(3) 给出了两个处理双亲/孩子关系和拥有关系的结构连接算法,它们都最多仅需要对参与连接的两个列表分别进行一次扫描. 与文献[3]中的 Anc-Des-B+算法相比,PC-Join 算法能够根据双亲结构信息利用 B^+-树索引尽可能多地跳过孩子列表中不需要参与连接的元素结点,从而进一步提高了连接的效率. 对 PC-Join,Holding-Join 算法进行了性能测试,并且将它们与 Stack-Tree-Desc[2]和 Anc-Des-B+[3]算法进行了性能比较,实验结果表明,与文献[2,3]的算法相比,本文提出的处理双亲/孩子关系和拥有关系的结构连接算法 PC-Join 和 Holding-Join 总体上具有更好的性能.

另外,PC-Join 算法和 Holding-Join 算法也适合于处理整体小枝连接查询[4,5].

XML 查询语言与 SQL 相比更具灵活性,特别是路径表达式的使用,因此,下一步我们将对 XPath 中的 preceding,following,preceding-sibling 和 following-sibling 轴的有效计算以及 XML 的查询优化技术等进行研究.

参 考 文 献

1 World-Wide Web Consortium. XQuery 1.0 and XPath 2.0 data model. W3C working draft, 23 July 2004. http://www.w3.org/TR/xpath-datamodel/

2 Al-Khalifa S., Jagadish H. V., Koudas N. *et al*.. Structural joins: A primitive for efficient XML query pattern matching. In: Proceedings of the IEEE ICDE Conference, San Jose, California, 2002, 141～152

3 Chien Shu-Yao, Vagena Z., Zhang Donghui *et al*.. Efficient structural joins on indexed XML documents. In: Proceedings of the VLDB Conference, HongKong, 2002, 263～274

4 Bruno N., Koudas N., Srivastava D.. Holistic twig joins: Optimal XML pattern matching. In: Proceedings of the ACM SIGMOD Conference, Madison, Wisconsin, 2002, 310～321

5 Jiang Hai-Feng, Wang Wei, Lu Hong-Jun *et al*.. Holistic twig joins on indexed XML documents. In: Proceedings of the VLDB Conference, Berlin, 2003, 273～284

6 Liu Yun-Sheng, Wan Chang-Xuan, Xu Sheng-Hua. Efficient implementing RPE query in an RDBMS. The Mini-Micro Systems, 2003, 24(10): 1764～1771(in Chinese)
(刘云生,万常选,徐升华. 基于关系数据库有效地实现 RPE 查询. 小型微型计算机系统, 2003, 24(10): 1764～1771)

7 Li Q., Moon B.. Indexing and querying XML data for regular path expressions. In: Proceedings of the VLDB Conference, Roma, 2001, 361～370

8 Zhang C., Naughton J., DeWitt D. *et al*.. On supporting containment queries in relational database management systems. In: Proceedings of the ACM SIGMOD Conference, Santa Barbara, California, 2001, 426～437

9 Milo T., Suciu D.. Index structures for path expressions. In: Proceedings of the ICDT Conference, Jerusalem, 1999, 277～295

10 Chan Chee-Yao, Garofalakis M., Rastogi R.. RE-Tree: An efficient index structure for regular expressions. In: Proceedings of the VLDB Conference, Hong Kong, 2002, 251～262

11 Kaushik R., Bohannon P., Naughton J. F. *et al*.. Covering indexes for branching path queries. In: Proceedings of the ACM SIGMOD Conference, 2002, 133～144

12 Jiang Hai-Feng, Lu Hong-Jun, Wang Wei *et al*.. XR-Tree: Indexing XML data for efficient structural joins. In: Proceedings of the IEEE ICDE Conference, Bangalore, India, 2003, 253～264

13 Yoshikawa M., Shimura T., Uemura S.. XREL: A path-based approach to storage and retrieval of XML documents using relational databases. ACM Transactions on Internet Technology, 2001, 1(1): 1～29

14 Tatarinov I., Viglas S. D., Beyer K. *et al*.. Storing and querying ordered XML using a relational database system. In: Proceedings of the ACM SIGMOD Conference, Madison, Wisconsin, 2002, 204～215

15 Wan Chang-Xuan, Liu Yung-Sheng. X-RESTORE: Middleware for XML's relational storage and retrieve. Wuhan University Journal of Natural Science, 2003, 8(1A): 28～34

16 Dietz P. F.. Maintaining order in a linked list. In: Proceedings of the Annual ACM Symposium on Theory of Computing, San Francisco, California, 1982, 122～127

17 Wan Chang-Xuan, Liu Yun-Sheng, Xu Sheng-Hua, Lin Da-Hai. Querying XML view in X-RESTORE. The Mini-Micro Systems, 2004, 25(10): 1870～1875 (in Chinese)
(万常选，刘云生，徐升华，林大海. 基于 X-RESTORE 查询 XML 视图. 小型微型计算机系统, 2004, 25(10): 1870～1875)

18 Liu Xi-Ping, Wan Chang-Xuan. Minimization of XPath queries with constraints. Journal of Computer Research and Development, 2004, 41(Suppl): 342～348 (in Chinese)
(刘喜平，万常选. 带约束 XPath 查询的最小化. 计算机研究与发展, 2004, 41(增刊): 342～348)

WAN Chang-Xuan, born in 1962, Ph. D., professor. His current research interests include XML database, Web information management, data mining, and electronic commerce.

LIU Yun-Sheng, born in 1940, professor, Ph. D. supervisor. His current research interests include modern database theory and technology, development of database and information systems.

XU Sheng-Hua, born in 1952, professor, Ph. D. supervisor. Her current research interests include information management and information system.

LIU Xi-Ping, born in 1981, master candidate. His current research interests include XML database and Web information management.

LIN Da-Hai, born in 1980, assistant. His current research interests include XML database.

Background

This work is supported by the Natural Science Foundation of Jiangxi province with the title "the Research on Query Optimization Techniques for Relational Database-Based XML Database System". In this project, we study the query optimization techniques of relational database-based XML database system. This work consists of two parts. One is the query optimization techniques for the RDB-based query middleware X-RESTORE. The main issue here is the optimization before transforming XML query to SQL query, including query minimization, view query rewriting and so on. The other part is the query optimization techniques inside the relational query engine, such as the optimal structural join algorithms, reasonable statistical information extracted from XML documents, cost-based estimation of the answer size of structural joins and the selection of query plans based on the estimation.