一种基于结构特征的XML数据查询方法*)

秦 杰[1] 赵淑梅[2] 杨树强[3]

(河南工业大学信息科学与工程学院 郑州 450052)[1] (国防科技大学计算机学院 长沙 410073)[2]

摘 要 针对XML数据特有的树型结构模式,提出了一种将树型结构的XML数据和查询语句转化为特定格式的字符串,基于串匹配原理对结构复杂的XML数据进行查询的方法,避免了传统的基于路径的查询方式所必需的路径之间的连接(join)操作,从而提高查询效率。利用本文提出的编码方式,可以建立关于XML数据结构和数据内容合为一体的索引。实验显示,本文使用的针对XML数据查询的方法比传统的基于连接操作的数据查询方式高效,且本方法具有良好的扩展性。

关键词 XML,结构编码,查询,索引

A Structure-encoded Method for Searching XML Datas

QIN Jie[1] ZHAO Shu-Mei[2] YANG Shu-Qiang[2]

(School of Information Science and Engineering, Henan University of Technology, Zhengzhou 450052)[1]

(School of Computer, National University of Deference Technology, Changsha 410073)[2]

Abstract Aiming at the special tree-structure of XML data, we propose a novel structure-encoded method for searching XML documents. By representing both XML documents and XML queries in structure-encoded sequences, we show that querying XML data is equivalent to finding subsequence matches. Unlike index methods that disassemble a query into multiple sub-queries, and then join the results of these sub-queries to provide the final answers. Our structure-encoded method uses tree structures as the basic unit of query to avoid expensive join operations. The proposed method also provides a unified index on both content and structure of the XML documents; hence it has a performance advantage over methods indexing either just content or structure. Experiments show that the proposed method is effective, scalable, and efficient in supporting structural queries.

Keywords XML, Structure-encoded, Query, Index

1 引言

XML(eXtensible Markup Language)正在成为Web数据表达和数据交换的标准,XML文档的大量出现,产生了对XML数据管理的需求。XML所描述的Web数据所具有的半结构化(Semi-structure)特性对传统的数据管理方式提出了挑战:传统的基于关系数据库方式的信息查找以及信息检索方式不能为具有树状层次结构的XML数据提供令人满意的查询效果。基于XML的信息查询成为一个研究重点[1~4]。

为了提高XML数据的查询处理效率,研究人员提出了多种不同的索引技术[1~5]。这些索引方式都需要对中间结果进行JOIN操作。然而,在查询过程中,最影响查询效率的就是JOIN操作。另外,现有的查询方法在处理带有"*"或者"//"符号的查询时,查询效率显得低下[6]。在这些已有的方法中,比较具有代表性的方法是XISS[1]和Index Fabric[2]。

XISS方法利用任何包含正则表达式的结构化查询都可以分解成基本的查询单元来完成查询这一特点。把XML数据中的每个不同名称的元素或者属性作为基本查询单元,分别对这些基本的查询单元建立索引。把任意一个复杂的路径表达式分解成若干基本的查询单元,这些基本的查询单元可以通过预先建立的索引直接进行查询。之后,将得到的中间结果通过JOIN操作,从而获得最终的查询结果。

Index Fabric把路径作为基本的查询单元,通过为XML数据集中所有从树根节点到树中其它节点的不同路径建立索引来加快查询速度。这种方法适合单路径查询。如果查询语句结构带有分支,则需要把查询语句分解成若干条单路径,分别执行查询,之后对各个单路径查询的结果进行JOIN操作,才能得到最终结果。

文[6]介绍了一种基于后缀树(suffix tree)[7]的字符串匹配方法(这里称之为RIST算法),该方法使用两元组(Name, Path)描述XML数据中的元素、属性和值。其中Name表示XML文档树中的节点名称,Path表示从文档树根到该节点的父节点路径。数据对象中的元素j与查询语句中元素i匹配的条件是查询对象的$Name_j = Name_i$,并且路径$Path_i$包含在$Path_j$中。该方法没有考虑查询语句元素带有分支的情况。按照该匹配条件,当查询语句带有分支,进行匹配时,如果查询语句中含有分支的元素与数据对象中某个元素匹配,但该数据元素没有分支,则匹配继续进行。而实际上该数据元素并不是一个真正有用的匹配节点,这样将导致一些无效的匹配操作。另外,文[6]采用的后缀树结构无法得到现有商用DBMSs的支持,具体实现起来比较复杂。

*)863课题:网络环境的新一代中间件核心技术及运行平台(No. 2004AA112020);高性能网络服务器评测(No. 2003AA111020)。秦杰 博士、副教授,研究方向:Web数据库技术;杨树强 教授,研究方向:数据库技术。

实际上，基于 XML 的半结构化信息查询过程就是树(查询语句)与图(数据对象)的匹配过程。由于树型结构和图型结构的数据都可以用特定的字符串形式加以描述[8]，因此可以把 XML 数据以及 XML 查询语句转变成一定结构的字符串编码序列。这样，基于 XML 的数据查询就转化成为在字符串集合(数据对象)中寻找连续(或者非连续)的子字符串(查询语句)进行匹配的过程。基于这种考虑，本文提出了一种新的基于结构特征查询 XML 数据的方式：将 XML 数据中的每个基本单元(元素、属性、值)使用三元编码(Name, Path, Branch)方式加以表达。通过这种编码方式，可以将 XML 数据以及 XML 查询语句转化成三元组表示的字符串，XML 数据查询就转变成为在 XML 数据集所对应的三元组形式的字符串中寻找与 XML 查询语句所对应的三元组形式的字符串的匹配问题，从而可以采用传统的信息检索技术中常用的串匹配方式完成 XML 数据查询。这样采用串匹配方式执行 XML 数据查询时，即使是带有分支的 XML 数据查询也不再需要分解成若干子查询，从而避免了 JOIN 操作。另外，本文采用的三元编码方式还可以对 XML 数据的结构和内容建立统一的索引机制，从而进一步提高对树型结构 XML 数据查询的执行效率。

本文下面内容安排如下：第 2 节介绍相关知识，第 3 节给出基于结构特征编码的 XML 数据查询算法，第 4 节是试验结果，最后是结论。

2 预备知识

定义 1 单路径查询是指查询语句逻辑结构没有分支的查询。

定义 2 复杂查询是指查询语句逻辑结构至少包含一个分支的树型查询。

基于 XML 的半结构化数据查询主要有 4 种最基本形式：单路径查询、带通配符(*)的单路径查询、带“//”[1] 符号的单路径查询和带分支的查询。其中前 3 种通称为简单路径查询，最后一种及其与前三种查询的混合体通称为复杂查询。实际上，任何一个半结构化查询都是由上述基本查询形式组合而成的。

下面以一个简单的例子来说明本文使用的方法。图 1 是一条以树型方式表达的包含购买者和购买产品情况的商品记录。

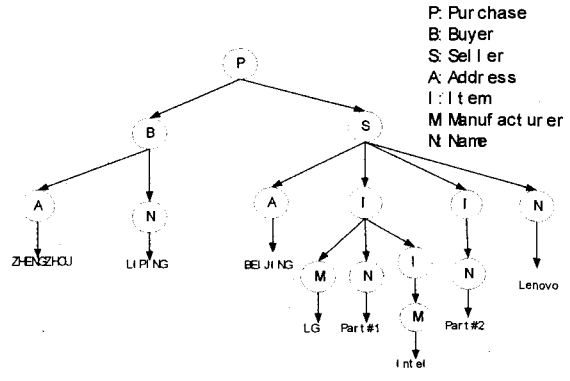

图 1 一条商品购买记录的树型表达

该记录对应 XML 文档的 DTD 为：

〈! ELEMENT purchases (purchase *)〉

〈! ELEMENT purchase (buyer, seller)〉

〈! ATTRIST buyer ID ID address CDATA name CDATA〉

〈! ELEMENT seller (item *)〉

〈! ATTRIST seller ID ID adderss CDATA name CDATA〉

〈! ELEMENT item (item *)〉

〈! ATTRIST item name CDATA manufacturer CDATA〉

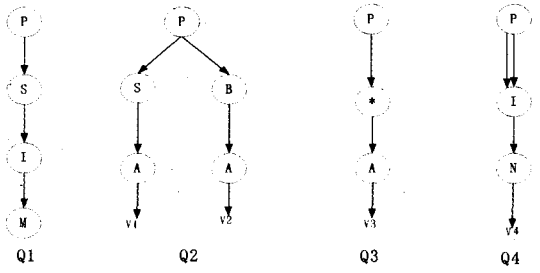

图 2 几种基本查询语句的树型表示

对于查询，

Q1：查找所有提供部件的制造商；

Q2：查找所有卖方地址在 v1、买方地址在 v2 的订单；

Q3：查找所有包含卖方地址为 v3，或者买方地址为 v3 的订单；

Q4：查找所有包含产品名称 v4 的订单。

图 2 列出了它们所对应查询的树型表达模式，分别是单路径查询(Q1)、带分支的查询(Q2)、带通配符(*)的单路径查询(Q2)、带“//”符号的单路径查询(Q4)。

一个 XML 文档可以表示成一棵 XML 文档树[5]。树型结构的 XML 文档可以用前序遍历的方式加以表达。对于图 1 中的商品购买记录，它对应的前序遍历序列为：

PBAv1Nv2SAv3 IMv4Nv5IMv6INv7Nv8 (1)

其中 V_1、V_2、…、V_8 分别代表数值 ZHENGZHOU、LIPIN、…、Lenovo。

由于同构树可能产生不同的前序遍历结果，这里规定：具有同一父节点的同层的兄弟节点按照其标示符的字典顺序自左至右排列；对于有同一父节点的同名兄弟节点之间自左至右排列的相对次序代表兄弟之间的先后顺序。显然，做出上述规定后，任意一个 XML 文档与它对应的前序遍历序列存在一一映射关系，因此(1)式的前序遍历序列就可以唯一地表示为

PBAv1Nv2SAv3 IIMv6Mv4Nv5INv7Nv8 (2)

式(1)、(2)的差别用下划线标出。为了实现基于串匹配原理的 XML 数据查询，需要对 XML 数据的结构信息进行标示。这里我们使用三元编码方式描述 XML 数据中的元素、属性和值。

定义 3 三元编码是一个三元组(Name, Path, Branch)，其中 Name 表示 XML 文档树中的一个节点，它可以是元素、属性的名称，也可以是元素、属性的具体值；Path 表示从文档树根到该节点的父节点路径；Branch 表示该节点孩子的数目。

[1] // 表示路径上连续 0 或多个任意的 XML 数据元素。

例如，在图1中节点B的三元编码是(B,P,2)。B是该节点元素的名称，P是从树根的B的父节点的路径；B的Branch值为2，表示该点有2个孩子(分支)。若Branch值为0，则表示该节点无分支。下文将会谈到对Branch值的判定，它是提高带分支查询语句效率的关键。

为了叙述简洁起见，本文下面的叙述中把元素、属性和值统称为元素。

定义4 三元编码序列是指树型结构的数据中所有元素用三元编码方式按照前序遍历方式排列组成的三元组字符串序列。

引理1 任何以XML表达的半结构化数据以及查询语句都可以用三元编码序列加以表示。

例如(2)式对应的三元编码序列为：

D=(P,ε,2) (B, P,2) (A, PB,1) (v1,PBA,0) (N,PB, 1) (v2, PBN,0) (S, P,4) (A, PS,1) (v3, PSA,0) (I,PS,3) (I,PSI,1) (M,PSII,1) (v6,PSIIM,0) (M, PSI,1)(v4,PSIM,0) (N, PSI,1) (v5,PSIN,0) (I,PS,1) (N, PSI,1) (v7; PSIN,0) (N,PS,1) (v8; PSN,0) (3)

查询语句Q1、Q2、Q3、Q4对应图的路径表达式和相应的三元编码序列分别如表1所示。

显然，在(3)式中有与查询Q1相应的三元编码序列相匹配的编码序列(对应(3)式中带下划线的序列)。同样，在(3)式中也可以找到与Q2、Q3、Q4所表达的三元编码序列相匹配的编码序列。这就从实例上说明了对XML数据的查询可以转化为字符串匹配的方式实现。

表1 查询语句Q1、Q2、Q3、Q4对应图的路径表达式和相应的三元编码序列

	路径表达式	相应的三元编码序列
Q1	/Purchase/Seller/Item/Manufacturer	(P,ε)(S,P)(I,PS)(M, PSI)
Q2	/Purchase [Seller [add = v1]]/Buyer[add=v2]	(P,ε,2)(S,P)(A,PS)(v1, PSA,0) (B, P)(A,PB)(v2, PBA,0)
Q3	/Purchase/ * [add=v5]	(P,ε)(A, P *)(v5, P * A,0)
Q4	/Purchase//Item [Manufacturer=v3]	(P,ε)(I, P//)(M, P//I)(v3, P//IM,0)

用字符串匹配的方式实现对XML数据查询的主要目的是避免查询语句的拆分，从而避免开销较大的JOIN操作，提高查询效率。

3 基于结构特征的XML数据查询算法

3.1 基于主存的串匹配算法

按照前面所述的三元编码方式，将XML数据转换成串的形式。对于没有分支的简单路径查询，将它对应的XML查询语句也转换成串的形式。这时，查询就相当于在一个长串(XML数据)中寻找一个特定的子字符串(XML查询语句)的过程。因此，使用传统的字符串匹配算法，如KMP算法、BM算法[9]等，就可以完成查询任务。当然，XML数据查询需要解决的关键问题是对含有分支条件的查询语句的处理。这里重点研究具有分支的查询。图3给出了基于串匹配原理的XML数据查询算法MatchSearch。算法的重点就是对查询语句的分支进行判断、处理。

```
Input: Q=q1…qn, S=S1…Sm,
Output: S中所有匹配Q的结果
MatchSearch(S→root,1);
Function MatchSearch(m, i)
  If i ≤n
    then
  for each node j is a descendent of node m do
    if (Namei = Namej) and (Pathi ≤ Pathj) and (Branchi ≤
      Branchj)//sj 匹配qi
      then MatchSearch(j, i+1)
      else
        if i>n
        then
        Output matched results;
        i=1;
      end
end
```

图3 XML数据查询算法MatchSearch

其中，Q是以三元编码形式表示的查询序列$q_i=(Name_i,Path_i,Branch_i)$；$S$是以三元编码形式表示的数据序列$S_j=(Name_j, Path_j, Branch_j)$。

算法从查询语句的第一个元素开始与查询数据对象的第一个元素进行匹配。假设在数据节点X已经匹配$q_1\cdots q_{i-1}$，为了匹配下一个元素q_i，就需要从X的所有后继中寻找与q_i相匹配的元素。查询语句的第i个元素与数据对象中的第j个元素匹配的条件是：两者名称相同($Name_i=Name_j$)，同时从查询语句的根到第i个元素的路径包含在从数据对象的根到第j个元素的路径中($Path_i\leqslant Path_j$)。另外，还要求元素i对应节点的分支数不大于被查询数据对象中相应元素j节点的分支数($Branch_i\leqslant Branch_j$)。

算法的核心是如何寻找下一个匹配因素。这里采用KMP算法[9]完成这一操作。KMP算法的最大特点是指示字符串的指针不需回溯，整个匹配过程中，对数据串仅需从头至尾扫描一遍，因此该算法的时间复杂度为$O(m+n)$。其中m为数据串长度，n为查询语句对应的串长度。

3.2 引入索引的匹配算法

MatchSearch算法是一种基于主存的串匹配算法。如果查询的数据对象文件大小大于主存，该算法则无法应用。另外，在使用该算法时，当我们在数据对象中寻找到一个匹配节点X后，为了寻找下一个匹配节点Y，必须从X的后续序列中逐一查找。实际上通过建立相应的索引，可以直接找到Y。利用关系数据库技术提供的索引机制可以进一步提高查找效率，同时查询的数据文件的大小也不再受主存的限制。下面考虑建立索引的问题。

采用上述三元编码方式后，串中数据排列位置存在两种关系：逻辑上的祖先-后继(A-D)关系(如(3)式中的元素(B, P,2) 与 (v1,PBA,0))、排列顺序上的先后(B-A)关系(如(3)式中的元素(B, P,2) 与(S, P,4))。显然，具有先后关系的元素并不一定具有祖先-后继关系，反之亦然。在查询过程中，当已经找到与q_{i-1}相匹配元素X，需要在X的后继中寻找与q_i相匹配元素Y时，可能有时还需要元素X与元素Y满足A-D关系，由此可以得到下面定理。

定理1 对于以三元编码序列表示的XML数据S和查询语句Q，当Q的第$i-1$个元素q_{i-1}已经在S中找到与之相匹配元素X时，则有：(1)与q_i相匹配元素Y必然在X的后继中；(2)当q_{i-1}是q_i的祖先时，元素X与元素Y之间必须同时满足A-D关系和B-A关系。

证明：对于查询语句中的任意两个元素q_{i-1}和q_i来说，两者之间存在两种位置关系：q_{i-1}是q_i的祖先；q_{i-1}与q_i位于

不同的分支。

(1) 当 q_{i-1} 与 q_i 位于不同的分支时，在对应的数据对象中，如果 q_{i-1} 已经找到与之相匹配元素 X，由于数据元素是按照先序遍历方式排列的，则与 q_i 匹配的元素 Y 必然在 X 的后继中。

(2) 当 q_{i-1} 是 q_i 的祖先时，如果 X 与 q_{i-1} 匹配，则与 q_i 匹配的元素 Y 必须是 X 的子孙，故此时元素 X 与元素 Y 之间必须同时满足 A-D 关系和 B-A 关系。 □

对于数据序列中的任意两个元素，通过检查这两个元素的 Path，可以判断它们之间是否存在逻辑上的 A-D 关系。然而，要确定它们之间是否具有排列上的 B-A 关系，还需要增加一些信息。这里为数据序列中的每个元素 X 增加一个二元标示符$\langle n_X, size_X\rangle$，其中 n_X 是 X 在三元编码序列中排列的顺序号，$size_X$ 是 X 在三元编码序列具有的后继元素的数目。这种二元标示符可以通过对数据元素进行深度优先遍历获得。例如序列(3)对应的二元标示符序列为：

(P,ε,2)〈1,21〉(B, P,2)〈2,20〉(A, PB,)〈3,19〉(v1, PBA,0)〈4,18〉(N,PB)〈5,17〉(v2, PBN,0)〈6,16〉(S, P, 4)〈7,15〉(A, PS,)〈8,14〉(v3, PSA,0)〈9,13〉(I,PS,3)〈10, 12〉(I,PSI)〈11,11〉(M,PSII)〈12,10〉(v6,PSIIM,0)〈13,9〉 (M, PSI)〈14,8〉(v4,PSIM,0)〈15,7〉(N, PSI)〈16,6〉(v5, PSIN,0)〈17,5〉(I,PS)〈18,4〉(N, PSI)〈19,3〉(v7; PSIN, 0)〈20,2〉(N,PS)〈21,1〉(v8; PSN,0)〈22,0〉

通过引入上述二元标示符，在索引中可以方便地确定任意两个元素之间的 B-A 关系。如果元素 X 的二元标示符为$\langle n_X, size_X\rangle$，元素 Y 的二元标示符为$\langle n_Y, size_Y\rangle$，则有：

定理 2 X 与 Y 满足 B-A 关系 iff $n_Y \in (n_X, n_X + size_X)$。

该定理的正确性显然，证明从略。

下面介绍建立索引的方法。

首先，使用 B^+ 树方式对数据对象中的所有元素以(Name, Path)为关键字建立索引。通过该索引可以在数据对象中直接找到那些与已经匹配的元素 X 满足 A-D 关系的匹配元素集合{Y}。当然，X 与{Y}中的一部分(或者全部)之间可能不满足实际排列顺序上的 B-A 关系，应当把这部分元素排除。因此，对于有相同(Name, Path)的元素，再使用它们各自的 n_X 建立 B^+ 树，从而就可以确定元素之间的 B-A 关系。

这样建立的索引结构包括两级 B^+ 树：以(Name, Path)为关键字、用于确定元素之间 A-D 关系的 B^+ 树；以 n_X 为关键字、用于确定元素之间 B-A 关系的 B^+ 树。

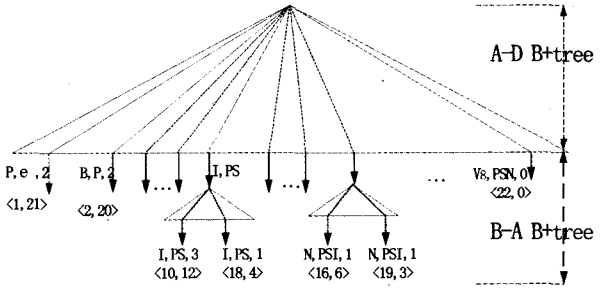

图 4 三元编码序列 D 对应的索引结构

这里将以(Name, Path)为关键字建立的 B^+ 树索引称为 A-D 树，以 n_X 为关键字建立的 B^+ 树索引称为 B-A 树。图 4 是对应于图 2 的索引结构。这样，通过建立两级 B^+ 树，就实现了对 XML 数据结构和内容的索引，避免了对半结构化数据的结构和内容分别建立索引，使得索引结构相对简单。

建立了上述索引结构之后，可以这样实现查询：假设在数据节点 X 已经匹配查询序列 $q_1 \cdots q_{i-1}$，为了寻找下一个与查询元素 q_i 相匹配的元素 Y，首先使用 q_i 的(Name, Path)作为关键字在 A-D 树中搜索。如果有满足条件的元素，将会得到一个指向($Name_Y$, $Path_Y$)的指针，该指针指向的是一棵以($Name_Y$, $Path_Y$)为根的 B-A 树，这棵 B-A 树中包含了所有与上一个匹配元素 X 之间满足 A-D 关系、可能与 q_i 匹配元素的集合{Y}；由于 X 与集合{Y}中的部分元素之间可能不存在 B-A 关系，需要将这类元素从{Y}中剔除。在 B-A 树中执行一个基于 $n_X < n_Y \leqslant n_X + size_X$ 范围的查询，从而找到 X 的真正后继{Y}。另外，在寻找满足条件的 Y 时，如果 q_i 的 Branch 项大于等于 1，则那些满足 $Branch_i \leqslant Branch_y$ 的 Y 才是真正匹配 q_i 的元素，因此，还需要在 x 的真正后继{Y}中寻找 Branch 项大于等于 $Branch_i$ 的 Y。同理可以继续寻找匹配 q_{i+1} 的元素。

图 5 给出了在这样的索引结构下设计的基于串匹配原理的查询算法。

```
Input: Q=q1…qn.;
    数据 S 以(Name, Path)为关键字的 B+树索引;
    //A-D 树
    以 nX 为关键字的 B+树索引;//B-A 树
Output: S 中所有匹配 Q 的结果;
BMS (S→root, 1);
Function BMS (m, i)
 1  If i≤n
 2    then
 3      if qi='//' then i=i+1;
 4      以(Namei, Pathi)为关键字对 A-D 树检索,将查到
          的结果赋给 T; // T 是指向 B-A 树根的指针
 5      while T≠nil do
 6        在T 所指的 B-A 树中,寻找在(nm,nm+size)内
            的元素,并将结果赋给 N;
 7        for each node j∈N do
 8          if Branchi≤Branchj
 9            then BMS(m+1, i+1)
10        end;
11        T=T.NEXT;
12      end
13    else
14      if i>n
15       then
16         Output matched results;
17         i=1;
18      end
19  end
```

图 5 引入索引之后的查询算法 BMS

算法对于通配符'*'和'//'的处理：如果在查询序列中包含'*'，则在执行算法的语句 4 时，'*'可以与任何一个元素匹配。这时可能有多个 B-A 树满足条件，语句 4 将返回一个指针数组 T，T 的每一个元素指向一个 B-A 树的根。这时如果'*'的 Branch 项为 0(*为叶节点)，只需要依次对这

些满足条件的 B-A 树执行语句 5，之后跳到语句 10 继续执行。如果‘*’的 Branch 项不为 0(*非叶节点)，则算法顺序执行。这样就实现了对通配符‘*’的查询处理。

当在查询序列中包含‘//’时，算法通过语句 3 直接跳过‘//’，寻找与‘//’的后继匹配的元素。这样就实现了对通配符‘//’的查询处理。

算法复杂性分析：设 XML 数据长度为 M，查询语句数据长度为 N。算法的核心是如何寻找下一个匹配因素，算法通过两次索引完成这一任务。首先通过语句 4 对满足 A-D 关系的 B^+ 树进行索引，需要时间为 $O(\log_2 M)$，之后对 B-A 树进行基于范围的查询，以寻找满足条件的匹配，需要时间为 $O(\log_2 M)$，因此查找一个匹配的时间为 $O(\log_2 M)$。由于查询语句数据长度为 N，故寻找一个满足条件的查询结果需要的时间为 $O(N \log_2 M)$，则由 KMP 算法可知，在对于长度为 M 的 XML 数据中寻找所有匹配需要的总时间为 $O(M+N \log_2 M)$。

由于 A-D 树以及 B-A 树索引占用空间均为 $O(M)$，故算法使用的索引占用空间为 $O(M)$。

4 性能测试

为了验证算法的实际性能，我们实现了 BMS 算法，并与 XISS 方法、Index Fabric 方法以及 RIST 算法进行了对比实验。实验环境：CPU 为 AMDXP1600+，内存 256M，Linux 操作系统，通过 Berkeley DB library[10] 提供的 B^+ 树 API 建立索引。选择了 3 个测试数据集进行比较测试：DBLP[11]，XMARK[12]，以及自制的数据集。

针对 DBLP 和 XMARK 进行的对比实验使用表 2 所示的查询语句。使用的 DBLP 大小为 30MB，包含 289627 条记录、2934188 个元素以及 364348 个属性；XMARK 数据集大小为 108M。

表 2 测试使用的查询语句及对应的数据集

	路径表达式	数据集
Q1	/inproceedings/title	DBLP
Q2	/book/author[text='David']	DBLP
Q3	//author[text='David']	DBLP
Q4	/book[key='books/bc/MaierW88']/author	DBLP
Q5	/site//item[location='US']/mail/date[text='02/18/2000']	XMARK
Q6	/site//person/*/city[text='Pocatello']	XMARK

从表 3 中可以看出，在执行带有分支的复杂查询时，基于串匹配的查询方式明显优于传统的基于连接的查询方式，进而也从实验上证明了 JOIN 操作在查询过程中的代价的确较高。

表 3 BMS 与 RIST、Index Fabric、XISS 查询性能比较

	BMS	RIST	Index Fabric	XISS
Q1	1.8	1.2	0.8	9.7
Q2	1.7	2.4	4.6	53.2
Q3	1.7	1.9	22.1	29.7
Q4	1.7	2.8	7.5	18.1
Q5	2.3	3.7	17.2	22.1
Q6	2.4	2.5	36.3	26.9

另外，在自制的数据集上分别使用 BMS 算法和 RIST 算法进行依次包括 2,4,6,8,10 个元素的查询语句测试。除了第一个查询之外，其余每个查询均含有分支。这里使用数据生成器产生的综合数据集是一个高度为 8、每个节点包含 10 个子节点的满树。这棵树共有 11,111,111 个节点，每个节点用随机分配的 26 个英文字母之一表示节点名称。

图 6 是实验结果，图中横轴表示查询语句中包含的元素数目，纵轴为查询使用的时间(单位为秒)。从图 6 中可以明显看出 BMS 算法的性能好于 RIST 方法，这是因为 RIST 方法在查询过程中没有对匹配的中间元素是否有分支进行判断，从而导致一些无效的匹配操作，消耗了处理的时间。

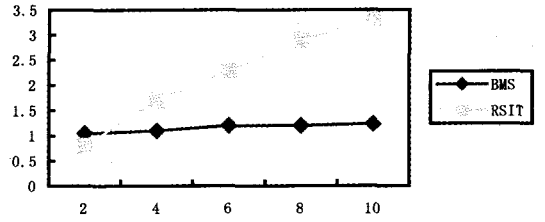

图 6 BMS 算法和 RIST 算法的执行结果比较

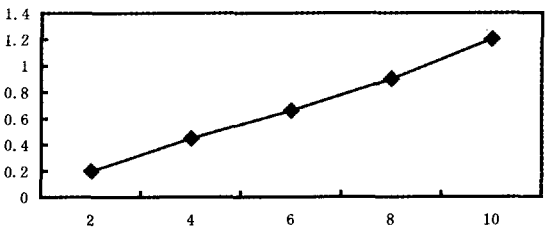

图 7 随数据集大小的变化 BMS 算法的执行时间

图 7 给出了包括 6 个元素的带分支的查询语句。在数据集大小发生变化时，使用 BMS 算法得到查询执行时间。图中横轴表示数据集的大小(数据集分别含有 2×10^6、4×10^6、6×10^6、8×10^6、1×10^7 个元素)，纵轴为查询使用的时间(单位为秒)。另外，使用 BMS 算法，在同样环境下，当查询语句分别含有 2,4,8,10 个元素时，得到的结果与图 7 基本一致。

通过上述实验，验证了 BMS 算法具有较良好的扩展能力，以及与同属于串匹配方法的 RIST 算法相比，具有更好的查询性能。

结束语 采用结构编码方式实现 XML 数据查询的一个明显优势是：把查询语句当作一个整体处理，而不需要像通常的做法那样，把一个复杂的查询分解成若干查询单元，分别执行查询，之后再通过 JOIN 操作获得最终结果，使得查询效率明显提高。本文提出的采用三元组形式描述 XML 数据的方法明确给出了 XML 数据元素的结构特征，可以充分利用现有的信息检索技术，采用类似字符串匹配的方式实现 XML 数据的查询。尤其在执行带有分支条件的查询时，这种方法的查询效率更高。

参 考 文 献

1 Li Q, Moon B. Indexing and querying XML data for regular path expressions. In: VLDB, September 2001. 361～370

2 Cooper B F, Sample N, Franklin M, et al. A fast index for semi-structured data. In: VLDB, September 2001. 341～350

3 Kaushik R, Bohannon P, Naughton J, et al. Covering indexes for branching path queries. In: ACM SIGMOD, June 2002

4 Deutsch A, Fernandez M, Florescu D, et al. A query language

for XML. In: Proceedings of the 8th International World Wide Web Conference, May 1999. 77～91
5 Bruno N, Koudas N, Srivastava D. Holistic Twig Joins: Optimal XML Pattern Matching. In: ACM SIGMOD, June 2002
6 Wang H X, Park S, Fan W, et al. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In: ACM SIGMOD, June 2003
7 McCreight E M. A space-economical suffix tree construction algorithm. Journal of the ACM, 1976, 23(2): 262～272
8 Shasha D, Wang J T L, Giugno R. Algorithmics and Applications of Tree and Graph Searching. In: ACM Symposium on Principles of Database Systems(PODS), May 2002. 39～52
9 van Leeuwen J. Algorithms for finding patterns in strings. In: Handbook of Theoretical Computer Science. Vol A, Algorithms and complexity. Chapter 5. Elsevier, Amsterdam, 1990. 255～300
10 Sleepycat Software. http://www.sleepycat.com. The Berkeley Database (Berkeley DB)
11 Ley M. DBLP database web site. http://www.informatik.uni-trier.de/ ley/db, 2004
12 XMARK: The XML-benchmark project. http://monetdb.cwi.nl/ xml, 2004

(上接第 91 页)

定义 17 回答用户目标是指自然查询句要求系统回答的内容。回答用户目标可能是比较判断目标，也可能是逻辑推理目标或者直接查询目标。

回答用户目标用回答用户目标队列来表示。

3 查询目标关系

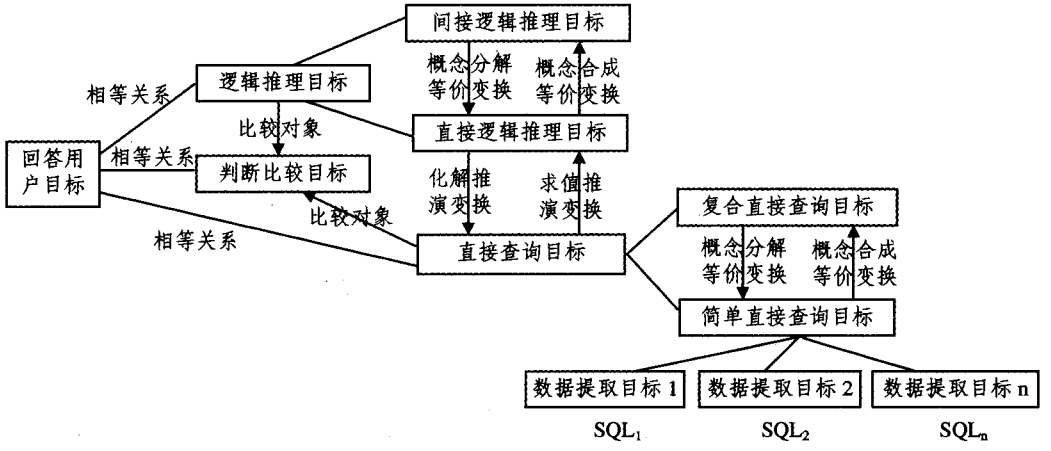

图 1 查询目标关系图

查询目标中各阶段目标的关系如图 1 所示。在实际中，许多查询句的查询目标只包括三个阶段中的部分阶段，分如下几种情况：

(1)当回答用户目标＝直接查询目标时，直接查询目标的内容就是回答用户目标的值。如：请说出张三的籍贯(祈使句)，哪里是张三的老家？(特殊疑问句)。

(2)当回答用户目标＝逻辑推理目标时，逻辑推理目标的内容就是回答用户目标的值。如：请说出张三的退休日期(祈使句)，张三哪年退休？(特殊疑问句)。逻辑概念“退休日期”的值就是回答用户目标的值。

(3)当回答用户目标＝判断比较目、只有一个比较者和一个被比较者(是非问句或正反问句)、比较者是直接查询目标或逻辑推理目标、被比较者为常量或直接查询目标或逻辑推理目标时，则回答用户目标的值是二者比较结果的逻辑值。如：张三是副教授吗？张三与李四的职称一样吗？张三与李四的退休年龄一样吗？

(4)当回答用户目标＝判断比较目、只有一个比较者和多个被比较者(选择问句)、比较者是直接查询目标或逻辑推理目标、被比较者均为常量时，则回答用户目标的值是比较结果为真的对应的被比较者。如：张三是讲师还是副教授？

(5)当回答用户目标＝判断比较目、只有一个比较者和多个被比较者(选择问句)、比较者是直接查询目标或逻辑推理目标、被比较者均为直接查询目标或逻辑推理目标时，则回答用户目标的值是比较结果为真的对应的被比较者概念对应的查询条件块。如：张三的职称是与李四一样还是与王五一样？

结论 本文深入研究了汉语查询句中查询目标信息，用祈使句或特殊疑问句查询时，回答用户目标一般等于直接查询目标或逻辑推理目标；用是非问句、正反问句、选择问句等方式查询时，回答用户目标一般等于判断比较目标。这些信息作者都设计了存储表示结构，可已用计算机软件来识别和转换，识别和转换的算法将于另文讨论。

参 考 文 献

1 郑逢斌. 计算机理解自然查询语言的研究与实现[D]:[西南交通大学博士研究生学位论文]. 2004
2 孟小峰，王珊. 中文数据库自然语言查询系统 Nchiql 设计与实现[J]. 计算机研究与发展，2001，38(9)：1080～1086
3 王英姿，宗成庆，陈肇雄，黄河燕. ITS 系统中自然语言人机接口的设计与实现[J]. 计算机研究与发展，1998，35(9)：814～818
4 许龙飞，杨晓昀，唐世渭. 基于受限汉语的数据库自然语言接口技术研究[J]. 软件学报，2002，13(4)：537～544
5 许龙飞，唐世渭. 数据库汉语自然语言查询模型研究[J]. 计算机科学，1999，26(8)：43～46
6 许龙飞. 数据库自然语言查询技术研究[J]. 计算机科学，1997，24(5)：50～54
7 卞世力，姚天顺，金鸿. 一个中间语言生成目标语言的原理和方法[J]. 软件学报，1994，5(9)：1～8
8 李保利，周锡令. 数据库自然语言接口系统的研究[J]. 计算机系统应用，1999(12)：31～34