论文题目：XML 数据的 twig 模式查询匹配算法研究

专　　业：计算机软件与理论

硕 士 生：雷欣

指导教师：高集荣 副教授

摘 要

近年来，随着数据库和网络技术的发展，XML 已经成为 Internet 上数据表示和交换的标准。随着 XML 技术的不断普及，Internet 上以 XML 技术作为载体的数据越来越多，从而对这些 XML 数据的有效管理和查询也越来越受到国内外研究者的关注。

对于 XML 文档的查询，目前提出的算法主要是基于如下两种思想：第一种是路径分解思想，这种方法将产生大量的不可避免的无用的中间结果。另一种是新近提出的整体小枝(holistic twig)模式查询匹配方法，它把用树结构建模的查询表达式——twig 模式(twig pattern)作为一个整体来处理。这种方法往往与一些特殊的编码和索引技术相结合，避免了大量不必要数据节点的扫描，使得算法的 I/O 代价、CPU 时间复杂度和空间复杂度大大降低，从而提高了查询效率。自从 Bruno N 等人于 2002 年提出 holistic twig 概念以来，研究者们已经提出了一系列 twig 模式查询匹配算法。其中 TJFast 算法基于 Extended Dewey 编码，只要访问 twig 模式中的叶子节点的输入集合流就可以有效地处理 XML 文档查询，是一种效率较好的 holistic twig 模式查询匹配算法。但是 Extended Dewey 编码不支持 XML 文档的动态更新操作，且 TJFast 算法设计思想上的缺陷也使得算法执行效率上可以进一步提高。

本文在总结和分析了主流的 XML 文档编码方案的基础上改进了 Extended Dewey 编码，使其能有效支持 XML 文档的动态更新操作。提出了一种新的基于新 Extended Dewey 编码的 twig 模式查询匹配算法——TwigMatch 算法，进一步提高了 twig 模式查询匹配算法的效率。该方法分三个步骤来处理一个 twig 查询模式，大大减少了查询路径匹配操作，提高了查询效率。同时本文还重组了传统的两阶段算法，恰当地选择中间结果的归并时机，获得了较好的内存利用率。

多组实验数据对比表明，本文的方法在效率上有较大的提高。

关键字： XML 文档、twig 模式查询匹配、XML 编码、Holistic twig 算法、Extended Dewey 编码

Title: Research on Twig Pattern Query Matching Algorithm for XML Data

Major: Computer Software and Theory

Name: Xin Lei

Supervisor: Jirong Gao Associate Professor

Abstract

In recent years, with the development of database and network technology, XML has become the standard of data representation and exchange on the Internet. With the ever-growing popularity of XML technology, there are more and more data which employs XML as the vector in the Internet, so many domestic and foreign scholars pay more and more attentions to the XML data management and query.

For the query in XML document, the algorithm which are already given at home and abroad nowadays are mainly based on the following two ideals: The first is to decompose the query expression to several simple paths expression. This method cannot avoid large amount of useless intermediate results. The other method is a newly proposed holistic twig query pattern matching approach, which processes the twig pattern as a whole. The method usually combines with some special encoding and indexing technology, avoids large amount of unnecessary scanning of nodes, reduces the I/O cost、CPU complexity and space complexity, improves the efficiency of query. From the concept of holistic twig was proposed by Bruno N in 2002, the researchers have proposed a series of twig pattern query matching algorithms. The TJFast algorithm bases on Extended Dewey encoding, only accesses the input stream of leaf nodes in the twig pattern, and processes the XML document query effectively. It is a kind of holistic twig pattern query matching algorithm with better efficiency. But the Extended Dewey encoding fails to support the dynamic update operation in the XML document, and the TJFast algorithm has backdrop in the design also makes it possible to increase the efficiency of execution.

In this paper, the Extended Dewey encoding was improved, it was made to support the dynamic update operation of XML document. A novel twig pattern query

matching algorithm named TwigMatch was also proposed, which is based on the improved Extended Dewey encoding, the efficiency is improved. It deals a twig query pattern in three steps, reduces the quantity of single path comparison, improves the query efficiency. At the same time, the traditional two phases algorithm was reconstructed, chose the right time for the merge operation, and gained better memory utilization.

The relative analysis of multiunit experimental data proved that the method in this paper has improvement in the efficiency.

Keywords: XML document, twig pattern query matching, XML encoding, holistic twig algorithm, Extended Dewey encoding

论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：雷欣

日　　期：2009.5.20

学位论文使用授权声明

本人完全了解中山大学有关保留、使用学位论文的规定，即：学校有权保留学位论文并向国家主管部门或其指定机构送交论文的电子版和纸质版，有权将学位论文用于非赢利目的的少量复制并允许论文进入学校图书馆、院系资料室被查阅，有权将学位论文的内容编入有关数据库进行检索，可以采用复印、缩印或其他方法保存学位论文。

学位论文作者签名：雷欣　　导师签名：

日期：09年5月20日　　日期：09年5月22日

第1章 绪论

近年来，随着数据库和网络技术的发展，XML[1](eXtensible Markup Language, 可扩展标记语言)已经成为 Internet 上数据表示和交换的标准。随着 XML 技术的不断普及，Internet 上以 XML 技术作为载体的数据越来越多，从而对这些 XML 数据的有效管理和查询也越来越受到国内外研究者的关注。

1.1 研究背景

随着 Internet 和信息技术的蓬勃发展，XML 被越来越多地运用于数据交换和数据存储领域的方方面面，已成为 Internet 上信息交换和表示的重要标准。XML 技术在改变着人们的生活、学习和工作方式，拓宽了人们获得知识和信息的途径，并且改变了人们的思维习惯。它是一种特殊的半结构化数据，以其自描述性、支持文档内容的验证和允许开发各种不同领域特定标记语言等优点弥补了 HTML[2](Hypertext Markup Language,超文本标记语言)的诸多缺陷，同时还具备更好的规范性、灵活性、可扩展性和语言表达能力，在替代了传统的 HTML 的同时还具有更广阔的应用领域，如数字图书馆、电子商务、web 服务等。XML 技术出现至今，受到了业界的广泛关注。多家知名厂商都参与到了 XML 各项标准的制定和完善中来，同时各厂商的主流商用产品都提供了对 XML 的支持，加速了 XML 技术的商业化应用。在学术科研领域，从 1998 年至今，已有多家知名研究机构和高校致力于 XML 数据库技术的研究，在提高 XML 数据的存储、传输和查询等方面做了很多有益的探索，取得了很多研究成果，并成功转化到商用领域，创造了很好的商业价值和社会效益。

XML 技术相关的主要研究方向有 XML 的查询数据模型、查询语言和查询代数的理论研究，XML 数据的编码方案和索引结构研究，XML 查询处理和查询优化以及纯 XML 数据库系统的研究等[3]。随着近年来 Internet 上 XML 数据的高速增长，XML 数据的查询处理和优化已经成为 XML 技术相关的众多研究领域

中较为关键的研究热点。在查询语言方面，研究者们针对 XML 数据特有的树状结构，提出了多种 XML 查询语言，如 XML-QL[4]、XPath[5]和 XQuery[6]等。尽管这些语言各具特点，但是它们都是针对 XML 数据的结构特点，以路径表达式作为核心，通过路径表达式来描述 XML 文档树中的结构信息。因此，XML 查询处理和优化的最关键问题就在于路径表达式的查询处理。

对于路径表达式的处理，最直接和简单的方法就是对 XML 文档树进行从根结点开始沿着文档树各边到各个结点的遍历，在遍历过程中寻找匹配路径表达式的目标结点。这种方法虽然简单直观，但是效率低下也是显而易见的。为此，研究者们提出了许多新的算法来提高路径表达式查询匹配的处理效率。主要可以归纳为以下三类：结构连接方法[7-12]、基于路径字符串匹配的方法[13,14]和 twig 模式查询匹配方法[15-18]。其中 twig 模式查询匹配方法是新近的研究成果，总体上较前两者具有较高的效率。

1.2 国内外研究现状

近年来，XML 作为 Internet 上数据表示和交换的事实上的标准，正在得到广泛的认可和应用。随着 XML 应用的不断普及和相关研究的不断深入，关于 XML 数据处理的研究正逐渐成为研究热点。国外诸多高校和科研机构都在致力于该领域的研究工作，其中最具代表性的 Stanford 大学和 Bell 实验室等都在该领域的研究中取得了突出的成果。国内的中国人民大学、复旦大学等[19-21]也有研究人员在该领域进行了大量卓有成效的工作。另外，近年来国际重要的数据库学术会议(如 SIGMOD、ICDE 和 VLDB 等)都收录了大量关于 XML 数据库方面的论文，并且都为该领域开设了相应的专题，该领域的学术研究和交流非常活跃。

对于 XML 数据查询处理和优化的核心问题——路径表达式处理，越来越受到研究者们的青睐，成为了研究的热点之一。早期的传统方法是直接对 XML 文档树进行遍历操作，从文档树的根结点出发，按照遍历顺序沿着树中的边寻找匹配路径表达式的目标结点。这种简单直观的方法不仅效率非常低，而且有时需要对整棵树进行遍历。尽管该方法由于其本质上的缺陷，不可能在效率上取得较大

的提高，研究者们还是进行了大量有益的探索，提出了自顶向下、自底向上和两者结合的基于导航的遍历方式以及借助模式信息和索引的优化遍历方法。

随着研究的深入，诸多研究者们先将一个复杂的路径表达式分解为若干个简单的路径表达式，再将简单路径表达式的匹配转化成结点之间的结构连接操作，最后将各个简单路径表达式的匹配结果合并成对应复杂路径表达式的匹配结果。这样，对于一个复杂路径表达式的匹配操作就转化为了若干次两个结点列表间的包含关系或者文档位置关系的结构连接操作。于是，高效的结构连接算法就成为研究者们关注的热点。目前在这方面的研究上国内外都提出了一系列有效的结构连接算法。文献[7]于 2001 年提出了 MPMGJN 算法，Zhang 等人在查询处理中引入了区间编码，并提出该多谓词归并连接算法，该算法在某些情况下存在多次重复扫描后代节点列表并产生大量中间结果的情况，因此效率较低。文献[8]于同年提出了 EE-Join/EA-Join 算法，但也存在 MPMGJN 算法同样的重复扫描后代列表的问题。文献[9]在索引定位技术、短路技术和预侦测技术的基础上提出了 IIMGJN 算法，有效地避免了不必要的重复扫描和搜索，大大减少了连接代价。后来的研究者们提出了基于缓存的归并结构连接算法 Stack-Tree 算法[10]，通过维护一个缓存栈，该算法只需要分别扫描祖先列表和后代列表各一次就可以完成连接操作。文献[11]沿着前面的思路，利用 B+树索引跳过许多可以事先判断并不参与连接的元素节点，进一步提高了结构连接的性能。文献[12]中提出的 PC-Join/Holding-Join 算法也可以看作是前面算法思路的延伸，该算法通过在每个元素节点的编码中加入其双亲结构信息，从而在连接时根据双亲结构信息并利用 B+树索引尽可能多地跳过后代列表中的不参与连接的结点，进一步提高了连接效率。

与此同时，研究者们从另一个思路来研究 XML 路径表达式的处理，即基于路径字符串匹配的查询处理方法。该类方法的关键在于分别将查询模式树和 XML 文档编码成字符串，按照某种遍历顺序和编码方案先将查询模式树转换成路径字符串，再将 XML 文档中的各结点按照某种方式编码并按对应顺序连接成字符串，最后通过字符串匹配和验证后输出最终结果。显然，该类方法需要进行大量的字符串模式匹配和复杂的验证操作，效率较低。该类方法的代表有 PRIX 算法[13]和 Vist 算法[14]。

文献[15]中 Bruno N 等对一个 XML 路径表达式中包含的多个祖先/后代关系结构连接进行整体考虑，提出了整体 twig 连接(holistic twig join)的思想和 TwigStack 算法。该算法将整个查询模式树作为一个整体来考虑，通过为每个查询节点维护一个祖先节点栈，TwigStack 算法仅仅需要对所有参与连接的输入数据结点列表分别顺序扫描一遍就可以实现整个 XML 路径表达式的计算，不需要将路径表达式分解成若干个结构连接操作，减少了输入数据结点列表的扫描的同时也减少了对中间结果的多次存取，从而获得更优的 I/O 性能。在 twig 概念以及整体 twig 连接的思想提出后，引起了研究者们的广泛关注。研究中们先后提出了新的 twig 模式查询匹配算法。在 TwigStack 算法提出不久之后，Jiang 等在文献[16]中结合 XR-Tree 索引[22]的思想，提出了 TSGeneric+算法，它能够跳过许多可以事先判断并不参与连接的数据结点。Lu J 等在文献[17]中对前缀编码 Dewey 编码进行扩展，提出了一种扩展 Dewey(Extended Dewey)编码，结合该编码以及有限状态自动机(FST)技术，他们提出了 TJFast 算法，该算法只需要访问 twig 查询模式树中的叶子节点，创新之处在于其编码可以通过一个 FST(Finite State Transducer,有限状态自动机)在需要时才计算出某个数据结点对应的从 XML 文档树的根结点到该数据结点自身的路径信息，节省了存储文档路径索引空间的同时也降低了路径表达式匹配时的 I/O 代价。但 TJFast 算法过分依赖于以上特性，进行了大量不必要的查询路径匹配操作，即大量的字符串模式匹配操作，耗费了较多的时间。同时，在求解叶子节点的公共分支时，没能利用到前缀编码的特性，也做了过多不必要的字符串比较操作。

1.3 本文研究内容

文献[17]中提出的 TJFast 算法在当前提出的一系列 twig 模式查询匹配算法中，是较高效的算法，其提出的 Extended Dewey 编码不仅具备前缀编码的特性，可以包含一个数据结点的所有祖先结点的编码信息，同时还可以在耗费较小的时空开销的情况下计算出该结点的所有祖先结点的标签名称，从而得到从根结点到自身的路径信息。为了克服 Extended Dewey 编码不支持 XML 文档的动态更新操

作的缺陷，本文借鉴了 LSDX 编码方案[23]的思想，对 Extended Dewey 编码进行了相应的改进，使之支持 XML 文档的更新操作而不需要对整个 XML 文档重新编码。

为了克服目前提出的一些 twig 模式查询匹配算法的缺陷，减少不必要的数据结点比较和查询路径匹配操作，本文提出了一种新的 twig 模式查询匹配算法。本文的主要工作可以归纳如下：

1. XML 文档的编码方案研究。本文总结了主流的 XML 文档编码方案，分析了各种编码方案的优点和缺点。
2. XML 文档的 twig 模式查询匹配算法研究。本文分析了在 Bruno N 等人提出整体twig模式查询概念以来研究者们提出的的多个twig模式查询匹配算法的思想及执行过程。
3. UED XML 文档编码方案的研究。文献[17]中提出的 Extended Dewey 编码在具备前缀编码特性的同时还可以通过一个有限状态自动机 FST 实时计算每个数据结点的所有祖先结点的名称信息。本文总结了几种主流编码方案，并借鉴 LSDX 编码方案的思想，对 Extended Dewey 编码进行了相应的改进，保留原有特性的同时，使之支持 XML 文档的动态更新操作而不需要对整个 XML 文档重新编码，有效降低了二次编码率。
4. 提出了一个新的 twig 模式查询匹配算法——TwigMatch 算法。该方法的输入包括 twig 模式中最顶层分支节点和各叶子节点的输入集合流，分三个步骤来处理一个 twig 查询模式，先进行从 twig 模式中最顶层分支节点到各个叶子节点的破裂边连接，然后再匹配除最顶层分支节点外的其它各分支节点，最后进行各叶子节点对应的从 twig 模式中根节点到自身的查询路径匹配。同时本文还重组了传统的两阶段算法，恰当地选择中间结果的归并时机，获得了较好的内存利用率。这种方法减少了数据结点比较和查询路径匹配的次数，提高了查询效率。
5. 构建了一个实验系统，并在该系统上实现了本文提出的 twig 模式查询匹配算法和文献[17]中提出的 TJFast 算法，通过多组实验对比，结果表明本文的方法在效率上有较大的提高。

1.4 论文的组织结构

全文共分为七章，的组织结构如下：

第 1 章“绪论”，课题研究背景、XML 数据查询处理技术在国内外的研究现状、目前提出的 XML 查询算法简介以及本文主要研究工作都在本章中做简要介绍。最后还介绍了本文的组织结构。

第 2 章“相关背景知识”，本章主要介绍 XML 技术及查询处理技术相关的知识，包括 XML 文档的基本构成和数据模型、XML 的模式语言、XML 的查询语言以及 twig 模式查询的概念。

第 3 章“XML 数据库技术简介”，本文在这章中首先总结了 XML 数据的编码方案，分析了各自的优缺点，并举出了编码的实例，然后重点分析了目前已经提出的几个典型 holistic twig(整体小枝)模式查询匹配算法，分析了各自的思想和执行过程，并给出了执行实例。

第 4 章“XML 文档的 UED 编码”，在本章中首先介绍了 Extended Dewey 编码的原理，然后介绍 UED 编码对 Extended Dewey 编码的改进方法，最后分析了两种编码的动态性能。

第 5 章“twig 模式查询匹配算法 TwigMatch”，本章详细介绍了本文提出的 twig 模式查询匹配算法，并对 TwigMatch 算法和 TJFast 做了详细的比较分析。

第 6 章“实验与结果分析”，本章介绍实验系统的实现及实验数据的对比分析。

第 7 章“结束语”，本章将总结全文，提出本文存在的不足和将来要做的工作。

第2章 相关背景知识

作为本文研究内容的基础知识，本章将简要介绍 XML 相关的背景知识，包括 XML 简介、XML 数据模型、XML 模式语言和查询语言等。

2.1 XML 简介

XML 是由 W3C(World Wide Web Consortium)的 XML 工作组与 1996 年起定义并维护的。该工作组对 XML 语言的描述为："XML 是 SGML(Standard Generalized Markup Language,标准通用标记语言)的子集，其目标是允许普通的 SGML 在 Web 上以目前 HTML 的方式被服务、接收和处理。XML 被设计成易于实现，且可以在 SGML 和 HTML 之间互相操作。" [1]

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bib>
  <book id="1">
    <author>Suciu</author>
    <author>Chen</author>
    <title>Computer Science</title>
    <chapter>
      <title>XML</title>
    </chapter>
  </book>
  <book id="2">
    <author>Tony</author>
    <title>Computer Software</title>
    <chapter>
      <title>Operating  System</title>
    </chapter>
  </book>
</bib>
```

(a)

```
<!ELEMENT bib(book*)>
<!ELEMENT book(author+,title,chapter*)>
<! ELEMENT author(#PCDATA)>
< !ELEMENT title(#PCDATA) >
<! ELEMENT chapter(title,section*)>
<!ELEMENT section(#PCDATA)>
```

(b)

图 2-1 XML 文档示例及对应的 DTD 文档

XML 是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以表示。它不像 HTML 语言那样，定义了一套固定的标记来表示页面元素的含义。XML 是一种元标记语言，用户可以定义自己需要的标记。这些标记必须根据某些通用的规则来创建，XML 标记描述的是文档内容的结构和含义，

而不是描述页面元素的格式化。文档本身只说明文档包括什么标记，而不是说明文档看起来是什么样的。

简单而言，一个 XML 文档主要由元素构成，元素包含属性、子元素和文本内容，子元素间可以层层嵌套，此外，一个完整的 XML 文档还可以包括 XML 声明、处理指令、注释和命名空间等部分。图 2-1(a)就是一个较完整的 XML 文档示例，它包含了一个 XML 文档基本的元素、属性和文本内容，此外还包括文档开头的声明部分。

2.2 XML 数据模型

数据模型是 XML 数据管理研究的核心问题。为了对 XML 数据进行有效的管理，就必须对 XML 数据进行建模，以正确反映 XML 数据的特性。当前的研究工作大都将 XML 数据建模为图模型[24-27]或树模型[28]。图模型将 XML 数据中的元素之间、元素和属性之间的联系抽象为有向边，把整个 XML 数据看成是一个复杂的图状结构。树模型将 XML 数据中的元素之间、元素和属性之间的联系抽象为树的有向边，把整个 XML 数据看成是一棵有向的标签树。相比于图模型，树模型得到了更广泛地使用，因此本文的研究也是基于树模型，以下对树模型做进一步介绍。

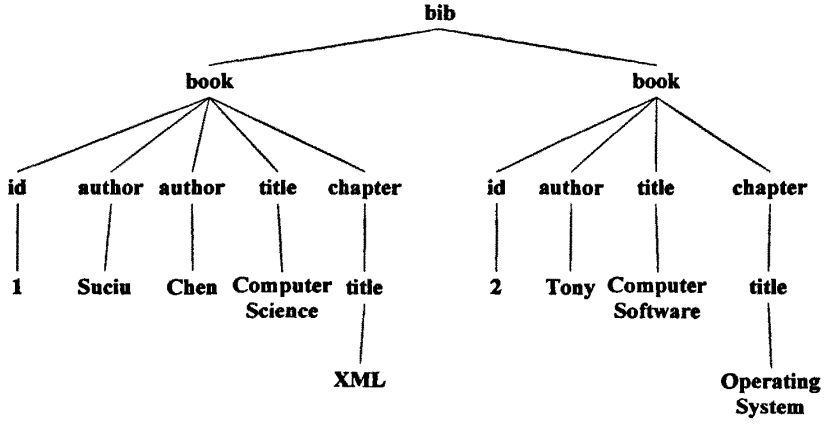


图 2-2 XML 文档的树模型表示

在树模型中，将 XML 文档中的各元素抽象为标签树中的结点，结点的类型

主要有根结点(root node)、元素结点(element node)、文本值结点(text node)和属性结点(attribute node)等。此外，该树模型是一棵有序树，标签树中的各结点之间对应顺序与每个结点在原 XML 文档中出现的顺序一致。

例 2-1 图 2-2 所示的就是图 2-1 中 XML 文档对应的树模型表示形式。在树模型表示形式中，整个 XML 文档被表示成以标签名为“bib”的结点作为根结点的标签树。原文档中的每个标签被表示为树中同名的结点，而元素之间、元素和属性之间的联系被抽象表示成模型树上的有向边。

2.3 XML 模式语言

XML 文档在本质上只是一个保存数据的结构化载体，为了得到有效的符合要求的 XML 文档，还必须明确文档中的数据必须符合的结构要求，即需要一种用来描述 XML 文档中数据结构的模型。这种数据模型不仅建立了 XML 文档中可以使用的 XML 词汇表，还定义了 XML 文档中元素的顺序和元素的嵌套规则，并建立了文档数据的数据类型。目前应用得比较广泛的是 DTD[29](Document Type Definition,文档类型定义)和 XML Schema[30](XML 模式)。

2.3.1 DTD

DTD 使用特定的语法来描述并约束 XML 文档的内容和结构。DTD 列出了可用在对应 XML 文档中的元素、属性、实体和符号表示法，以及它们之间的相互关系。DTD 通过指定文档结构的一系列规则来约束文档的结构，例如 DTD 可以规定文档的根元素必须是 bib，bib 元素只能包含一个或者多个 book 元素，book 元素可以包含一个或者多个 author 元素，title 元素紧跟在其后，之后可以是 0 个或者多个 chapter 元素，这些约束规则都是在 DTD 中定义的。图 2-1(b)就是一个 XML DTD 的例子。

一个 DTD 文档通过具体说明每一个元素和属性的名称、元素与子元素之间的嵌套关系、子元素的出现顺序和次数等来定义 XML 文档的结构模型，其通过操作符*(元素出现 0 次或者多次)、+(0 次或者一次)、?(0 次或者 1 次)、|(符号两

边的元素出现其一)来定义子元素的出现次数。在 DTD 文档中用关键字 ELEMENT 表示元素，ATTLIST 表示属性，#PCDATA 表示数据。

虽然 DTD 在表示和约束 XML 文档结构方面已经起到来很好的作用，可以在 XML 文档被程序解析时利用 DTD 来验证该 XML 文档的有效性，但仍存在一些不足，尤其在一些新兴领域应用中，体现出了以下几个局限性。

首先是 DTD 几乎不具备数据类型定义的能力，尤其是对元素的内容而言。在 DTD 中一切都被默认定义为字符串，不支持数字、复合类型等数据类型。

第二个问题是 DTD 和 XML 文档使用的是两种不同的语法。给数据处理上带来了不必要的麻烦，也使得 XML 不具备自描述的特性。

第三个问题是 DTD 的约束定义能力不足，无法对 XML 文档进行更细致的语义约束。

最后一个问题是 DTD 不够结构化，重用的代价相对较高。

鉴于 DTD 的诸多不足，已经不能满足各种应用的需要，促使人们开发出了一种更好的模式语言来替代它，这就是 XML Schema。

2.3.2 XML Schema

W3C 于 1998 年开始指定 XML Schema 的第一个版本，于 2001 年 5 月正式由官方推荐使用。

例 2-2 图 2-3 是一个完整的 XML Schema 的例子。

下面针对 DTD 的不足来介绍 XML Schema 在相应方面的改进。

在例 2-2 中共定义了 6 个元素。在 XML Schema 中用 xsd:element 元素来声明 XML 元素，用 xsd:attribute 元素来声明 XML 属性。这两个元素已经在前缀所指向的命名空间中定义好了。在每个 xsd:element 或 xsd:attribute 元素中，使用 name 和 type 属性来分别说明 XML 文档的元素或者属性的名称和数据类型，这样通过属性 type 就解决了 DTD 中只能使用字符串的单一类型。此外，还可以使用 complexType 元素来自定义复合数据类型，从而用户可以根据自己的需要自定义数据类型，更好的扩充了 XML 文档可以使用的数据类型。在该例子中，complexType 元素中包含一个 sequence 子元素，它被称为模型组。通过模型组可

以把子元素声明或引用组合起来，从而构建更加有意义的内容模型。对于 DTD 的第二个问题，XML Schema 使用标准的 XML 语法，从而 XML Schema 和 XML 之间不存在语法上的差异，XML Schema 文档可以像其它 XML 文档一样被处理和解析。对于第三个问题，XML Schema 除了使用模型组来限制子元素的出现顺序外，还为 element 元素提供 minOccurs、default、fixed 等属性来限定该元素的最少出现次数、默认值、固定取值等。对于元素值的进一步约束 XML Schema 也提供了完备的机制，如可以将一个元素的值限定在一个数值区间内、取特定的枚举值、正则表达式匹配和字符串的长度限定等。

```
<? xml version="1.0">
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema>
  <xsd:element name="bib">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="author"minOccurs="1"/>
        <xsd:element name="title"type="xsd:string"/>
        <xsd:element name="chapter"type="xsd:string"/>
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="title"type="xsd:string"/>
              <xsd:element name="section"type="xsd:string"minOccurs="0"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
<xsd:schema>
```

图 2-3 XML Schema 示例

2.4 XML 查询语言

在 XML 使用的初期，学者们就注意到了在包含丰富信息的海量 XML 数据中检索用户关心的有用信息的重要性，从而很多学者的研究重点就指向了 XML 查询语言的研究。由于 XML 文档是一种半结构化数据，无法使用关系数据库中已经比较成熟的 SQL 查询语言，而是需要一种新型的 XML 查询语言来对 XML 文档进行查询。为此，W3C 提出了多种查询语言，其中最重要的有两个：XPath[5] 和 XQuery[6]，此外还有其它组织提出的 Lorel[24]、Quilt[31]等，它们的共同特点是采用路径表达式来对 XML 文档进行定位。下面对技术上相对成熟，已经得到广

泛使用的 XPath 和 XQuery 做简要介绍。

2.4.1 XPath

在 XPath 语言中，将 XML 文档建模成具有树形结构的文档树来访问。通过 XPath 可以定位 XML 文档树中的任意结点。XPath 使用路径表达式来与 XML 文档中的对应部分进行匹配。表达式是 XPath 的主要构件，其中最重要的表达式是定位路径(location path)表达式，简称路径表达式。

定位路径表达式分为绝对和相对定位路径表达式两种。每个定位路径表达式包含一个或者多个定位步(location step)，每个定位步之间用左斜杠“/”或者双左斜杠“//”间隔。其中绝对路径从文档的根结点开始定位路径，以“/”开头来表示；相对路径则以“//”开头表示，它直接从某个定位步开始进行路径定位。

定位步是一个定位路径表达式的组成部分，而一个定位步又包含以下三个部分：

一个轴，它指定了定位步选择结点与上下文结点之间的树状关系。如 child 轴表示选择上下文结点的孩子结点，XPath 共定义了 13 个轴以满足各种操作的需求。这 13 个轴可以分为三大类：自身轴、反向轴和向前轴。其中自身轴包括 self；反向轴包括 ancestor、parent、preceding、preceding-sibling 和 ancestor-or-self；向前轴包括 descendant、child、attribute、following 和 following-sibling；此外还有 namespace 轴。

一个结点测试，它用于指定定位步中选择结点的结点类型或结点名。

0 个或多个谓词，它使用专有的谓词表达式进一步过滤定位步选择的结点集合。

以上三部分由双冒号“::”和一对中括号“[]”间隔，共同组成一个定位步，其中“::”用于分隔轴名和结点测试，“[]”用于表示谓词。例如，在定位步 child::chapter[title=“XML XPath”]中，child 是轴名，chapter 是结点测试，而[title=“XML XPath”]就是一个谓词。

对于一个定位步内部的计算先从轴和结点测试开始，产生初始结点集合后，利用其后的各个谓词对初始结点集合进行筛选，最后得到结果结点集合。

下面简要介绍在本文中将会用到的几个常用的 XPath 语法。

1. /bib/book/author

这是一个绝对单路径表达式，查询所有满足条件的 author 结点，该 author 结点的双亲结点为 book，book 结点的双亲结点为根结点 bib。

2. //chapter/title

这是一个相对单路径表达式，查询所有双亲结点为 chapter 的 title 结点。

3. //book[@id]

这是一个带有属性谓词的相对单路径表达式，查询所有具有 id 属性的 book 结点。

4. //book[/author="Tony"]/title

这是一个带有分支节点的复合路径表达式，即 twig 模式查询表达式。查询所有满足条件的 title 结点，该 title 结点的双亲结点 book 同时也是值为“Tony”的 author 结点的双亲。

2.4.2 XQuery

XQuery 是 W3C 于 2001 年首次推出的另一个被广泛使用的 XML 查询语言规范，其在推出之后又经历了几次更新，现在已经比较完善。XQuery 是定义来对 XML 数据集进行查询的，它对于 XML 数据的关系就如同我们熟知的 SQL 对于关系数据库的关系一样。XQuery 是基于 XPath 构建的，它基于 XPath 的路径表达式来实现其模式查询匹配工作，一条单独的 XPath 路径表达式本身就是一条符合 XQuery 语言语法的语句。XQuery 在具备 XPath 的所有功能的同时，还扩充了部分新的功能，例如，对范围限定的支持：/bib/book/author[1to2]，表示查询 XML 文档中出现的前两个 author 元素；使返回结果按照字母顺序排列等对结果集的转换方法。

XQuery 采用的语法相对于高级语言比较简洁，混合了 XML、注释、函数、XPath 表达式以及将它们结合到一起的专用表达式的语法，其中专用表达式包括序列表达式、算术表达式、布尔表达式、FLWOR 表达式、条件表达式和定量表达式等。XQuery 的优越性主要体现在对转换结果集的支持、提供了 FLWOR 表

达式，以支持对记录选择条件和结果转换指令的良好结合以及对函数和运算符的支持等方面。

2.4.3 XML 查询中的 twig 模式查询

本文的研究重点在于对 XML 查询中的 twig(小枝)模式查询的研究，下面就先介绍一下 XML 查询中的 twig 模式查询以及在 XPath 中是如何表示一个 twig 模式的。

文献[15]中 Bruno N 等人最先进行了 XML 查询中关于 twig 模式的相关研究，twig 是 XML 查询处理中一个重要的新兴的查询模式，其主旨就是从 XML 文档树中搜索出所有满足树状结构的查询模式的结果。文献[15]中给出了 twig 模式的如下定义：

定义 2-1 给定一个具有 n 个节点的 twig 模式 Q 和一个 XML 数据库 G，Q 在 G 中的一个匹配是指在 Q 中的节点和 G 的数据结点之间存在的映射关系满足如下条件：

1. 在该映射下，对应于查询节点的目标数据结点所含的数据必然满足查询节点上的谓词约束；
2. 目标数据结点间与查询节点组合间必须具有一致的结构关系，包括双亲/孩子关系和祖先/后代关系。

根据以上映射条件的定义，文献[15]将 twig 模式查询匹配问题定义为：

定义 2-2 给定 twig 模式 Q 和一个具有某种索引结构的 XML 数据库 G，所谓的 twig 模式查询匹配问题就是搜索 XML 数据库 G 并得到所有满足 Q 模式的 XML 数据片段，表示为 n 元组的形式。

在实际的 twig 模式查询中，通常运用得比较多的情况是一个 twig 模式中的节点在 XML 数据库中的映射数据结点是该查询需要的输出结果，其余节点以及连接它们之间的边构成了对该目标节点的条件约束。因此，结合本文描述的需要，下面先给出一些定义。

定义 2-3 目标节点

在 XML twig 模式中存在一个节点 n，它映射到 XML 数据中的数据结点就

是最终的输出结果，则该节点就称为该 twig 模式查询的目标节点。

定义 2-4 目标路径

在 XML twig 模式 Q 中，目标节点所在的从 twig 模式的根节点到叶子节点的路径称为该 twig 模式的目标路径。

定义 2-5 分支节点

在 XML twig 模式 Q 中，孩子节点数目大于等于 2 的节点称为分支节点。

定义 2-6 条件路径

在 XML twig 模式 Q 中，除了目标路径外，其它所有从根到叶节点的路径都起到了对目标节点的条件约束作用，称这些路径为条件路径。在条件路径中，往往包含一些条件谓词，用来指定元素名、限定元素取值或属性取值和限定分支节点等。

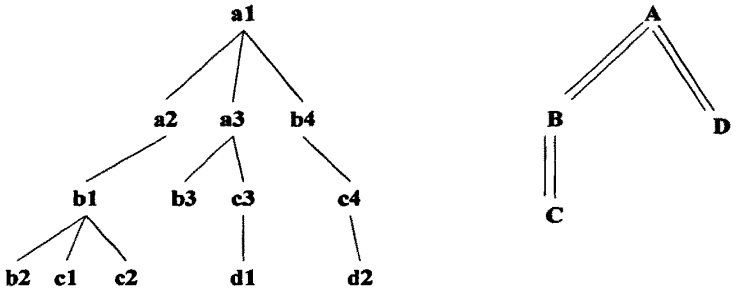


图 2-4 XML 文档树 T 和 twig 模式树 Q

例 2-3 对于图 2-4 中的 XML 文档树 T 和路径表达式 "*/A[//B//C]//D*"，对应的 twig 模式查询树为 Q。Q 所对应的查询模式就是一个 twig 模式。(*a1,b1,c1,d1*) 就是 T 中满足 Q 的一个匹配。Q 中的目标节点为 D，目标路径为 "*/A//D*"，分支节点为 A，条件路径为 "*/A//B//C*"。

为了不造成概念上的混淆，在这里做如下约定：本文将 XML 文档中的元素、属性和文本值结点统称为数据结点，在单独使用的情况下(即只描述 XML 文档)可简称结点，而一个 twig 模式中的元素、属性和文本值都统称为节点。

2.5 本章小结

XML 技术问世至今才十多年的时间，技术趋于成熟和被广泛应用也只是在

最近今年，但 XML 技术凭借其突出的优点而蓬勃发展，迅速成为 Internet 上数据表示、交换和集成的重要标准。作为本文研究内容的相关背景知识，本章简要介绍了与 XML 查询处理相关的概念和技术。

第3章 XML 数据库技术简介

目前的研究将 XML 数据库分为三大类：Native XML Database(NXDB)、XML Enabled Database(XEDB)和 Hybrid XML Database(HXDB)。本章主要总结已经提出的主流 Native XML Database 数据编码方案和 twig 模式查询匹配算法。

3.1 XML 数据的编码方案

用户对 XML 数据进行查询，通常使用 XPath 等 XML 的查询语言来表示其查询意图，然后 XML 数据库系统利用给出的查询路径表达式，在目标 XML 文档树上进行导航并返回匹配结果。在这个对目标 XML 文档树进行导航的过程中，为了求得满足条件约束的匹配结果，必然少不了对祖先/后代、双亲/孩子的包含关系和之前/之后、左兄弟/右兄弟的文档位置关系的结构判断，从而不可避免在 XML 文档上进行结构连接[12]的计算。为了有效地进行结构连接计算以有效地支持 XML 查询，研究者们进行了大量的卓有成效的工作并提出了可以显著提高结构连接效率的各种索引技术和编码方案[21]。索引技术建立了 XML 文档树的路径索引，从而可以通过路径索引来加速 XML 的结构连接计算；编码方案对 XML 文档树中的结点按照不同的思路进行编码，也就是给 XML 文档树中的每一个结点赋予一个在对应 XML 文档中唯一的编码，然后在进行结构连接计算时可以借助编码直接在常数时间内判断结点之间的结构关系，而不必对原文档进行效率低下的遍历。

目前研究文献中提出的 XML 文档树的编码方案，可以归为两大类：基于区间的编码和基于路径的编码[3]。基于区间的编码方案利用了 XML 文档有序的特点，根据某种遍历 XML 文档树的顺序赋予每一个结点一对编码，该对编码在数值上构成一个数据区间，在进行结构连接时可以根据每个结点编码区间的关系来快速判断结点间的结构关系；而基于路径的编码方案则是利用了 XML 文档结构的树状特点和文档的嵌套结构，根据 XML 文档树路径上各结点的关系，给每个

元素和属性结点赋予一个编码，该编码与其所在的从根结点到自身的路径上的所有结点相关联，从而每一个编码都包含了自己所在路径的信息。目前已经提出的比较成熟的编码方案主要有：位相量编码[32](bit-vector encoding)，区间编码[7,8,12,33](region encoding)，二叉树编码[34,35](PBiTree encoding)和前缀编码[17,23,36](prefix encoding)。

3.1.1 位相量编码

XML 文档树 T 上的每一个结点的编码由一个 n 位相量构成，n 是文档树 T 的结点总数，该 n 位相量上的第 i 位标示文档树上第 i 个结点的存在情况，“1”表示存在，“0”表示不存在。在一个自顶向下(或自底向上)的编码策略中，每一个结点的编码继承其祖先结点(或后代结点)的编码的所有位置上的“1”，从而每个结点都包含了其祖先(或后代)结点的信息。例如，图 3-1 中文档树 T 的一个结点 C 的位相量编码记为 c(C)=101000。在自顶向下(或自底向上)的编码策略中，如果文档树 T 的第 i 个结点是结点 u 本身或它的祖先(或后代) 结点，则 c(u)[i]=1，否则 c(u)[i]=0。对于自顶向下的编码策略下，可以利用二元位运算 AND(&)快速地检测一个结点 u 是否为另一个结点 v 的祖先：u 是 v 的祖先，当且仅当：

$$c(u)\&c(v)=c(u) \qquad \text{公式(3-1)}$$

例如，图 3-1 中 c(C)&c(D)=101000&101100=101000=c(C)表明结点 C 和 D 之间的祖先/后代关系。

对于自底向上的编码策略,可以利用二元位运算 OR(|)快速地检测一个结点 u 是否为另一结点 v 的后代：u 是 v 的后代，当且仅当：

$$c(u)|c(v)=c(v) \qquad \text{公式(3-2)}$$

例如，图 3-1 中 c(C)|c(E)=011100|001000=011100=c(C)表明结点 C 和 E 之间的祖先/后代关系。

3.1.2 区间编码

按照某种规则为 XML 文档树 T 中的每一个结点赋予一对编码[*start*，*end*]以构成一个区间，并且使该区间满足：一个结点的区间编码包含在它的祖先结点的

区间编码之中。也就是说，文档树 T 的结点 u 是结点 v 的祖先，当且仅当：

$$start(u)<start(v)且\ end(v)<end(u) \qquad 公式(3\text{-}3)$$

研究者们提出了一系列确定区间的方案，可以分为全局区间编码和局部区间编码[37]，由于局部区间编码的提出是为了提高更新效率，却以降低 XML 文档中结点间的关系判断性能为代价，丧失了区间编码的最关键特性，属于全局区间编码的特例，所以下面只介绍全局区间编码。

文献[33]中的 Dietz 编码根据 XML 文档树 T 的先序和后序遍历次序来分别确定 start 和 end 值。树 T 中的每个结点被赋予一个先序遍历序号和后序遍历序号组成的二元组[*pre,post*]。在树的遍历过程中，一个祖先结点 u 在先序遍历(后序遍历)中必然先于(后于)其后代结点 v 被访问。因此，结点 u 和 v 满足祖先/后代关系，当且仅当：

$$pre(u)<pre(v)\ post(v)<post(u) \qquad 公式(3\text{-}4)$$

例如，图 3-1 中后代结点 E 的区间[5,3]包含在祖先结点 A 的区间[1,12]中。通常，在 Dietz 编码的基础上，还可以对 XML 文档树 T 中的每个结点赋予一个 *parent* 值，用以表示结点的双亲结点在先序遍历下的序号，构成[*pre,post,parent*]，从而可以快速地判断结点间的双亲/孩子关系。此外还可以为 XML 文档树 T 中的每个结点赋予一个 *level* 值，构成 [*start,end,level*]，其中 *level* 表示该结点在 XML 文档树 T 中的层次，从而只要区间编码满足祖先/后代关系的同时，两元素的 *level* 值相差 1 就可以判断它们为双亲/孩子关系。

文献[8]中 Li-Moon 编码从另一个思路确定区间的起始，其编码规则为：XML 文档树 T 中的每一个结点被赋予一个二元组[*order,size*]，其中，*order* 表示结点的扩展先序遍历序号，它是在先序遍历序号的基础上，为了给新插入结点预留空间而将序号取值由连续变为非连续。所以，Li-Moon 编码在一定范围内支持 XML 文档的更新操作，但当预留空间耗尽后将无法再继续插入新结点。*size* 表示结点的后代范围。在该编码方案下，结点的编码[*order*，*size*]还必须满足如下条件：

对于文档树中的结点 c 和它的双亲结点 p，要满足：

$$order(p)<order(c)\ 且\ order(c)+size(c)<order(p)+size(p) \qquad 公式(3\text{-}5)$$

例如，图 3-1 中，祖先结点 C 和后代结点 E 满足：order(C)<order(E)且 order(E)+

size(*E*)=55+10<order(*C*)+size(*C*)=40+30。

对于文档树 *T* 中兄弟结点 *x* 和 *y*，如果在先序遍历中结点 *y* 是结点 *x* 的右兄弟，则：

$$order(x)+size(x)<order(y) \quad \text{公式(3-6)}$$

例如，图 3-1 中左兄弟结点 *C* 和右兄弟结点 *D* 满足：order(*C*)+size(*C*)=40+30<order(*D*)=80。

因此，对于文档树中的任意两个结点 *u* 和 *v* 满足祖先/后代关系，当且仅当：

$$order(u)<order(v)\text{且 }order(v)+size(v)<order(u)+size(u) \quad \text{公式(3-7)}$$

即后代结点 *v* 的编码区间[order(*v*),order(*v*)+size(*v*)]包含在祖先结点 *u* 的编码区间[order(*u*),order(*u*)+size(*u*)]内。

Li-Moon 编码与 Dietz 编码相比，其优势之处在于对文档更新的支持。

文献[7]中提出的 Zhang 编码的规则如下：对 XML 文档树中的每一个结点赋予一个二元组[*begin,end*]。对文档树进行先序遍历，在遍历过程中对每个结点分别访问两次并产生两个相关的序号作为该结点编码的 *begin* 和 *end* 的取值。其中 *begin* 的取值为遍历该结点的所有后代结点之前访问该结点时的序号，*end* 的取值为遍历完该结点的所有后代结点之后再回溯到该结点时的序号。所以，文档树 *T* 中的任意两个结点 *u* 和 *v* 满足祖先/后代关系，当且仅当：

$$begin(u)<begin(v)\ \text{且 }end(v)<end(u) \quad \text{公式(3-8)}$$

也就是祖先结点 *u* 的编码区间[begin(*u*),end(*u*)]包含后代结点 *v* 的编码区间[begin(*v*),end(*v*)]。例如，图 3-1 中的祖先结点 *C* 的编码区间[4,9]包含后代结点 *E* 的编码区间[7,8]。

文献[12]中提出的 wan 编码的规则如下：为 XML 文档树中的每一个结点赋予一个二元组<*order,maxOrder*>，其中 order 为结点的扩展先序遍历序号，定义与 Li-Moon 编码中的 *order* 相同，*maxOrder* 为结点后代中最大的扩展先序遍历序号，即以该结点为根的子树中最右下角结点的扩展先序遍历序号。XML 文档树中任意两个结点 *u* 和 *v*，*u* 是 *v* 的祖先结点，当且仅当：

$$order(u)<order(v)\text{且 }maxOrder(v)<maxOrder(u) \quad \text{公式(3-9)}$$

即结点 *u* 的编码区间[order(*u*),maxOrder(*u*)]包含结点 *v* 的编码区间[order(*v*),

maxOrder(v)]。例如，图 3-1 中的祖先结点 A 的编码区间[1,40]包含后代结点 E 的编码区间[30,30]。

3.1.3 二叉树编码

为了有效地进行 XML 文档查询中包含连接的计算，香港科技大学的 Wang W、Jiang HF 和 Lu HJ 等提出了基于完全二叉树的 PBiTree 编码。该编码方案巧妙地将 XML 文档树嵌入到一棵与之相对应的完全二叉树——PBiTree 中，在判断两个结点之间的祖先/后代、双亲/孩子关系的包含关系时只需进行整数和移位运算，相比于浮点数运算，在现代计算机体系结构中将具有更高的效率。

下面先介绍 PBiTree 的概念和性质再介绍 PBiTree 编码。

定义 3-1 PBiTree[34]

PBiTree 是一棵带标记的完全二叉树。树中的每个结点被赋予一个中序遍历序号，作为该结点的 PBiTree 编码。

性质 3-1 在 PBiTree 中，对于一个给定的结点 n_i，其处在第 h_j 层的结点 n_j 可以使用公式(3-10)直接计算出来：

$$F(n_i, h_j)=2^{h_j+1}\left\lfloor n_i / 2^{h_j+1} \right\rfloor+2^{h_j} \qquad \text{公式(3-10)}$$

性质 3-2 对于一个给定结点的编码 n，它的高度 height(n)正好是其 PBiTree 编码的二进制表示形式中最右边“1”的位置。因此，结点 n 的层次 level(n)可以由公式(3-11)计算而来，其中 H 表示整棵 PBiTree 的高度：

$$\text{level}(n)=H-\text{height}(n)-1 \qquad \text{公式(3-11)}$$

由以上性质可知，在 PBiTree 中，从任何一个结点开始都可以计算出在它之上的任何一个层次的祖先结点的 PBiTree 编码；任何一个结点的 PBiTree 编码中隐含了该结点的层次和高度信息。

树状结构的 XML 文档树，通常并不是一棵完全二叉树，为了利用完全二叉树的性质，首先需要将原始 XML 文档中的结点嵌入到一棵 PBiTree 中，该过程称为二叉化(binarization)。二叉化的过程可以使用如下的单射函数递归表示：

1. $h(u_i)=h(u_j)$当且仅当 $u_i=u_j$；

2. h(u_i)在 PBiTree 中是 h(u_j)的祖先当且仅当 u_i 在原始的 XML 文档树中是 u_j 的祖先。

其中 u 表示 XML 文档中的结点，h(u)表示结点 u 映射到 PBiTree 上的结点。

沿着 PBiTree 编码的思路，文献[35]中学者们提出了基于 EXN-Tree 的二叉树编码。EXN-Tree 的定义如下;

定义 3-2 EXN-Tree

EXN-Tree 是满足如下条件的一棵树

1. 完全二叉树;
2. 给树的每条边以及每个结点进行编码。边编码确定规则：左孩子所属边编码为 0，右孩子所属边编码为 1，到根结点的边编码为 1;
3. 从根到一个结点的路径上的边编码组成一个 2 进制数，将该二进制书转化为十进制数作为该结点的编码。每个结点由该编码唯一确定;
4. 根结点所在高度为 0，其它各结点依次从上到下递增。

从以上定义可以看出，EXN-Tree 在本质上与 PBiTree 是一样的，只是在生成完全二叉树的各结点的编码时采取了另一种思路。

3.1.4 前缀编码

前缀编码的思想最初在 Dewey 编码[36]中体现出来，所以前缀编码也可以称为 Dewey 编码。在前缀编码中直接将一个结点的双亲结点的编码作为该结点编码的前缀。所以一个结点的前缀编码通常包含两个部分：双亲结点的编码和自身的编码。在前缀编码方案下，任意两个结点间的祖先/后代关系判断就转化为了一个字符串是否为另一个字符串的前缀的判断，作为前缀者即为祖先结点。而双亲/孩子关系的判断也变得很直观，只要一个结点的编码的双亲结点编码部分等于另一结点的编码，则该结点就是孩子结点且两结点满足双亲/孩子关系。此外，前缀编码的又一个重要性质是它的字典有序性：以结点 r 为根的子树中的任一个结点 u，它的前缀编码大于(或小于)它的左兄弟子树(右兄弟子树)中所有结点的前缀编码。因此，前缀编码在有效支持祖先/后代、双亲/孩子的包含关系的计算的同时，还能够有效地支持之前/之后、左兄弟/右兄弟的文档位置关系的计算。

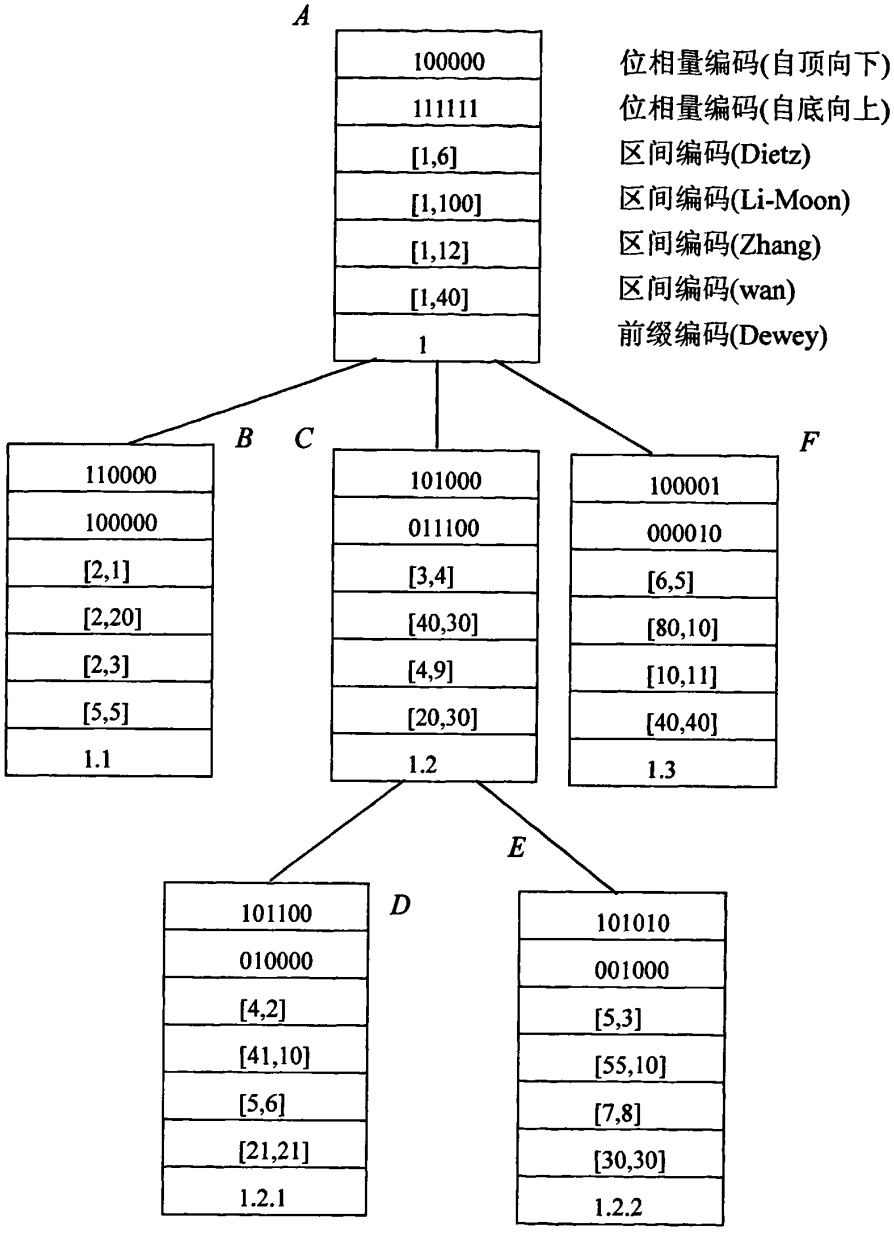


图 3-1 XML 文档编码示例

基于前缀编码的优良特性，本文选取了基于前缀编码 Dewey 编码的 Extended Dewey 编码[17]作为本文的编码方案。Extended Dewey 编码不仅具备 Dewey 的所有特性，还可以通过一个 FST 从一个结点的编码快速计算出从 XML 文档树中的根结点到自身的标签路径。从而可以利用此特性优化 twig 模式的查询，提高查询效率。

3.2 XML 数据库的索引技术

索引技术可以给 XML 文档的查询效率带来很大的提高。索引技术主要涉及两个方面的问题：一是建立索引的对象，即在什么数据或内容上建立索引；二是索引的逻辑组织结构[3]。

在如今已比较成熟的关系数据库技术中，建立索引的对象较直观和简单，即关系数据库中的元组中的某个属性。但是在 XML 数据库中，不仅有 XML 文档中关键字搜索这样的类似于关系表中的值查询，更多的是类似于 XPath 或 XQuery 中的同时对值和结构(如一个结点与另一个结点之间的关系)的查询。因此，在 XML 数据库中，既要在某些值上建立索引又要对文档结构建立索引。在 XML-enable 数据库中，通常需要建立如下几个索引[21]：值索引，建立在属性值或者元素内容上的索引；结点名索引，建立在结点标记上的索引；边或路径索引，建立在 XML 文档树的各边上的索引。

关于索引的组织结构。运用得比较广泛的主要有 B+树及其相关变体、哈希表和倒排列表。

在 XML 数据库中，最常用的索引结构还是 B+树和哈希表。此外，研究者们针对 XML 数据的某些特点，开发出了一些新的索引结构，以下做简要介绍：

Fabric 索引[38]将半结构化数据之间的关系用路径表达式来表示，再将路径表达式编码成字符串，然后以这些字符串为对象建立索引结构。该索引适合于对复杂数据字符串的快速检索。RPC 索引[39]不仅是一种索引技术同时也是一种存储技术，它是一种面向相对区间编码的索引技术。其基本思想是把一个完整的 XML 文档树划分成若干子树，使结点聚集存储在存储介质中，每个子树在存储时的大小接近于存储块的大小，然后各块链接起来形成一棵完整的树。在这样的结构上进行查询时需要进行多级存储地址的运算，查询效率受到一定的影响。XR-Tree 索引[22]由一个复合索引键和附加的刺中列表相关联，在 B+树的基础上演变而来。这种索引结构更适合结构连接的计算。F-index 索引[19]是一种新的扁平结构索引，它将 XML 文档树中不同结点的可达信息事先记录下来并进行做有效组织，以使得在过滤结点时只需要访问很少的结点就可以完成结点的筛选，进而避免了现有

方法在遇到复杂结构索引时性能下降的问题。

3.3 XML 查询处理算法

近年来，国内外学者针对 XML 查询技术进行了大量研究并提出了很多算法。对 XML 文档查询处理主要从以下三个思路展开[20]：(1)将 XML 文档的查询转化为两个结点流之间的包含关系或文档位置关系的连接操作——结构连接操作。现在所提出的结构连接算法都是基于归并的思想，即先将一个查询请求分解为多个简单路径表达式，然后归并每个简单路径表达式的计算结果来产生最终结果。在一些改进算法中，在对两个结点列表进行连接的过程中利用索引来跳过尽可能多的结点以减少扫描代价。(2)将 XML 文档树和查询表达式转换成等价的字符串，然后按照某种既定的规则对两个字符串进行匹配，最后进行验证并输出最终结果。(3)从 2002 年 Bruno N 等人提出 twig 模式[15]概念以来，研究者们在这方面已经提出了一系列有效的算法，该类算法将整个查询模式当成一个整体进行匹配操作，算法效率有较大的提升。人们将这类算法统称为 twig 模式查询匹配算法。其中具有代表性的算法有 TwigStack 算法[15]、TSGeneric+算法[16]和 TJFast 算法[17]以及新近提出的 TwigList 算法[18]。以下先重点介绍 twig 模式查询匹配算法的研究概况再分别介绍以上几个具有代表性算法的思想。

3.3.1 twig 模式查询匹配算法

在 holistic twig 概念[15]提出来前，对 twig 模式查询匹配主要按照如下步骤来设计算法[20]：

1. 将 twig 模式分解为若干双亲/孩子或祖先/子孙的二元结构关系；
2. 利用传统的结构连接算法从 XML 文档树中查询出所有满足的结点集合；
3. 合并所有中间结果得到满足全局结构关系的最后结果。

基于以上算法设计的基本思路，学者们从不同角度对 twig 模式查询匹配算法进行了改进，文献[20]中将其归纳为以下几个方面:

1. 提高结构连接性能

该思路一是设计新的高效的结构连接算法，如 MPMGJN 算法[7](multi-predicate merge join)，二是在结构连接中利用相应的索引信息，以减少参与连接的结点数量。

2. 增大分解粒度

该思路就是采取增大 twig 模式分解粒度来达到减少结构连接的数目，进而提高整体查询性能。

3. 流式处理

该思路按照 twig 模式设置相应的堆栈结构，每个 twig 模式中的节点有一个堆栈与之对应，然后在顺序扫描 XML 文档树的过程中将遇到的对应于 twig 模式标签的数据结点压入相应的堆栈，在遇到对应于 twig 模式叶子节点的数据结点时，回溯经过的相应堆栈即可得到满足要求的输出结果。

TwigStack 算法 twig 模式查询最早由 Bruno N 等人研究并提出了经典的 TwigStack 算法。该算法分为两阶段：(1)计算 twig 模式中的从根到叶子的单个查询路径的局部匹配结果；(2)归并上一步的局部匹配结果得到满足 twig 模式的最终匹配结果。TwigStack 算法用堆栈保存查询路径的匹配结果。首先采用从 twig 模式的根节点开始的自顶向下的匹配方法找到可能满足 twig 模式查询要求的数据结点并将它们保存到对应的栈中，算法确保每一个来自输入集合流 T_q 的当前元素在被压入对应栈 S_q 之前，满足如下条件：(1)对于任意 $q_i \in \text{children}(q)$，集合流 T_q 的当前元素以集合流 T_{qi} 的当前元素为后代；(2)对于每一个集合流 T_{qi} 都递归满足第(1)点。满足以上两个条件的元素称为可扩展的元素。当递归处理到 twig 模式的叶子节点时，将其对应的保存在栈中的查询路径匹配结果输出。然后归并所有查询路径结果得到满足 twig 模式查询的最终结果。

例 3-1 对于图 2-4 中所示的文档树 T 和 twig 模式查询树 Q，TwigStack 算法分别为 A、B、C 和 D 四个节点建立四个栈 S_A，S_B，S_C 和 S_D 保存满足 Q 的输入元素。在每个栈 S_u 中为其每个元素 e 设置了一个指针指向其在 Q 中的父节点对应的栈 $S_{parent(u)}$ 的栈顶来指明 e 在 T 中对应的祖先。首先 TwigStack 算法按照深度优先的顺序依次获得可扩展的输入元素 $a1$、$b1$ 和 $c1$。因为 $a1$ 与 $b1$、$b1$ 与 $c1$ 之间满足 Q 中的祖先/后代关系，所以它们都是可能满足 Q 的元素。依次将它们

压入各自对应的栈 S_A，S_B 和 S_C 中，同时 *b1* 指向 S_A 栈顶，*c1* 指向 S_B 栈顶，此时的情况如图 3-2(a)所示。

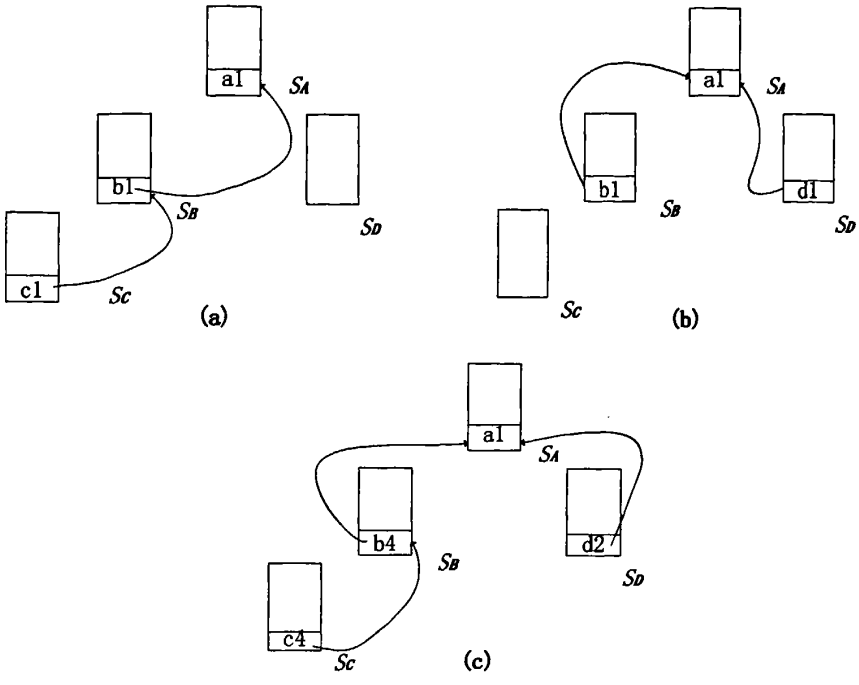


图 3-2 TwigStack 算法实例

其中由于 *b2* 没有 *C* 类型的后代元素，一定不满足 *Q* 的要求，所以算法直接跳过 *b2*，又由于 *C* 是叶子节点，所以输出对应的查询路径匹配结果(*a1*,*b1*,*c1*)并将叶子节点 *C* 的实例 *c1* 弹出栈 S_C。同理，然后可以得到 *c2* 也是满足 *Q* 的元素，将其压栈并输出(*a1*,*b1*,*c2*)，再弹栈。接着，元素 *a3*，*b3* 和 *c3* 依次被处理，都由于不满足 *Q* 的要求，被算法直接跳过。当处理到 *d1* 时，将其压入栈 S_D 中，此时的情况如图 3-2(b)所示，并因为节点 *D* 是 *Q* 中的叶子节点而输出对应的查询路径匹配结果(*a1*,*d1*)，再将 *d1* 弹出栈 S_D。最后得到 *b4*，*c4* 和 *d2* 三个满足 *Q* 的元素，依次将它们压入对应栈的同时将原先栈 S_B 中的元素 *b1* 弹出，再令 *b4* 指向 S_A 的栈顶，*c4* 指向 S_B 的栈顶，*d2* 指向 S_A 的栈顶，此时各栈的情况如图 3-2(c)所示。输出(*a1*,*b4*,*c4*)，(*a1*,*d2*)。最后归并这些查询路径的匹配结果得到最终满足 *Q* 的匹配结果。

TSGeneric+算法　香港科技大学的 Jiang HF 等人在 XR-tree 索引结构[22]的基础上对 TwigStack 算法进行了改进，提出了 TSGeneric+算法，该算法能有效地利

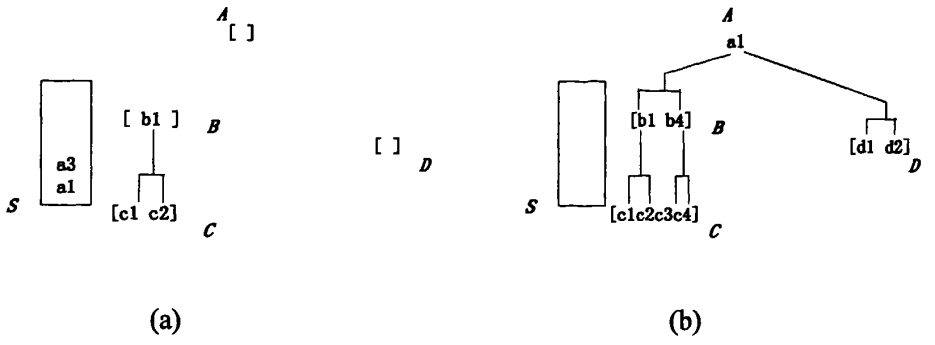
用索引跳过不参与到连接中的祖先和后代数据结点。TSGeneric+算法的思想与 TwigStack 类似，只是在寻找可扩展的元素的时候，利用了 XR-tree 索引结构来定位输入集合流的头指针，从而可以跳过一些不参与连接的元素，进一步提高了计算效率。

TJFast 算法 Lu JH 等人对前缀编码 Dewey 进行扩展，提出了支持路径计算的 Extended Dewey 编码(扩展 Dewey 编码)，并在 Extended Dewey 编码的基础上提出了一个 holistic twig 连接算法 TJFast 算法。该算法利用了 Extended Dewey 的特性，只需要访问 twig 模式的叶子节点的输入集合流。对于每一个叶子节点的输入元素，TJFast 算法首先使用一个 FST(有限状态自动机)沿着从根到该元素的路径把它们由编码转换成 XML 文档中的元素标签名称序列，再通过查询路径模式和用元素标签名称表示的路径字符串间的比较来确定输入集合流的当前元素是否满足查询路径。当 twig 模式中的每个叶子节点都找到了匹配的输入实例时，再通过为 twig 模式中分支节点分配的栈结构以及每个叶子节点对应的路径字符串间的比较来寻找叶子节点间的公共分支节点，如果该节点存在则存入对应的栈中并返回编码值最小的叶子节点。对于返回的叶子节点，如果对应的分支节点栈不为空，则说明该节点可能构成最终的匹配结果，应作为局部匹配结果输出。

例 3-2 对于图 2-4 中所示的文档树 T 和 twig 模式查询树 Q，TJFast 算法为叶子节点建立输入集合流，T_C={*c1,c2,c3,c4*}，T_D={*d1,d2*}。算法首先通过 FST 计算元素 *c1* 和 *d1* 的 Extended Dewey 编码得到各自的路径字符串*/a1/a2/b1/c1* 和*/a1/a3/c3/d1*，通过和 twig 模式中的路径模式*/A//B//C* 和*/A//D* 比较，*c1* 和 *d1* 都符合查询路径要求。然后在各自的路径字符串上寻找分支节点 *A* 的实例集合，对于 *c1* 为{*a1,a2*}，*d1* 为{*a1,a3*}，*a1* 被作为公共分支节点 *A* 的实例压入到栈 S_A 中，并返回节点 *C* 的当前实例 *c1*，输出部分匹配结果(*a1,b1,c1*)输入集合流 T_C 向后移动到 *c2*。同理，(*a1,b1,c2*)，(*a1,d1*)，(*a1,b4,c4*)和(*a1,d2*)依次被输出。

TwigList 算法 香港中文大学的 Lu 等人新近提出的 TwigList 算法采用简单的队列保存各节点的后代列表，是一种自顶向下构造后代列表，然后枚举 twig 模式查询匹配结果的算法。算法分文两个阶段：(1)构造后代列表；(2)枚举匹配结果。由于在构造的后代列表结构包含了所有的匹配信息，所以可以在第二阶段

直接枚举出最终结果，不需要进行归并，且能更好地处理包含双亲/孩子关系的查询。对于 twig 模式查询树中的每一个节点 V_i，TwigList 算法简单的维持一个队列 L_{Vi}，利用队列和队列中为每个元素分配的指针来保存完整的 twig 模式匹配结果。



(a)　　(b)

图 3-3 TwigList 算法实例

例 3-3 对于图 2-4 中所示的文档树 T 和 twig 模式查询树 Q，TwigList 算法首先建立一个用于第一阶段后代列表构造的工作栈 S，并为 twig 模式查询树 Q 中的 A、B、C 和 D 四个节点分别建立四个队列 L_A,L_B,L_C 和 L_D 来保存各节点对应的匹配元素。在 L_A 中，每个元素 a_i 维持两个指针，一个为其 B 类型的后代指明在队列 L_B 的起始范围($start_B$,end_B)，另一个为其 D 类型的后代指明在队列 L_D 中的起始范围($start_D$,end_D)。此外，每个队列 L_B 中的元素将维持一个指针，为其 C 类型的后代指明在队列 L_C 中的起始范围($start_C$,end_C)。在构造后代列表的初始阶段，将栈 S 以及 L_A,L_B,L_C 和 L_D 置为空。TwigList 算法按照深度优先的顺序访问 XML 文档树 T，即以 *a1*，*a2*，*b1*，*b2*，*c1*，*c2*，*a3*，*b3*，*c3*，*d1*，*b4*，*c4* 和 *d2* 的顺序。访问时确保栈中的元素之间的祖先/后代关系，即栈下层的元素始终是栈上层元素的祖先。在 *a1*，*a2*，*b1* 和 *b2* 依次压入栈 S 后，此时 *a1* 和 *a2* 的 $start_B$ 和 $start_D$ 指针分别指向 L_B，L_D 的队尾(即 length(L_B)+1 和 length(L_D)+1)，length(L)表示队列 L 的长度。它们的 end_B 和 end_D 此时还处于未知状态，将在元素出栈时更新。类似地，此时 *b1* 和 *b2* 的 $start_C$ 指向 L_C 的队尾 length(L_C)+1 处，其指针 end_C 也将在元素出栈时更新。当 *c1* 要压入栈 S 中时，由于当前栈 S 中栈顶的元素 *b2* 不是 *c1* 的祖先，则将 *b2* 弹出 S，又由于 *b2* 现在没有且以后也不会有 C 类型的后代元素，所以 *b2* 在弹栈后被舍弃，不能进入队列 L_B 中。同理当 *c2* 压入 S 时 *c1* 被弹出 S 并进入队列 L_C 中。此后，处理到 *a3* 时，由于目前栈 S 中的 *a2*、*b1*

和 *c2* 都不是 *a3* 的祖先，所以在依次弹出 *S* 的同时顺序进入各自队列中，其中 *a2* 由于不满足 twig 模式 *Q* 的要求被舍弃，此外还要对出栈的 *b1* 更新后代范围指针，令 $b1.end_C$=length(L_C)。此时栈 *S* 和各队列的情况如图 3-3(a)所示。此后，依次访问 *b3*、*c3*、*d1*、*b4*、*c4* 和 *d2*，*b3* 在 *c3* 压栈时弹出并舍弃，*c3* 和 *d1* 在 *b4* 压栈时被弹出并进入各自队列，接下来 *c4* 和 *d2* 分别入栈。最后栈 *S* 中的剩余元素依次出栈并在进入各自队列的同时更新各元素的后代范围指针。最后各队列的状态如图 3-3(b)所示。

3.4 本章小结

本章首先总结了 XML 数据库技术中的数据编码方案，分析了各主流方案对 XML 结点间关系判断的方法，然后分析了本文研究的又一个重点——XML 数据的 twig 模式查询匹配算法，分析了多个已经提出的 twig 模式查询匹配算法，并且都给出了执行实例分析。

第4章 XML 文档的 UED 编码

在第 1 章的绪论部分提及了改进的前缀编码——Extended Dewey 编码，该编码具有独特的特性，但不具有良好的动态性能，不能支持 XML 文档的动态更新操作。所以，在本章中，将介绍本文对 Extended Dewey 编码的改进方案，使其在保留原有特性的同时，具备良好的动态性能。

4.1 XML 文档树的 UED 编码方法

4.1.1 Extended Dewey 编码

首先，介绍 Extended Dewey 编码的编码方法。Extended Dewey 编码属于前缀编码的一种，是在 Dewey 编码的基础上提出来的，在具备 Dewey 编码的性质的同时，还可以从编码本身通过一个 FST 推导出从根结点到自身的路径上的所有结点的名称。用函数 $label_{ED}(u)$表示结点 u 的 Extended Dewey 编码。根结点的编码为 ε，其余结点的编码由两部分构成，一部分为双亲结点的编码，一部分为该结点的自身编码。设 p 是 c 的双亲结点，则：

$$label_{ED}(c) = label_{ED}(p).x \qquad \text{公式(4-1)}$$

其中 x 为结点 c 自身的编码，由此可知 x 的取值就是该 c 的编码的最后一个“.”之后的部分。x 由以下规则唯一确定：

1. 如果 c 是文本结点，则 x=-1；
2. 否则，假设数据结点 c 的名称是 $CT(t_p)(k=0,1,...n-1)$中的第 k 个标签，其中 t_s 表示结点 s 的标签名；$CT(t)$表示标签 t 的孩子名字线索(child names clue)，即在相应的 DTD 中规定的标签 t 的所有孩子的标签名。
   1) 如果 c 是 p 的第一个孩子结点，则 $x=k$；
   2) 否则，假设 c 的左兄弟的 x 值为 y，则

$$x=\begin{cases}\left\lfloor \frac{y}{n} \right\rfloor .n+k & \text{如果}(y \bmod n)<k; \\ \left\lceil \frac{y}{n} \right\rceil .n+k & \text{其它情况;}\end{cases} \qquad \text{公式(4-2)}$$

其中 n 表示 CT(t_p)的大小，即标签 t_p 的孩子的个数。

从以上 Extended Dewey 编码的确定方法上可以看出，在确定 XML 文档树中的每个结点的编码的同时，完成了一次 XML 文档数据结点到正整数的映射，该映射的依据就是结点的标签名和其在文档中的位置信息。这样在从某个结点的 Extended Dewey 编码到对应的从 XML 文档树根结点到自身的标签路径的解析时，就可以从该正整数通过一个有限状态自动机逆映射回对应的标签名。

4.1.2 FST 自动机原理

文献[17]中定义了一个有限状态自动机 FST 来将结点 u 的 Extended Dewey 编码转化为从 XML 文档的根结点到 u 的路径信息。

定义 4-1　函数 F(t,x): $\Sigma \times Z \rightarrow \Sigma$

用 Z 表示非负整数集合，Σ表示 XML 文档树 T 中的所有结点名称的集合，对于树 T 中的一个标签 t，假设 CT(t)=$\{t_0, t_1, \ldots, t_{n-1}\}$，函数 F($t$,$x$): $\Sigma \times Z \rightarrow \Sigma$可以定义为：

$$F(t,x)=t_k, \text{ 其中 } k=x \bmod n \qquad \text{公式(4-3)}$$

定义 4-2　FST(有限状态自动机)

对于给定的孩子名字线索和一个 Extended Dewey 编码，可以使用一个确定 FST 来将该给定的 Extended Dewey 编码转化为一个该编码对应的结点的从根结点到自身的路径。这个 FST 是一个 5 元组(I,S,i,δ,o)，其中

1. $I=Z \cup \{-1\}$表示 FST 的输入集合；
2. $S=\Sigma \cup \{PCDATA\}$表示 FST 状态的集合，*PCDATA* 表示一个数据结点的文本值的状态；
3. i 是 XML 文档的根结点的标签名，表示 FST 的初始状态；
4. FST 的状态转换函数定义如下：

对于所有 $t \in \Sigma$，如果 $x=-1$，$\delta(t,x)=PCDATA$，否则 $\delta(t,x)=F(t,x)$　　公式(4-4)

5. 输出值 o 为当前状态的名称。

图 4-1 所示为图 2-1(b)中描述的 DTD 对应的 FST。

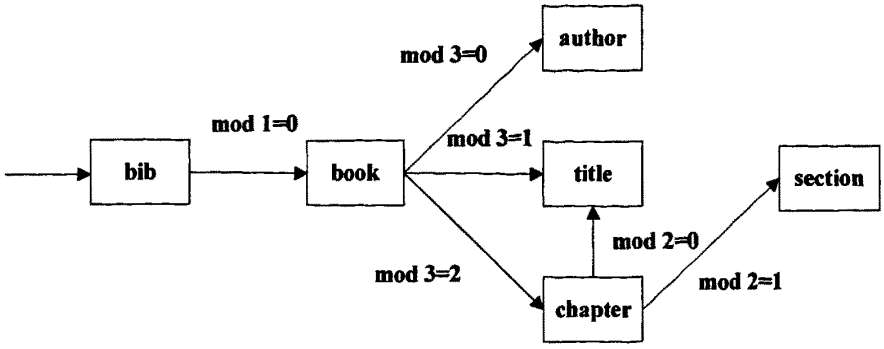


图 4-1 有限状态自动机 FST 示例

4.1.3 UED 编码

为了支持未来插入的新结点，Li-Moon 编码保留了额外的编码空间，PBiTree 和 EXN-Tree 编码保留了编码。虽然这些方法都可以很好的支持 XML 文档的插入更新操作，但当所有预留空间和编码使用完的时候，XML 文档的更新将不可避免对大量结点的重新编码。为了解决这个问题，本文借鉴 LSDX 编码[23]的思想为 XML 文档中的所有结点建立唯一的编码。

用 $\text{lable}_{UED}(u)$表示结点 u 的 UED 编码，XML 文档树中的根结点的 UED 编码为“id”，其中 id=0,1,...n，表示各个 XML 文档的唯一 ID。对于任一个非根结点 u，其编码由两部分组成：双亲结点的编码和自身编码。设 p 是 c 的父结点：

$$\text{lable}_{UED}(c)=\text{lable}_{UED}(p).x \qquad \text{公式(4-5)}$$

用 $\text{self}_{UED}(s)$表示 x 的取值。对于文本结点：

$$\text{self}_{UED}(c)=\text{“t”} \qquad \text{公式(4-6)}$$

对于非文本结点：

$$\text{self}_{UED}(c)=\text{self}_{ED}(c) \qquad \text{公式(4-7)}$$

由此可以看出，在对原始 XML 文本进行初次 UED 编码的时候，其编码原理和 Extended Dewey 编码是完全一致的，但 UED 和 Extended Dewey 的区别体现在对经过初次编码后再对原始 XML 文档进行动态更新(插入、修改和删除)操作后。Extended Dewey 编码不支持动态更新操作，而 UED 对 Extended Dewey

编码的改进之处就在于对动态更新操作的支持上。在介绍 UED 的动态更新操作前，先介绍几个必要的定义。

本文在文献[23]的启发下，对字符串的英文字母字典序定义进行相应的延伸，将字符集由有限集 26 个英文字母延伸到无限集阿拉伯数字，同时用“*”来分隔字符串中的各个阿拉伯数字。

定义 4-3 字符串在英文字母字典序(alphabetical order)意义下的大小

对于两个字符串 $S_1 = s_0 s_1 ... s_{n-1}$，$S_2 = s'_0 s'_1 ... s'_{n-1}$，其中 $s_i, s'_i \in \sum$ (i = 0,1,...,n-1)，$\sum$ 表示由 26 个英文字母组成的有限字符集合，设 j 为 S_1 和 S_2 中从左到右数起第一个不同的英文字母的位置，即 $S'_1 = s_0 ... s_{j-1}$，$S'_2 = s'_0 ... s'_{j-1}$，S'_1 和 S'_2 为相同的两个字符串。对于 S_1 和 S_2 的在英文字母字典序意义下的大小，由以下规则决定：

$$\begin{aligned} S_1 > S_2 \quad & \text{iff} \quad s_j > s'_j \\ S_1 < S_2 \quad & \text{iff} \quad s_j < s'_j \\ S_1 = S_2 \quad & \text{iff} \quad S_1 \text{和} S_2 \text{是相同的字符串} \end{aligned} \qquad \text{公式(4-8)}$$

s_j 和 s'_j 的大小由英文字母字典序大小决定。

定义 4-4 字符串在整数序意义下的大小

对于两个字符串 $S_1 = s_0 * s_1 ... * s_{n-1}$，$S_2 = s'_0 * s'_1 ... * s'_{n-1}$，其中 $s_i, s'_i \in \sum$ (i = 0,1,...,n-1)，$\sum$ 表示由任意整数组成的无限字符集合，设 j 为 S_1 和 S_2 中从左到右数起第一个不同的数字的位置，即 $S'_1 = s_0 * ... * s_{j-1}$，$S'_2 = s'_0 * ... * s'_{j-1}$，$S'_1$ 和 S'_2 为相同的两个字符串。对于 S_1 和 S_2 的在整数意义下的大小，由以下规则决定：

$$\begin{aligned} S_1 > S_2 \quad & \text{iff} \quad s_j > s'_j \\ S_1 < S_2 \quad & \text{iff} \quad s_j < s'_j \\ S_1 = S_2 \quad & \text{iff} \quad S_1 \text{和} S_2 \text{是相同的字符串} \end{aligned} \qquad \text{公式(4-9)}$$

s_j 和 s'_j 的大小由整数大小决定。

定义 4-5 整数字符串的位

对于一个整数字符串 $S = s_0 * s_1 ... * s_{n-1}$，其整数字符串的位就是由“*”分隔开的每一部分，即 s_j (j=0,1...,n-1)都是该整数字符串的一个位。其中 s_{n-1} 为该串的最末一位。

定义 4-6 整数字符串的位数

对于一个整数字符串 $S = s_0 * s_1 ... * s_{n-1}$，其整数字符串的位数(简称位数)为 n，也就是由“*”分隔的整数的个数。

定义 4-7 合法编码

对于一个字符串 $S = s_0 * s_1 ... * s_{n-1}$，当 s_{n-1} 为一个满足 Extended Dewey 编码定义的整数时，称之为一个满足 UED 要求的合法编码，简称合法编码。

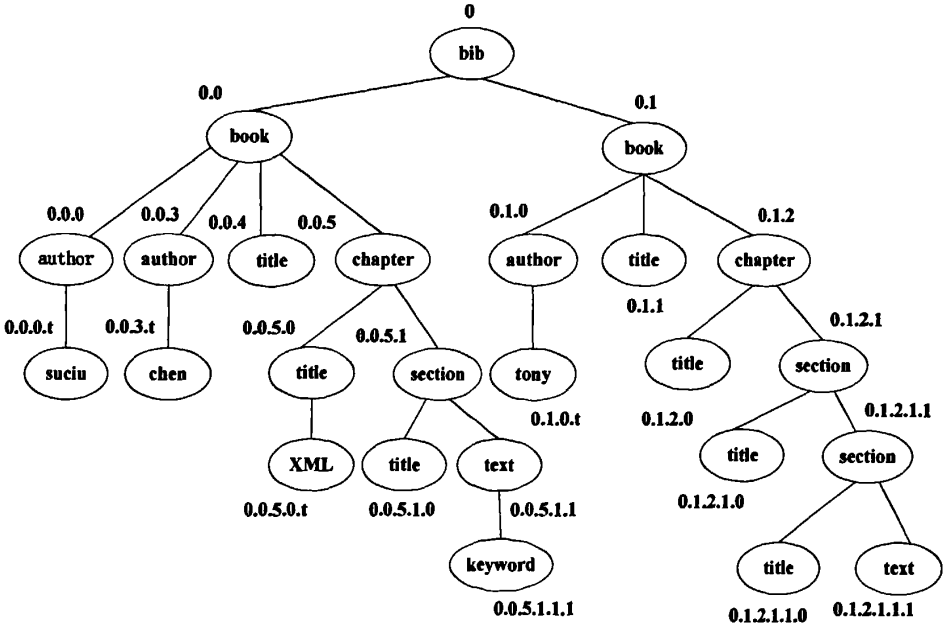


图 4-2 XML 文档树 UED 编码示例

例 4-1　图 4-2 是一个用 UED 编码对 XML 文档进行编码的例子，本文对 Extended Dewey 编码进行了一点小改动，将 XML 文档树的根结点编码为“id”，其中 id= 0，1，2，...表示文档的 ID，将文本结点编码的自身部分定义为"t"，以

区分于新插入的编码。此外，其它都与 Extended Dewey 编码一样。

在明确字符串在英文字母字典序意义下的大小、在整数序意义下的大小以及相关定义后，就可以给出新插入结点的 UED 编码的产生规则，随后介绍的新插入结点 UED 编码的计算算法要遵循该基本规则：

规则 4-1 新插入结点 UED 编码产生规则

如果在结点的插入点之前不存在别的结点，则先取得插入点之后的第一个兄弟结点的 UED 编码的自身编码部分，然后取小于该自身编码的所有位数相同的合法编码中的最大者作为新结点的 UED 编码的自身编码部分；否则，新插入结点的自身编码部分必须确保其大小处于插入点前后兄弟结点的自身编码之间，而且必须是合法编码，这里的大小指的是字符串在整数序意义下的大小。

规则 4-2 新插入结点的合法 UED 编码取值的确定规则

新插入结点的 UED 编码的取值由两部分构成，即双亲结点的 UED 编码和自身的 UED 编码，根据以上新插入结点 UED 编码产生规则，对于自身的 UED 编码的确定也可以分为两种情况，对于每种情况都要分别确定自身的 UED 位数和具体的数值。

1. 新插入结点前无其它结点

- 位数确定：其位数等于其右兄弟 u 的自身 UED 编码的位数，记结点 u 的自身 UED 的位数为 B(u)，则新插入结点 s 的位数为：B(s)=B(u)；
- 数值确定：记 u 自身部分的 UED 编码的最末一位为 last(u)，则新插入结点 s 的自身 UED 由两部分组成：前 B(s)–1 位取值与 u 的前 B(s)–1 位相同，最末一位 last(s)为比 last(u)小的合法编码取值中的最大者。例如图 4-3 中编码为 0.-1 的 book 元素和编码为 0.0.-3 的 author 元素。

2. 新插入结点位于两结点之间

设插入点 s 前后的两个结点分别为 *pre* 和 *pos*。

- 位数确定：位数的确定分为以下三种情况：
  1) B(*pre*)=B(*pos*)且 *pre* 和 *pos* 之间存在合法的数值可作为 s 的 UED 编码，则 B(s)=B(*pre*)
  2) B(*pre*)=B(*pos*)且 *pre* 和 *pos* 之间不存在合法的数值可作为 s 的 UED

编码，则 B(s)=B(pre) + 1。例如图 4-3 中编码为 0.0*0 的 book 元素、编码为 0.0.0*0 和 0.0.0*3 的 author 元素。

3) B(pre)!=B(pos)，B(s)取 B(pre)和 B(pos)中的大者。例如图 4-3 中编码为 0.0*1 的 book 元素、编码为 0.0.0*-3 和 0.0.0*3 的 author 元素。

- 数值确定：新插入结点的自身 UED 由两部分构成，分别因以上三种情况而异：

1) 前 B(s)–1 位取值与 pre 的前 B(s)–1 位相同，last(s)取合法数值中的最小者；

2) 取 $self_{UED}$(pre)作为 $self_{UED}$(s)的前 B(s)–1 位，last(s)取大于等于 0 的合法正整数中的最小者。例如，图 4-3 中编码为 0.0*0 的 book 元素和 0.0.3*0 的 author 元素。

3) 如果 B(pre)=B(pos)–1，则取 $self_{UED}$(pre)作为 $self_{UED}$(s)的前 B(s)-1 位，last(s)取比 last(pos)小的合法值中的最大者；例如，图 4-3 中编码为 0.0.0*-3 的 author 元素。如果 B(pos) = B(pre)–1，则取 $self_{UED}$(pre)的前 B(s)–1 位作为 s 的前 B(s)-1 位，last(s)取比 last(pre)大的合法值中的最小者。例如，图 4-3 中编码为 0.0*1 的 book 元素和编码为 0.0.0*3 的 author 元素。

例 4-2　图 4-3 所示为在图 4-2 的 XML 文档树中进行新结点插入操作后的编码情况。其中用虚线椭圆框表示的是新插入的结点，这些结点共分两批插入。其中，实线连接的是第一批插入的，虚线连接的是第二批插入的。各个结点的插入情况在上面已经说明。

在确定了新结点的插入更新方案后，对于已有结点的修改更新就变得相对简单了，可以通过删除已有结点再在同样的位置插入修改后新结点的方法来确定修改后结点的 UED 编码。对于被修改结点的后代结点也要将它们的编码做相应的修改，使其满足前缀编码要求。

至于结点的删除更新就更简单了，只需简单地将结点的 UED 编码删除即可，不会对其它结点的编码产生任何影响，也不需要计算新的编码。

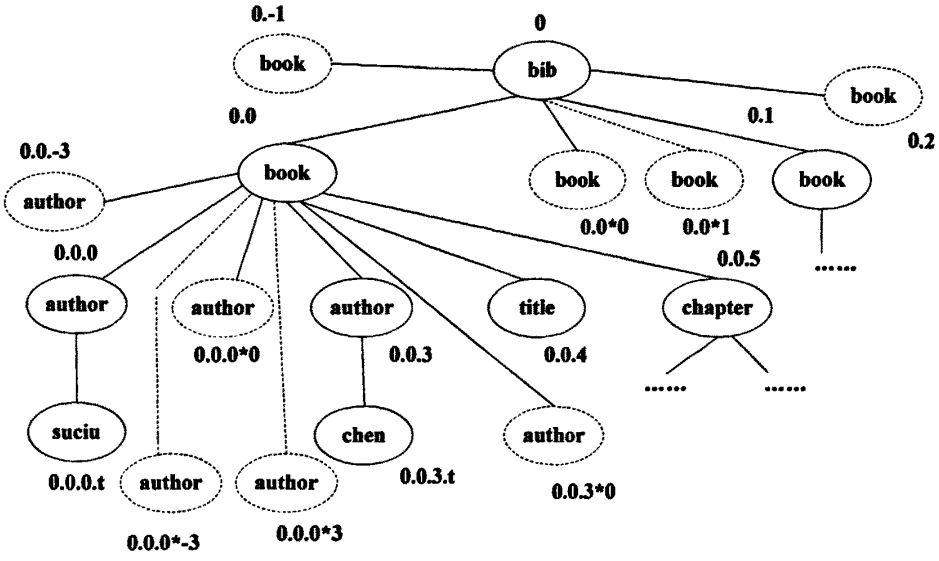


图 4-3 XML 文档树新结点插入更新的 UED 编码示例

例 4-3 图 4-4 中将 title 元素的标签名修改为 author 后，其 UED 编码的确定就是按照先删除 title 元素，再在该位置上插入 author 元素来确定的。

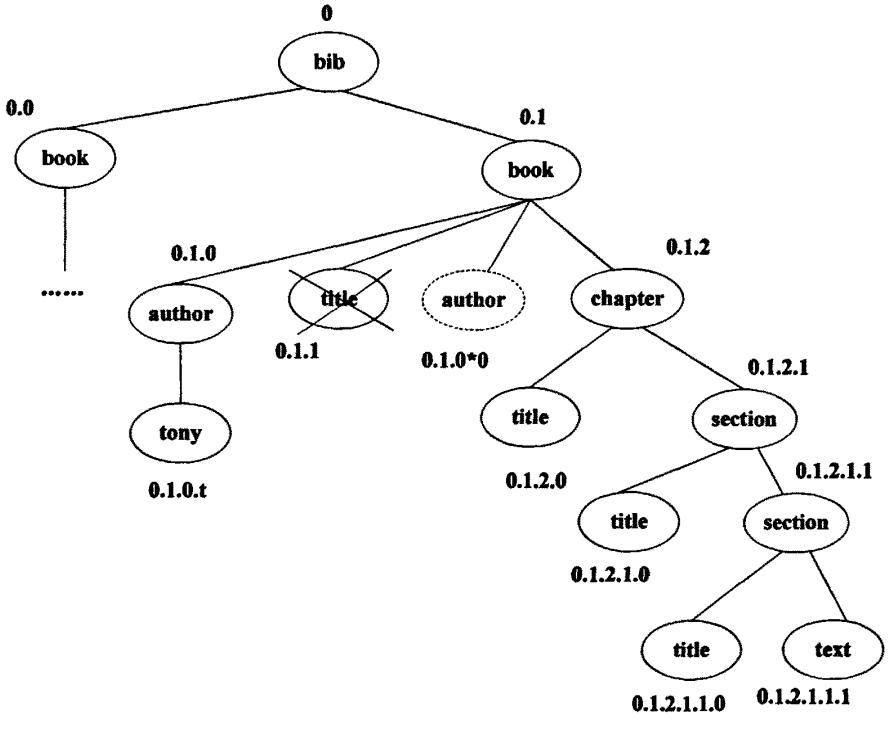


图 4-4 XML 文档树结点修改更新的 UED 编码示例

4.2 编码的动态性能分析

4.2.1 Extended Dewey 编码动态性能分析

首先分析基于 Extended Dewey 编码的 XML 文档树在动态插入新结点时，已有结点的编码受到的影响。本文仅考虑符合 DTD 要求的新结点的插入操作。

当在已经用 Extended Dewey 编码过的 XML 文档树中插入新结点，并不会改变插入点之前的各兄弟结点对应的 n、k 值以及各结点对应的前一结点的编码值，所以新插入结点对其编码没有影响，但是对处于插入点后的各兄弟结点来说，由于前一个兄弟结点的编码改变了，所以在插入点后的所有结点的编码需要相应改变，包括他们的所有子孙结点的编码也要改变。因此，在插入新结点后，处于插入点之后的所有兄弟结点及其子孙结点的 Extended Dewey 编码都必须改变。

在 XML 文档树中，假设根结点为第 0 层，往下依次为第 1 层，第 2 层等，在层数越小且越靠近第一个兄弟结点的位置插入新结点，受到影响的结点就越多。最坏的情况是在第一层的第一个兄弟结点之前插入新结点，则除了该 XML 文档的根结点外，所有的结点的编码都必须做相应的改动。由以上分析可知，基于 Extended Dewey 编码的 XML 文档树的更新效率是很低的，最坏情况下要对整个文档进行重新编码。

4.2.2 UED 编码动态性能分析

本文提出的 UED 编码和 Extended Dewey 编码原理在本质上是一致的，其实在对原始的 XML 文档进行编码时，它们是完全一样的，UED 只是借鉴文献[23]中的字符串英文字母字典序的思想，在 Extended Dewey 编码思想的基础上增加了对 XML 文档动态更新的支持，避免了对大量已有结点的重新编码，有效地降低了 XML 文档更新操作的二次编码率。UED 编码在提高了 XML 文档的动态更新的性能的同时，还很好的保留了 Extended Dewey 编码的优点，同样支持通过 FST 来计算任一结点的从根结点到自身的路径上的所有祖先结点的标签名称的特性。

4.3 编码对 XML 文档查询的支持

UED 编码以及对已编码文档进行更新后的新编码能够很好地支持高效的 XML 文档查询。对于没有进行更新操作的原始 XML 文档，其编码和 Extended Dewey 编码完全一致，文献[17]中已经对其查询性能做了详细论述。本文将对进行了更新操作的 XML 文档的查询操作进行介绍。本文将新插入和被修改后的结点的新 UED 编码记为 NUED(New UED)，该编码可以和 UED 编码一样用于判断已有编码之间、UED 和 NIUED 之间以及 NIUED 之间的祖先/后代、双亲/孩子、左兄弟/右兄弟、之前/之后关系。同时，从 NUED 编码也可以通过 FST 计算出从 XML 文档的根结点到自身的路径上的所有结点的标签名称。

定理 4-1 UED 编码支持左兄弟/右兄弟、之前/之后关系的判断。

证明：

1. 当 XML 文档初次编码后没有经过更新操作，则所有结点的编码都和 Extended Dewey 编码一样，所以支持左兄弟/右兄弟、之前/之后的关系判断。
2. 当 XML 文档在经过初次编码后进行了更新操作，插入了新结点或者修改了原有结点，则分以下两种情况讨论：
   - UED 与 UED 之间的关系判断，即原结点之间的关系判断，由 1)知支持左兄弟/右兄弟、之前/之后的关系判断；
   - UED 与 NUED 之间和 NUED 与 NUED 之间的关系判断，即原结点与新结点之间和新结点之间的关系判断，结点间的关系判断方法与 Extended Dewey 编码一样，只是在判断两编码之间的大小时利用定义 4-4 中的字符串在整数序意义下的大小来判断。

所以，UED 编码支持左兄弟/右兄弟的、之前/之后关系的判断。[证毕]

例 4-4 图 4-3 中编码为 0.0.0 和 0.0.3 的 author 元素之间的关系判断属于原结点之间的关系判断，编码为 0.0.0 和 0.0.0*-3 的 author 元素之间的关系判断属于原结点与新结点之间的关系判断，而编码为 0.0*-3 和 0.0*0 的 author 元素之间的关系判断属于两个新结点之间的关系判断。

4.4 UED 编码对 FST 自动机原理的支持

UED 编码同样支持 Extended Dewey 特有的自动机原理，只是在处理新结点 u 的 NUED 编码时取 last(u)作为 FST 的输入值，其它与文献[17]中 FST 的定义一样。

例 4-5 编码为 0.0.3*0 的元素 author 的 FST 处理转换图如图 4-5 所示。其中箭头 1 表示初始状态由根结点开始，箭头 2 由 0mod1=0 进入 book 标签状态，箭头 3 由“3*0”取 0mod3=0 进入 author 标签状态，最后得到该编码“0.0.3*0”对应的从根结点到自身的路径为/bib/book/author，在该例子中前两步与文献[17]中定义的 FST 处理一样，处理上的差异体现在第三步，在该步中只取“3*0”中的最末一位“0”作为 FST 的当前输入。

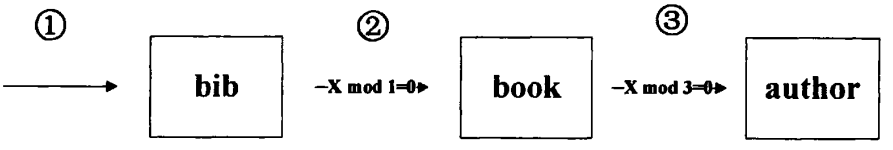


图 4-5 FST 状态转换实例

4.5 本章小结

本章针对 Extended Dewey 编码在 XML 文档更新支持上的不足，借鉴文献[23]中 LSDX 编码的思想，提出了一种 UED(Updatable Extended Dewey)编码，该编码在保留原 Extended Dewey 优点的同时，强化了对 XML 文档动态更新时，编码对更新的支持。本章首先介绍了 Extended Dewey 编码方法及其 FST 自动机原理，然后提出了本文的 UED 编码方法及对应的 FST 自动机原理，最后分析了两种编码方法的动态特性及对 XML 文档查询的支持。

第5章 twig 模式查询匹配算法

XML 数据的 twig 模式查询匹配算法的查询效率对于 XML 数据查询效率的提高具有重要意义，目前研究提出的 twig 模式查询匹配算法大多属于传统的基于归并的两阶段算法，时间和空间效率都有待提高。本章将介绍本文提出的一种改进的基于归并的 twig 模式查询匹配算法——TwigMatch 算法。

5.1 TwigMatch 算法介绍

文献[17]中的 TJFast 算法属于传统的两阶段算法，即 twig 模式匹配阶段和中间结果归并阶段[15]。其中第一阶段又分为两步进行，首先进行 twig 模式中的各叶子节点对应查询路径的单路径匹配操作，然后反复的用字符串的模式匹配来求从根数据结点到自身的路径上的分支数据结点的实例，再利用 Extended Dewey 编码的判断数据结点祖先/后代关系和双亲/孩子关系的前缀编码的特性来求叶子数据结点的公共分支数据结点，并将公共分支结点保存到对应的堆栈中，算法的效率有待进一步提高。本文提出的算法本质上也是基于 Extended Dewey 编码，只是 UED 编码对 Extended Dewey 编码而言，增加了对动态更新操作的支持。算法在大大减少查询路径匹配操作的同时对传统的两阶段算法进行阶段重组，恰当的选择中间结果的归并时机，减少了中间结果驻留内存的时间也减少了归并中间结果的时间，从而获得较好的内存利用率和查询效率。

TwigMatch 算法以 twig 模式中的各个叶子节点和最顶层分支节点的输入集合流[15]作为输入，twig 模式查询匹配阶段分三步进行，首先充分利用了 UED 编码判断文档位置关系的特点进行最顶层分支节点到各个叶子节点的破裂边连接，跳过一部分不参与连接的数据结点的同时还能将所有输入流的头指针所指的数据结点限制在与目标 twig 模式较为相近的范围内，然后再进行除最顶层分支节点外的其它分支节点的匹配，减少了分支节点的匹配操作，最后才进行 twig 模式中的各个叶子节点的对应查询路径的单路径匹配操作。

设节点 q 的输入集合流为 T_q，T_q 中保存着在 XML 文档树中对应于节点 q 的各数据结点的UED编码，流中的所有数据结点的UED编码按照编码的升序排列。例如，“1.2”位于“1.3”之前，而“1.3”位于“1.3.2”之前。

在一个输入集合流 T_q 上的操作包括 current(T_q),advance(T_q)和 eof(T_q)。current(T_q)返回流 T_q 头指针指向的数据结点的 UED 编码，advance(T_q)表示向后移动流 T_q 的指针，使其指向流 T_q 的下一个数据结点，eof(T_q)测试流 T_q 是否已经访问完毕。在不至于引起混淆的前提下，本文将以 r 为根节点的 twig 模式简称为 r。

定义 5-1 n 的解扩展[16]

XML 文档树中 twig 模式 n 的匹配结果。即对于以节点 n 为根的 twig 模式，如果存在一个完全由该查询模式的各节点对应的各输入流的头指针所指数据结点构成的解，就称节点 n 具有一个解扩展。

定义 5-2 破裂边[16]

对于 twig 模式中的一条以节点 p 和 c 为端点的边(p,c)，如果数据结点 current(T_p)和数据结点 current(T_c)不具有祖先/后代或者双亲/孩子关系，则称边(p,c)为破裂边。

例 5-1 图 2-4 中，设 T_A 和 T_C 的头指针所指数据结点分别为 *a2* 和 *c3*，由于 *a2* 和 *c3* 不具有祖先/后代关系，所以此时边(A,C)为破裂边。

TwigMatch 算法为 twig 模式中的各叶子节点和最顶层分支节点分别建立输入集合流，设 q 表示一个 twig 模式，p_n 表示一个从根节点到节点 $n \in q$ 的一个查询路径模式。在 TwigMatch 算法中，本文使用以下关于查询节点的自解释的操作：leafNode(n)返回 twig 模式中以 n 为根的叶子节点的集合；isBrokenEdge(p,q)判断 twig 模式中的边(p,q)是否为破裂边，是则返回 True，否则返回 False；compare(*code1*,*code2*)判断两 UED 编码之间的大小，如果 *code1* 大于 *code2* 则返回 True，否则返回 False；end(*root*)判断 twig 模式中以 *root* 为根的子树中的所有叶子节点所对应的输入流是否已经访问完毕，如果任何一个叶子节点对应的输入流访问完毕则返回 True，否则返回 False；isAncestorOf(a,d)判断编码分别为 a 和 d 的数据结点之间的关系，如果 a 是 d 的祖先或双亲则返回 True，否则返回 False。。此外还用到 argS_{min}(*nodes*)操作，即从 *nodes* 表示的节点集合中选择一个节点，该

节点对应的输入流的当前数据结点是 *nodes* 中的所有节点对应的输入流当前数据结点中最小的。

传统的基于归并的两阶段算法，第一阶段是算法的主体，进行 twig 模式查询匹配并存储中间结果，第二阶段对前一阶段的中间结果进行归并，两阶段间具有明确的先后关系。TwigMatch 算法对传统的两阶段算法进行了重组，两个阶段的分工仍然是 twig 模式查询匹配和中间结果归并，但是两个阶段在算法的执行过程中交叉进行，恰当地选择归并时机，将不会参与到以后中间结果归并的当前中间结果进行归并输出且释放所占用的内存空间。

5.1.1 算法第一阶段——twig 模式查询匹配阶段

算法 TwigMatch(*Q*,*T*)
输入：twig 模式 *Q* 以及 *Q* 中各叶子节点和最顶层分支节点对应的输入集合流
输出：twig 模式 *Q* 的解扩展
/*设 twig 模式中的最顶层分支节点为 *topBranchingNode*，根节点为 *root*；设 *leavesSet* 为 twig 模式中的各个叶子节点组成的集合*/
1：while(¬end(*root*)&& ¬eof($T_{topBranchingNode}$))
2： f_{act} = getNext(*leavesSet*);
3： storeMiddleSolutions(f_{act});
4： advance(T_{fact});
5： if(isBrokenEdge(*topBranchingNode*,f_{act}))
6： connectBrokenEdges();

函数 storeMiddleSolutions(f_{act})
/*为 twig 模式中的每个叶子节点 $leaf_i$ 分别建立中间结果向量 $middleResult_i$，分类存储中间结果，使用向量 $middleResult_i$ 存储叶子节点 $leaf_i$ 对应的中间结果。为 twig 模式中的每个分支节点分别建立分支节点实例向量 $branchingNodeSet_i$，分类存储匹配的分支节点实例*/
1：将 f_{act} 对应的当前路径字符串数组按照至底向上的方式与对应的路径模式 p_{fact} 进行匹配并得到中间结果 p_j 以及对应的分支节点实例 $branchingNode_j$。
2：for each p_j
3： $middleResult_i$.add(p_j);
4： $branchingNodeSet_i$.add($branchingNode_j$);

算法 getNext(*leavesSet*)
输入：twig 模式 *Q* 和 *Q* 中各叶子节点对应的输入集合流
输出：返回下一个处理的 twig 模式中的叶子节点
1：if($T_{topBranchingNode}$ 的头指针没有后移)

```
2:   leaf_min=argS_min(leavesSet);
3:   while(hasBranchingNode(leaf_min)) do
4:     if(matchSinglePath(leaf_min))
5:       return leaf_min;
6:     else
7:       advance(T_leafmin);
8:       if(isBrokenEdge(topBranchingNode,leaf_min))
9:         connectBrokenEdges();
10:        if((T_topBranchingNode 的头指针后移)
11:          mergeMiddleSolutions();
12:  end while
13:  while(True) do
14:    if(matchBranchingNode())
15:      if(∀ leaf∈leavesSet 在 XML 文档树中从根元素结点到自身的路径匹
           配 twig 模式中对应的路径模式 p_leaf)
16:        return leaf_min;
17:      else
18:        mismatchLeaf= 不满足查询路径模式 p_leaf的叶子节点;
19:        advance(T_mismatchLeaf);
20:        connectBrokenEdges();
21:        if((T_topBranchingNode 的头指针后移)
22:          mergeMiddleSolutions();
23:    else
24:      advance(T_leafmin);
25:      if(isBrokenEdge(topBranchingNode,leaf_min))
26:        connectBrokenEdges();
27:        if((T_topBranchingNode 的头指针后移)
28:          mergeMiddleSolutions();
29:  end while
30: else
31:   mergeMiddleSolutions();
```

函数 hasBranchingNode($leaf_{min}$)

```
/*令 leaf_min 对应的分支节点集合从上至下分为 branchingNodeSet_i*/
1: for each branchingNodeSet_i
2:   for each n_i∈branchingNodeSet_i
3:     if(!isAncestorOf(n_i,current(T_leafmin)))
4:       branchingNodeSet_i.remove(n_i);
5: for each branchingNodeSet_i
6:   if(isEmpty(branchingNodeSet_i))
7:     return False;
8: return True;
```

```
函数 matchSinglePath(leaf)
/*设元素 current(T_leaf)对应的从 XML 文档根结点到自身的路径字符串为 n1/n2/...
/n_k,p_leaf表示 twig 模式中的叶子节点 leaf 对应的路径模式*/
1：if(n1/n2/.../n_k与 p_leaf匹配 && n_k与 leaf 匹配)
2：    return True;
3：else
4：    return False;

函数 connectBrokenEdges()
输入：twig 模式 Q 中最顶层分支节点和各叶子节点对应的输入集合流
输出：如果最顶层分支节点与各个叶子节点之间不存在破裂边则返回 True，否则
      返回 False
/*设 leavesSet 为 twig 模式中的各个叶子节点组成的集合*/
1：leaf=leavesSet 中的第一个元素；
2：while(True)
3：    if(isBrokenEdge(topBranchingNode, leaf))
4：        join(topBranchingNode, leaf);
5：        if(leaf 是 leavesSet 中的最后一个元素 &&
             T_topBranchingNode 的头指针没有移动)
6：            getResult();
7：        else if(T_topBranchingNode 的头指针没有移动)
8：            leaf=leavesSet 中的下一个元素;
9：        else
10：           leaf=leavesSet 中的第一个元素;
11：   else
12：       if(leaf 是 leavesSet 中的最后一个元素)
13：           getResult();
14：       else
15：           leaf=leavesSet 中的下一个元素;

函数 getResult()
1：    if(¬end(root))
2：        return True;
3：    else
4：        return False;

函数 join(p,q)
1：while(¬eof(T_p) && ¬eof(T_q) && isBrokenEdge(p,q))
2：    if(compare(current(T_p),current(T_q))==False)
3：        fwdToAncestorOf(p,q);
4：    else
```

```
5:      fwdBeyond(q,p);

函数 fwdToAncestorOf(p,q)
1: while(¬eof(Tp)&&compare(current(Tp),current(Tq))==False)
2:    if( (p,q)不是破裂边)
3:       return True;
4:    advance(Tp);
5: return False;

函数 fwdBeyond (p,q)
1: while(¬eof(Tp)&&compare(current(Tp),current(Tq))==False)
2:    advance(Tp);
3: if((p,q)不是破裂边)
4:    return True;
5: else
6:    return False;

函数 matchBranchingNode()
/*设 branchingNodeSet 为 twig 模式中除最顶层分支节点 topBranchingNode 外的其
它所有分支节点的集合*/
1: for each bi in branchingNodeSet do
2:    if(leafNode(bi)不具有公共的分支节点)
3:       return False;
4: return True;
```

在主算法 TwigMatch 中，当最顶层分支节点和各叶子节点输入集合流没有被访问完毕的时候执行循环，在每次循环中，调用算法 getNext 进行 twig 模式查询匹配，在 XML 文档中计算下一个处理的数据结点，然后调用函数 storeMiddle Solutions(f_{act})保存目前找到的中间结果到对应中间结果向量中，并不需要考虑中间结果的连接，该算法并不会因为某一输入集合流的结束而结束，当分支节点集合不为空时，还要判断其它流中是否还有该分支节点集合中数据结点的后代。对于返回的节点 f_{act}，则是满足分支匹配和查询路径的节点，应当将其对应的所有满足对应路径模式的中间结果存储于对应的中间结果向量中，以待连接时处理。最后将节点 f_{act} 对应的输入集合流的头指针向后移动。

算法 getNext(*leavesSet*)是 TwigMatch 算法的核心操作，其功能为进行 twig 模式查询匹配，然后返回满足 twig 模式的节点，该节点对应的输入集合流的当前元素具有最小编码值。

在算法中分三步完成 twig 模式查询匹配。首先，调用函数 connectBroken Edges()来进行从 twig 模式最顶层分支节点到各叶子节点的破裂边连接，以缩小进入下一步叶子节点公共分支节点判断操作的数据结点的范围。然后，进行各叶子节点的公共分支节点匹配。最后，在前两项操作的基础上，进行每条从 twig 模式根节点到各叶子节点的查询路径匹配。第一步操作跳过了一些不必要参与下一步操作的数据结点，在宏观上将进入下一步操作的 XML 文档中的数据结点限制在一个相对较小的范围内，采用代价较小的破裂边连接操作来避免了不必要的从 twig 模式根节点到各叶子节点的查询路径匹配。

在进行完破裂边连接且流 $T_{topBranchingNode}$ 的头指针没有后移时分两种情况来处理：一是所有的叶子节点输入流中的当前数据结点编码最小的节点 $leaf_{min}$ 的祖先中有已经被加入到对应分支节点向量中的数据结点，则判断其是否与对应的查询路径相匹配，如果匹配就将 $leaf_{min}$ 返回。第二种情况是 $leaf_{min}$ 对应的分支节点向量中没有包含 $leaf_{min}$ 祖先，则要判断各个输入流的当前数据结点是否满足 twig 模式中的分支节点。如果满足且所有输入流当前数据结点对应的路径匹配对应查询路径，则返回具有最小编码的节点 $leaf_{min}$。而当其中任何一个节点不满足对应的查询路径时，则把其头指针向后移动并重新进行破裂边连接。

在 getNext(*leavesSet*)算法中还要适时地根据最顶层分支节点的处理情况来进行中间结果的归并操作。当完成一次破裂边连接操作后都要判断流 $T_{topBranchingNode}$ 头指针是否发生了后移，如果是则说明前一个最顶层分支节点的数据结点所包含的中间结果已经全部处理完并存储到中间结果向量中，需要归并中间结果并清空各中间结果向量，避免可归并中间结果继续驻留内存造成内存空间的浪费。

join(p,q)函数进行破裂边连接。该函数通过 compare(p,q)来判断两节点的之前/之后关系，然后通过 fwdToAncestorOf(p,q)和 fwdBeyond(p,q)来移动输入流的头指针。该函数不断移动流指针，直到 twig 模式中的节点 p 和 q 连接成功或者输入流 T_p 或 T_q 访问完毕。该函数的工作原理如下：如果边(p,q)不是破裂边，或者 T_p 或 T_q 访问完毕，该函数将返回。否则，如果数据结点 current(T_p)位于数据结点 current(T_q)之前，则调用 fwdToAncestorOf(p,q)来移动 T_p 指针使得 current(T_p)成为

current(T_q)的第一个祖先或双亲数据结点(如果该祖先或双亲数据结点不存在，则使 current(T_p)成为 current(T_q)之后的第一个数据结点)；否则，向前移动 T_q 的指针使 current(T_q)成为 current(T_p)之后的第一个数据结点。

函数 connectBrokenEdges()负责核心操作 getNext 三步操作中的第一步，在该算法中依次连接最顶层分支和各叶子节点，如果在各输入流没有访问完毕前将最顶层分支节点同各叶子节点连接成功，则返回 True，否则返回 False。

函数 matchBranchingNode()进行核心操作 getNext 三步操作中的第二步，在经过前一阶段破裂边连接操作后，最顶层分支节点已经是各个叶子节点的分支节点，所以该函数依次对除最顶层分支节点外的其它分支节点对应的叶子节点进行公共分支节点的匹配。在该函数中按照各分支节点从上层到下层的顺序依次处理，根据前缀编码的特点，可知分支节点实例的编码必须是对应各叶子实例的编码的前缀，所以先计算各叶子数据结点编码的最大公共前缀，然后再在这个公共前缀中寻找分支节点的实例，这样就可以缩小计算范围。如果对于每个分支节点都匹配成功，则返回 True。

5.1.2 算法第二阶段——中间结果归并阶段

函数 mergeMiddleSolutions()
/*将 twig 模式按照从根节点到最顶层分支节点再到各叶子节点 $leaf_1,...,leaf_n$ 划分为 n 个简单查询路径模式 $p_1,...,p_n$，其中 n 为 twig 模式中的叶子节点数目，每个叶子节点 $leaf_i$ 对应的中间结果 $middleResult_i$ 与简单查询路径模式 p_i 相匹配*/
1：while($middleResult_1$ 没有访问完毕)
2： for i=1 to n-1
3： 令简单查询路径模式 p_i 与 p_{i+1} 之间的分支节点分别为 $b_1,...,b_k$;
4： while($middleResult_{i+1}$.getCurrent()与 $middleResult_i$.getCurrent()不具有匹配的分支节点 $b_1,...,b_k$ && $middleResult_{i+1}$ 没有访问完毕)
5： $middleResult_{i+1}$.advance();
6： if($middleResult_{i+1}$ 访问完毕)
7： $middleResult_1$.advacne();
8： 将 $middleResult_2,...,middleResult_n$ 的头指针指向第一个元素;
9： continue;
10： if(i==n-1)
11： 输出中间结果向量 $middleResult_1,...,middleResult_n$ 中的当前元素;
12： $middleResult_1$.advance();
13： 将 $middleResult_2,...,middleResult_n$ 的头指针指向第一个元素;

14：清空 $middleResult_1,...,middleResult_n$ 和 $branchingNodeSet_1,...,branchingNodeSet_m$ 并释放所占用的内存空间;

在函数 mergeMiddleSolutions()中进行中间结果的归并操作，由于在第一阶段 twig 模式查询匹配并存储中间结果时无法保证各中间结果向量中各元素的有序存储，所以在第二阶段归并时，采用多次扫描中间结果向量的方式进行，也就是对于中间结果向量 $middleResult_1$ 中的每个元素，分别到 $middleResult_i(i=2,...,n)$中去重复匹配。考虑到各中间结果向量中的存储的中间结果数量较小，多次扫描并不会耗费太多的 CPU 时间。当输出所有归并后的最终 twig 模式查询匹配结果后，即可清空各中间结果向量和分支节点集合并释放内存空间，然后进入下一轮算法第一阶段 twig 模式匹配再进行第二阶段中间结果归并，如此往复交叉进行直到输出所有 twig 模式查询匹配结果。

5.2 算法分析

5.2.1 TwigMatch 算法正确性分析

以下根据算法的 twig 模式查询匹配的三个步骤分别分析每步的正确性。

1. 从 twig 模式中的最顶层分支节点 *topBranchingNode* 到各叶子节点 $leaf_i$ 的破裂边连接

设该步中各输入集合流为 $T_{topBranchingNode}$ 和 $T_{leaf1},T_{leaf2}...T_{leafn}$，分别对应于 twig 模式中最顶层分支节点和各叶子节点；各集合流中的元素为 $T_{topBranchingNode}$: $e_0,e_1...e_{n_0}$, $T_{leaf1}:e_{leaf1.0},e_{leaf1.1}...e_{leaf1.n_1}$,...,$T_{leafn}:e_{leafn.0},e_{leafn.1}...e_{leafn.n_n}$；首个解扩展在各输入集合流中出现的位置分别为：$p_b,p_1,p_2,...,p_n$。

在 TwigMatch 算法中依次对 *topBranchingNode* 和各叶子节点进行破裂边连接操作，当对边(*topBranchingNode*，$leaf_1$)连接时，根据破裂边连接算法 join 知，在流 $T_{topBranchingNode}$ 和 T_{leaf1} 的头指针分别指向位置 p_t 和 p_1 之前(即边(*topBranchingNode*，$leaf_1$)仍然是破裂边)，两个流的头指针将不断通过函数 fwdToAncestorOf(*p*,*q*) 和 fwdBeyond(*p*,*q*) 调用向 p_t 和 p_1 位置靠近，直到

($topBranchingNode,leaf_1$)不是破裂边——头指针分别指向 p_t 和 p_1。在完成边($topBranchingNode,leaf_1$)连接后，算法依次连接($topBranchingNode,leaf_2$),...($topBranchingNode,leaf_n$)，各流的头指针将不断向 $p_t,p_1,p_2,...,p_n$ 靠近，最后在完成边($topBranchingNode,leaf_n$)的连接后，各流的头指针将指向 $p_t,p_1,p_2,...,p_n$。

2. 匹配除最顶层分支节点外的各分支节点

在该步中 TwigMatch 算法利用了前缀编码的特点，依次从各叶子节点对应的当前数据结点编码的最大公共前缀中寻找分支节点实例，因为在 XML 文档树中两个数据结点的祖先分支结点只能出现在各自文档路径的公共部分上——前缀编码的最长公共前缀对应的路径。

3. twig 模式中从根节点到各叶子节点的查询路径的单路径匹配

在该步中算法利用 UED 编码和 FST 来进行 twig 模式中各叶子节点的对应的查询路径匹配，确保各叶子节点输入流中的当前数据结点对应的从根结点到自身的文档路径与 twig 模式中的对应路径模式 p_{leaf} 匹配。

5.2.2 TwigMatch 算法和 TJFast 算法比较分析

TwigMatch 算法通过增加耗时较少的操作来减少耗时较大的操作，从而在算法时间上获得了较好的性能，此外重组了传统的两阶段算法，获得了较好的内存利用率。

1. 算法的主要操作

TwigMatch 算法的 twig 模式查询匹配阶段分为三个步骤，每步完成的任务不一样，所进行的主要操作也不一样。

第一步主要进行破裂边的连接操作。对于前缀编码而言，对破裂边(p,q)的连接操作就是比较节点 p 对应的当前数据结点的编码 $label_{UED}(current(T_p))$和节点 q 对应的当前数据结点的编码 $label_{UED}(current(T_q))$，如果 $label_{UED}(current(T_p))$是 $label_{UED}((current(T_q))$的前缀或者 $label_{UED}((current(T_q))$是 $label_{UED}(current(T_p))$的前缀，则连接成功。也就是说，破裂边的连接操作可以转化为字符串的前缀判断。

第二步主要进行分支节点的匹配操作。在判断过程中也是利用了前缀编码的特性，在各分支节点对应的叶子节点的数据结点编码的最大公共前缀中寻找分支

节点实例。所以该步骤中先进行字符串的前缀判断，再扫描一遍提取分支节点的实例。

第三步主要进行 twig 模式中各叶子节点对应的查询路径的单路径匹配操作。单路径匹配可以通过 twig 模式中的叶子节点对应的从 twig 模式中根节点到自身的查询路径模式 p_{leaf} 和从该叶子节点输入流当前数据结点的 UED 编码得到的路径字符串数组的匹配来完成。所以该阶段是进行两字符串数组间的匹配，此操作是三步中最耗时的，应该尽量使操作数量减到最少。

TJFast 算法的 twig 模式查询匹配阶段分为两个步骤，第一步进行 twig 模式中各叶子节点对应的查询路径的单路径匹配操作，第二步主要进行分支节点的匹配操作。

TJFast 算法的第一步操作和 TwigMatch 算法的第三步的操作一样，其第二步主要在第一步已经匹配的路径上通过不断的字符串模式匹配来寻找各叶子节点对应的分支节点，并将各叶子节点的公共分支数据结点保存到栈中。

从以上分析可以看出，TwigMatch 算法通过前两步的字符串的前缀判断大大减少了第三步的字符串数组间的匹配，只有那些有可能构成最终结果的叶子节点对应的输入数据结点才进行查询路径的单路径匹配，避免了 TJFast 算法中每个叶子节点的输入数据结点都进行查询路径的单路径匹配。此外，TJFast 算法在其第二步的分支节点匹配时，没有利用前缀编码的特性，在分支节点实例计算中采取至上而下或者至下而上的方式遍历路径字符串，进行了一些不必要的字符串匹配。

总之，TwigMatch 算法采用不同于 TJFast 算法的设计策略在两方面上减少了算法的执行时间，第一通过时间消耗较少的字符串前缀判断来大大减少了时间消耗较大的字符串数组间的匹配；第二利用前缀编码的特性，有效地缩小了分支节点实例的扫描范围。

2. 算法的 I/O 代价和 CPU 时间复杂度

由于 TwigMatch 算法和 TJFast 算法在算法的宏观设计思想上相似，而具体实现策略不一样，即都是以通过前缀编码和 FST 来进行 twig 模式中的叶子节点的查询路径匹配以及分支节点匹配为基本操作。所以对于只包含祖先/后代关系

的查询，TwigMatch 算法和 TJFast 算法的 I/O 代价和 CPU 时间复杂度的都和输入集合的大小以及最后的匹配结果大小的总和呈线性相关。

此外，TJFast 属于传统的两阶段算法，最终结果的归并操作直到全部中间结果计算完毕以后才进行，归并之前要么存储于外存当中，要么驻留于内存，对于前者将增加额外的 I/O 代价，而后者将占用大量的内存资源。TwigMatch 算法重组了传统的两阶段算法，适时地选择中间结果的归并时机，将不可能参与到后面中间结果归并的当前中间结果适时归并且释放其占用的内存空间。既可以减少 I/O 代价，又避免了大量内存空间的占用。

3. 算法的空间复杂度

对于只包含祖先/后代关系的 twig 模式，TwigMatch 算法与 TJFast 算法在最坏情况下的空间复杂度是 $O(d^2*|b|+d*|f|)$，即与 twig 模式中的分支节点和叶子节点的数量呈正向相关，其中 f 为叶子节点的数量，b 为分支节点的数量，d 为输入集合中最长的编码长度。

5.3 实例分析

例 5-1 如图 5-1 所示的 XML 文档树和和图 5-2 所示的 twig 模式，使用 TwigMatch 算法进行 twig 模式查询匹配。算法初始时的输入集合流为：$T_{topBranchingNode}=T_A=\{a1,a2,a3\}$，$T_{leaf1}=T_B=\{b1,b2,b3,b4\}$，$T_{leaf2}=T_D=\{d1,d2\}$，$T_{leaf3}=T_E=\{e1,e2,e3\}$。

TwigMatch 算法首先进行最顶层分支节点 A 到各叶子节点 B、D 和 E 的破裂边连接，对于各输入集合流的当前元素 $a1$、$b1$、$d1$ 和 $e1$，此时 twig 模式中的边 (A,B)和(A,E)都不是破裂边，但边(A,D)是破裂边。在对破裂边(A,D)进行连接后，最顶层分支节点流的当前元素指向 $a2$，此时由于流 T_A 头指针的后移，导致原先的非破裂边(A,B)和(A,E)成为了新的破裂边，所以，破裂边操作将继续。在依次连接(A,B)、(A,D)和(A,E)后，各输入流的当前元素分别指向 $a2$、$b2$、$d1$ 和 $e2$，此时第一次破裂边连接完成。然后算法进入下一步分支节点匹配，在数据结点 $d1$ 和 $e2$ 对应路径的最长公共前缀 *"/r/a2/b3/c2"* 上找到分支节点 C 的一个实例

c2，分支匹配成功并返回此时具有最小编码值的 *B* 节点作为 getNext(*leavesSet*)的返回值。在存储了节点 *B* 对应的中间结果(*a2,b2*)和分支节点实例 *a2* 到对应的向量中后，T_B 的指针指向 *b3*，由于 *b3*、*d1* 和 *e2* 是 *a2* 的后代，所以它们对应的中间结果(*a2,b3*)、(*a2,c2,d1*)和(*a2,c2,e2*)被依次存储到对应的中间结果向量中并将对应流的头指针后移。

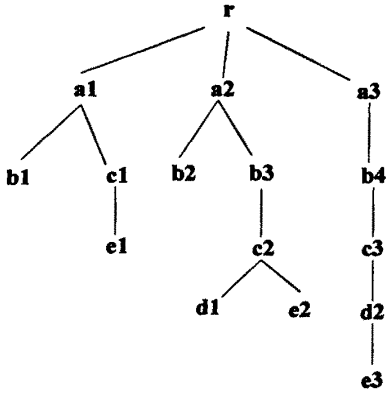


图 5-1 XML 文档树

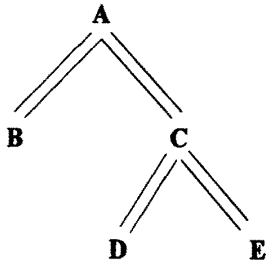


图 5-2 twig 模式

此后，各流的头指针分别指向 *a2*、*b4*、*d2* 和 *e3*，当再次进行破裂边连接后，由于最顶层分支节点 *A* 的头指针移动到了 *a3*，所以需要对 *a2* 对应的中间结果进行归并操作。第一次归并操作在当前中间结果(*a2,b2*)、(*a2,b3*) 、(*a2,c2,d1*)和(*a2,c2,e2*)之上进行，归并后得到两组 twig 模式查询匹配的解：1. (*a2,b2*)、(*a2,c2,d1*)、(*a2,c2,e2*)；2. (*a2,b3*)、(*a2,c2,d1*)、(*a2,c2,e2*)。算法在归并产生 *a2* 对应的解后就清空各中间结果向量和分支节点向量，释放占用的内存空间。最后，算法将 *a3* 对应的中间结果(*a3,b4*)、(*a3,c3,d2*)和(*a3,c3,e3*)归并产生最后一组解并释放中间结果占用的内存空间后结束。

5.4 本章小结

在本章中，针对目前效率较高的 holistic twig 模式查询算法 TJFast 的不足，基于 UED 编码提出了一种新的 holistic twig 模式查询匹配算法 TwigMatch。首先介绍 TwigMatch 算法的基本思想、各步骤的具体实现及分析了算法的正确性，最后与 TJFast 算法进行了相应的比较分析。

第6章 实验与结果分析

为了验证 twig 模式查询匹配算法的效率，本文使用由 ERCIM 组织提供的 XMark[40]标准来生成实验用的数据集 XMark 数据集。该标准被广泛用于评估 XML 数据库在现实应用中处理不同查询请求的效率，具有优良的相关性、可移植性、可扩展性和简单易用性。本文使用由 Xmark 标准提供的 XML 数据生成器生成五个大小分别为 5M、10M、15M、20M 和 25M 的 XML 文档。文献[17]中的 TJFast 算法是目前效率较高的 twig 模式查询匹配算法，在实验中，本文将 TwigMatch 算法同 TJFast 算法进行比较，TwigMatch 算法和 TJFast 算法都是用 Extended Dewey 编码作为原始 XML 文档预处理的 XML 编码，但是由于算法思想的差异，本文的算法不仅扫描较少的结点数，而且避免了大量不必要的查询路径匹配操作的同时，还适时地归并中间结果并释放占用的内存空间，从而在查询效率和空间利用率上要优于 TJFast 算法。

本文的所有实验都是在如下配置的笔记本 PC 上进行的：

CPU：Intel(R) Core(TM)2 Duo CPU 5750 @2.00GHz

内存：2G

硬盘：160G

操作系统：Windows XP Professional Service Pack 2

编程环境：Java JDK 1.5、Eclipse 3.2

6.1 评测数据集介绍

一个基于恰当评测标准的优秀的数据集的结构必须足够复杂，能够捕捉 XML 数据表示上的所有特征。现在用于 XML 评测的主流标准主要有：XMark[40]、XOO7[41]和 Xmach-1[42]等。本文采用的是根据 XMark 评测标准生成的 XMark 数据集。它包含一个模拟网上拍卖网站的应用场景。该数据集中主要表示的实体有物品条目(item)、人(people)、开放的拍卖(open_auction)、结束的拍卖

(closed_auction)和分类信息等。使用该标准提供的数据生成器可以根据需要生成大小不一的 XML 评测数据集。

为了全面的验证算法的效率，本文设计了以下查询表达式：

表 6-1 实验中使用的 twig 模式查询表达式

名字	查询表达式	说明
Q1	/site/people/person/name	简单的由 XML 文档中的根结点到某个叶子结点的简单路径查询
Q2	/site/people/person[/name][//age]//income	具有相对路径和分支节点的 twig 模式，其中分支节点为 person，目标节点为 income
Q3	//person[//watch]//interest	具有相对路径和分支节点的 twig 模式，其中分支节点为 person，目标节点为 interest
Q4	//listitem[//bold]//text[//emph]//keyword	具有相对路径和两个分支节点的 twig 模式，其中最顶层分支节点为 listitem，另外一个分支节点为 text，目标节点为 keyword

本文针对以下两方面在同 TJFast 算法比较的基础上来验证本文提出算法的性能：

查询性能：针对同一数据集上的相同的查询请求，算法返回查询结果的时间越短，则说明算法具有较高效的查询性能。

可扩展性：随着被查询的目标 XML 文档的大小的增加，XML 文档的预编码时间和执行查询的时间都会按一定的比例增长。具有较好可扩展性的查询算法可以使得该增长接近于线性，而具有较差可扩展性的算法的查询时间的增长趋向于指数比例。

6.2 XML 文档的预编码

本文使用 Extended Dewey 编码来对原始 XML 文档进行预处理，通过对 XML 文档的深度优先遍历过程依次计算每个数据结点(元素、属性以及文本值)的 Extended Dewey 编码，并使用 B+树来存储编码，作为 XML 文档预处理后的

Extended Dewey 编码的索引结构。该 B+树的逻辑组织结构如图 6-1 所示。

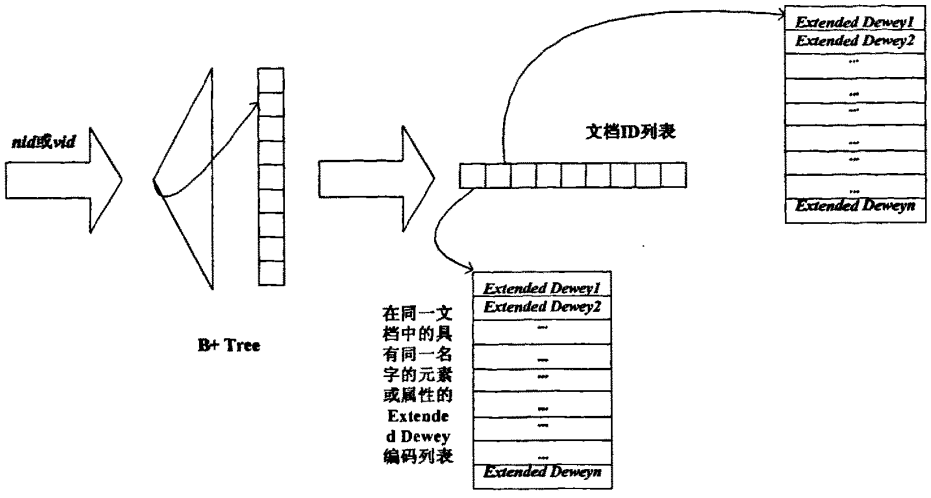


图 6-1 XML 文档 UED 编码的 B+树逻辑存储结构

其中 *nid* 表示文档中元素和属性名称的标识，*vid* 表示文档中文本值的标识。在本文中使用两个结构相同的 B+树分别存储 XML 元素(包括属性元素)和值(包括属性值和元素值)的 Extended Dewey 编码，分别称为 Element B+ Tree 和 Value B+ Tree。Element B+ Tree 以 nid 为键值，为每个文档中的每个元素分别建立存储索引，由 nid 可以关联到一个文档 ID 的列表，在列表中存储了指向每一文档的该元素的所有 Extended Dewey 编码的列表，其中每个编码按照深度优先遍历 XML 文档时的访问先后顺序存储，即按照编码大小升序排列。Value B+ Tree 与 Element B+ Tree 组织结构相同，查询键值为 vid。实验中本文使用由开源 native XML 数据库 Xindice[43]提供的 B+树 API 建立 Element B+ Tree 和 Value B+ Tree。

本文采用 Apache 开源项目 Xerces XML 解析器[44]来进行 XML 文档的解析并完成深度优先遍历以计算 Extended Dewey 编码。Xerces 包括 Java，C++和 Perl 三种语言的版本，本文使用基于 Java 的 Xerces-J。Xerces-J 支持 W3C 制定并推荐的解析 XML 文档的标准接口 SAX(Simple API for XML)接口和 DOM(Document Object Module)接口。

SAX 和 DOM 根据不同的原理来解析 XML 文档。SAX 是一种由事件驱动的轻量级的处理接口，在解析文档前不需要读入整个 XML 文档，而是顺序读入文档的过程中解析文档并根据当前处理的不同 XML 内容实时生成对应的标志文件状态的事件(如元素标签开始、元素标签结束等)，用户可以针对不同的事件定义

不同的处理函数。这样，在解析器顺序解析 XML 文档的过程中，就可以根据当前的事件和用户已定义好的对应事件的处理函数来处理该文档，直到文档结束。DOM 是一种由文档驱动的处理接口，在解析文档前需要读入整个 XML 文档，然后在内存中建立对应于该文档的 DOM 树。在 DOM 树建立完成后，对于文档的处理就转化为对 DOM 的操作，用户可以通过对 DOM 树的遍历来完成对相应文档的遍历操作。在 XML 文档较小的时候，DOM 可以将文档整体读入，并在内存中建立 DOM 树，从而具有较好的性能，但当文档较大的时候，DOM 在处理上就会耗费更多的时间和空间。

本文采用 SAX 接口来处理 XML 文档，在顺序读入 XML 文档的同时完成深度优先遍历并计算 Extended Dewey 编码，然后将编码存储于 B+树中，从而完成对原始 XML 文档的编码预处理。

6.3 twig 模式查询匹配算法的性能比较

为了评测并比较 TwigMatch 算法和 TJFast 算法在查询性能和可扩展性上的差异，针对表 6-1 中设计的查询语句，分别使用两种算法在不同大小的 XMark 数据集上进行了大量的实验。实验结果如图 6-2、6-3 所示。其中对于同一数据集使用不同算法来执行相同的查询表达式，是为了评测算法的查询性能；对于大小不同的数据集使用相同的算法来执行相同的查询表达式，是为了评测算法的可扩展性。

6.3.1 算法查询性能比较

此外，为了说明 TwigMatch 算法相比于 TJFast 算法可以有效减少参与比较数据结点的数量和避免不必要的单路径匹配操作，从而具有更好查询性能。本文还对两个算法在相同数据集上执行相同的查询表达式时所做的主要操作数量做了实验比较，结果如图 6-6、6-7 所示。从实验结果中可以看出，TwigMatch 算法通过从 twig 模式中最顶层分支节点到各叶子节点的破裂边连接操作以及其它各分支节点的匹配操作大大减少了不必要的查询路径的单路径匹配，还利用前缀编

码特性缩小了分支节点匹配时的字符串遍历范围，从而获得较好的查询性能。

图 6-2 和 6-3 所示的是算法在 10MB 和 20MB 大小的数据集上执行查询的时间(ms)。图中分别给出了两个算法的执行时间，均不包括查询 B+树索引获取输入集合流的时间，计算时间的起点都是从算法开始执行起。从图可以看出，TJFast 算法的执行时间是 TwigMatch 算法执行时间的 1.7 到 2.8 倍，尤其是在数据集较大且 twig 模式较复杂的情况下，由于路径匹配和分支节点匹配较频繁，所以 TJFast 算法在时间上需要较大的开销。

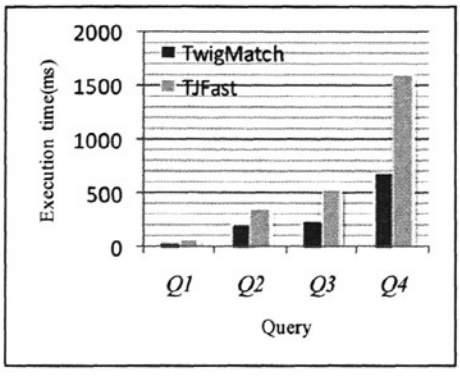


图 6-2 算法在 10MB 大小的数据集上的执行时间(ms)比较

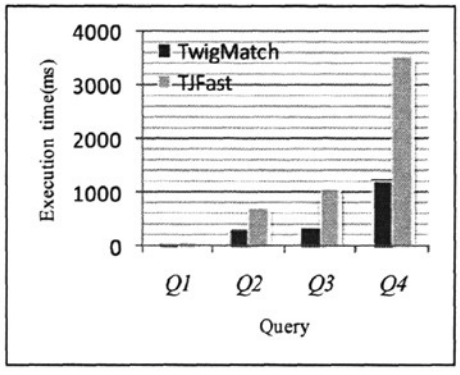


图 6-3 算法在 20MB 大小的数据集上的执行时间(ms)比较

6.3.2 算法可扩展性比较

图 6-4 和 6-5 所示是算法在不同大小的数据集上分别执行 twig 模式查询 *Q2* 和 *Q4* 的时间比较。从图中可以看出，TwigMatch 算法和 TJFast 算法都具有较好的可扩展性，算法的执行时间同数据集的大小基本上成线性相关。

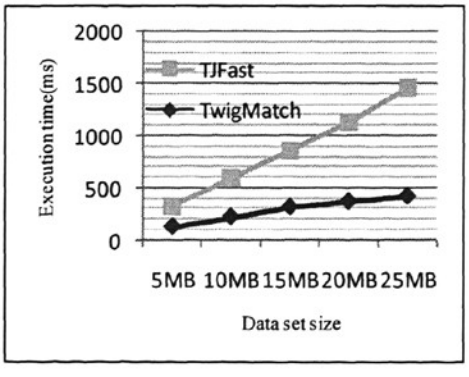


图 6-4 算法在不同大小数据集上执行 *Q2* 的时间(ms)

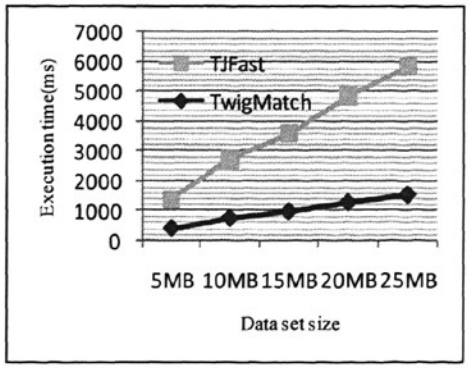


图 6-5 算法在不同大小数据集上执行 *Q4* 的时间(ms)

6.3.3 算法主要操作次数比较

表 6-2 TwigMatch 算法和 TJFast 算法在不同大小数据集上执行的主要操作数量

数据集名称	大小(MB)	结点数	查询路径的单路径匹配数								TwigMatch 算法破裂边连接次数			
			TwigMatch				TJFast							
			Q1	*Q2*	*Q3*	*Q4*	*Q1*	*Q2*	*Q3*	*Q4*	*Q1*	*Q2*	*Q3*	*Q4*
XM1	5	143676	2144	798	982	1284	2144	2711	3820	9350	0	1215	603	1774
XM2	10	288298	4292	1818	1866	2731	4292	5440	7762	18899	0	2690	1214	3713
XM3	15	431516	6442	2652	3006	3950	6442	8178	11835	27714	0	3963	1864	5475
XM4	20	573237	8539	3411	3671	5397	8539	10783	15575	38813	0	5089	2347	7346
XM5	25	714073	10662	4371	4562	6756	10662	13503	19708	47044	0	6504	2980	9052

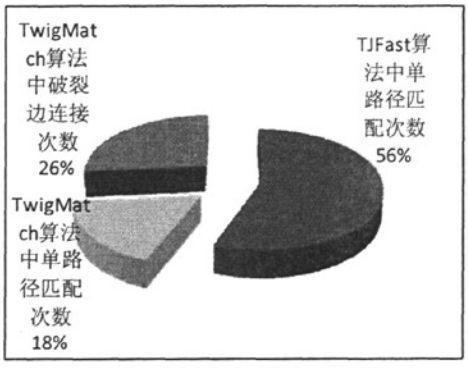


图 6-6 算法在 20MB 数据集上执行 Q2 的的主要操作次数比较

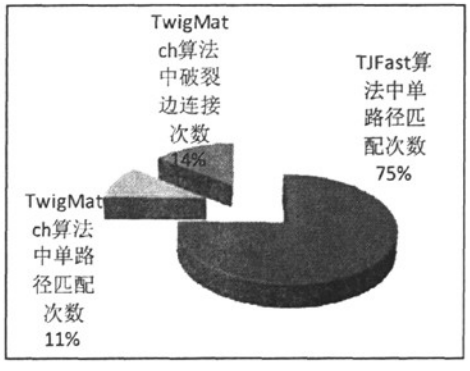


图 6-7 算法在 20MB 数据集上执行 Q4 的主要操作次数比较

表 6-2 详细记录了实验中各查询语句在各大小的数据集上的执行时，TwigMatch 算法和 TJFast 算法进行的查询路径的单路径匹配操作数量比较以及 TwigMatch 算法进行的破裂边连接次数。从表 6-2 中实验数据可以看出，TwigMatch 算法通过前两阶段的过滤后，避免了大量的查询路径匹配操作，尤其是在 twig 模式较复杂的情况下，例如 *Q4*。TwigMatch 算法减少查询路径匹配操作的代价来至于破裂边的连接以及分支节点的匹配，如表 6-2 中所示，每个查询都增加了一定数量的破裂边连接操作。但是，正如前面分析的那样，由于破裂边的连接操作实质上就是字符串间的前缀判断，相比于两字符串数组的匹配，在时间上

具有一定的优越性。

此外，从图 6-6 和 6-7 中还可以看出，TwigMatch 算法增加的破裂边连接次数与查询路径匹配次数之和也小于 TJFast 算法中的查询路径匹配次数，尤其从 *Q2* 和 *Q4* 的比较来看，twig 模式越复杂，减少的操作次数越多，从而本文算法获得了较好的查询性能。

6.4 本章小结

本章首先介绍了实验用的数据集和用于测试的查询路径表达式，然后对 XML 文档的 UED 编码逻辑存储结构进行介绍，最后通过实验数据从算法的查询性能、可扩展性以及主要操作的数量方面对 TwigMatch 算法和 TJFast 算法进行了比较分析。

第7章 结束语

XML 作为 Internet 上的一种事实上的数据表示和交换标准，已经得以广泛地应用。因此，随着 XML 以及计算机网络技术的不断向前发展，以 XML 技术作为载体的数据将越来越多，如何快速且有效地管理和查询 XML 数据是现在 XML 数据库研究的一个热点方向，已经引起了国内外学者的广泛关注。在 XML 数据查询技术的研究中，结构连接是核心操作，其中 twig 模式查询匹配算法是目前的重要研究方向。它具有较小的 I/O 代价和 CPU 时间复杂性，对于提高 XML 数据的查询效率具有重要意义。自从 Bruno N 等人于 2002 率先从事相关研究并发表相关研究成果以来，已经提出了许多 twig 模式查询匹配算法：如 TwigStack、TSGenetic+、TJFast 以及新近提出的 TwigList 算法等。其中的 TJFast 算法基于 Extended Dewey 编码，只需要访问 twig 模式中的叶子节点的输入集合流，进一步降低了 I/O 代价，具有较好的时间和空间复杂性。但是，Extended Dewey 编码不能支持 XML 文档的动态插入更新，即使是只在 XML 文档中插入一个新数据结点也不可避免要对大量数据结点重新进行编码。此外，TJFast 算法设计上存在不足，大量不必要的查询路径匹配，消耗了大量 CPU 时间。最后，本文还对现有的两阶段算法进行重组，获得了较好的内存利用率。本文在此基础上进行研究，并在总结和分析前人的研究成果的基础上提出了本文的解决方案，主要成果如下：

1. 在 Extended Dewey 编码的基础上提出了支持 XML 文档动态更新的 UED(Updatable Extended Dewey)编码。本文总结分析了一系列主流编码方案，并借鉴 LSDX 编码方案的思想，对 Extended Dewey 编码进行了相应的改进，提出的 UED 编码在保留原有特性的同时，使之支持 XML 文档的动态更新操作而不需要对整个 XML 文档重新编码，有效降低了二次编码率。
2. 针对 TJFast 算法以及传统两阶段算法的的不足，提出了 TwigMatch 算法。该方法的输入包括各叶子节点和最顶层分支节点的输入集合流，分三个

步骤来处理一个 twig 查询模式，先进行从 twig 模式中最顶层分支节点到各个叶子节点的破裂边连接，然后再匹配除最顶层分支节点外的其它各分支节点，最后进行各叶子节点对应的从 twig 模式中根节点到自身的查询路径匹配。同时本文还重组了传统的两阶段算法，恰当地选择中间结果的归并时机，及时归并中间结果并释放内存空间，获得了较好的内存利用率。这种方法减少了数据结点比较和查询路径匹配的次数，提高了查询效率。

虽然本文提出的方案在一定程度上解决了 XML 数据编码方案和 twig 模式查询中的一些问题，但是在实际应用中仍然存在一些其它亟待解决的问题，进一步的研究工作可以从以下两方面着手：

1. XML 索引技术的研究。本文没有进行 XML 索引技术的研究，也没有在算法中运用适当的索引结构来充分挖掘算法的效率，使用适当的适合 twig 模式查询匹配的索引结构将会使算法的效率得到进一步的提升。
2. XML 文档的动态编码方案研究。本文提出的 UED 编码增加了对 XML 文档动态更新的支持，但是对于 XML 文档中的某个已经存在的结点的修改更新，还是避免不了对其对应的后代结点的影响，需要对其后代重新编码，如果被修改结点的后代结点数量较大，编码的更新效率难免较低。文献[39,45]中对 XML 文档的更新问题已经做了一些有益探索，今后需要以实际需求为向导，研究新的 XML 文档的动态编码更新机制，更好地解决 XML 文档的编码动态更新问题。

参考文献

[1] World Wide Web Consortium. Extensible Markup Language (XML) 1.0(Fifth Edition).W3C Recommendation.26 November 2008.http://www.w3.org/TR/REC-xml/.

[2] World Wide Web Consortium. HTML 4.01 Specification. W3C Recommendation 24 December 1999. http://www. w3.org/TR/html4/

[3] 万常选.XML 数据库技术[M].北京:清华大学出版社,2005.1.

[4] Deutsch A,Femandez M,Florescu D,Levey A,Sciu D.A Query Language for XML[J].Computer Network ,1999.31(11-16):1155-1169

[5] World Wide Web Consortium.XML Path Language (XPath) Version 1.0.W3C Recommendation.16 November 1999. http://www.w3.org/TR/xpath.html/

[6] World Wide Web Consortium.XQuery 1.0:An XML Query Language.W3C Recommendation 23 January 2007. http://www.w3.org/TR/xquery/

[7] Zhang C,Naughton J,Dewitt D,Luo Q,Lohman G.On Supporting Containment Queries in Relational Database Management Systems [C].In:Aref WG,ed.Proc. of the 2001 ACM SIGMOD Int'l Conf. on Management of Data(SIGMOD),Santa Barbara,Califormia,USA,2001.425-436.

[8] Li QZ,Moon B.Indexing and Querying XML Data for Regular Expressions[C]In: Apers PMG,Atzeni P,Ceri S,Paraboschi S,Ramamohanarao K,Snodgrass RT,eds. Proc. of the 27th Int'l Conf. on Very Large Data Bases(VLDB),Roma,Italy,2001. 361-370.

[9] 刘云生,万常选,徐升华.基于关系数据库有效地实现 RPE 查询[J].小型微型计算机系统,2003.24(10):1764-1771.

[10] Al-khalifa S,Jagadisha HV,Koudas N,Patel JM,Srivastava D,Wu Y.Structural Joins:A Primitive for Efficient XML Query Pattern Matching[C].In:Agrawal R, Dittrich K,Ngu AHH,eds.Proc. of the 18th Int'l Conf. on Data Engineering(ICDE), IEEE Computer Society,Washington,DC,USA,2002.141- 152.

[11] Chien SY,Vagena Z,Zhang D,Tsotras VJ,Zaniolo C.Effficient Structural Joins on Indexed XML Documents [C].In:Bressan S,Chaudhri AB,Lee ML,Yu JX, Lacroix Z,eds.Proc. of the 28th Int'l Conf. on Very Large Data Bases(VLDB),HongKong,

China,2002. 263-274.

[12]万常选,刘云生,徐升华,刘喜平,林大海.基于区间编码的 XML 索引结构的有效结构连接[J].计算机学报,2005.25(l):113-127.

[13]Rao P,Moon B.PRIX:Indexing and Querying XML Using Prufer Sequences[C]. In:Titsworth F,ed.Proc. of the 20th Int'l Conf. on Database Engineering(ICDE), Boston,MA,USA,2004.288-300.

[14]Wang HX,Park S,Fan W,Yu S.VIST:A Dynamic Index Method for Querying XML Data by Tree Structures[C].In:Halevy AY,Ives ZG,Doan A,eds.Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD),San Diego, CA,2003.110-121.

[15]Bruno N,Koudas N,Srivastava D.Holistis Twig Joins:Optimal XML Pattern Matching[C].In:Franklin MJ,Moon B,Ailamaki A,eds.Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data(SIGMOD),Madision,Wisconsin, USA,2002.310-312.

[16]Jiang HF,Wang W,Lu HJ.Yu JX.Holistic Twig Joins on Indexed XML Documents[C].In:Freytag JC,Lockemann PC,Abiteboul S,Carey MJ,Selinger PG,Heuer A,eds.Proc. of the 29th Int'l Conf. on Very Large Databases(VLDB),Berlin,Germany,2003.273-284.

[17]Lu JH,Ling TW,Chan CY,Chen T.From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching[C].In:Böhm K,Jensen CS, Haas LM,Kersten ML,Larson P,Ooi BC,eds.Proc. of the 31st Int'l Conf. on Very Large Data Bases(VLDB),Trondheim,Norway,2005.193-204.

[18]Qin L,Jeffery XY,Ding B.TwigList:Make Twig Pattern Matching Fast[C].In: Proc. of the 12nd Int'l Conf. on Database System for Advanced Applications(DASFAA), Bangkok,2007.850-862.

[19]周军锋,孟小峰,蒋瑜,谢敏.F-Index:一种加速 Twig 查询处理的扁平结构索引[J]. 软件学报,2007.18(6):1429-1442.

[20]孔令波,唐世渭,杨冬青,王腾蛟,高军.XML 数据的查询技术[J].软件学报,2007. 18(6):1400-1418.

[21]孔令波,唐世渭,杨冬青,王腾蛟,高军.XML 数据索引技术[J].软件学报,2005. 16(12):2063-2079.

[22]Jiang HF,Lu HJ,Wang W,Ooi BC.XR-Tree:Indexing XML Data for Efficient Structural Joins[C].In:Dayal U,Ramamritham K,Vijayaraman TM,eds.Proc. of the

19th Int'l Conf. on Data Engineering(ICDE),Bangalore,2003.253-263.

[23]Duong M,Zhang Y.LSDX:A New Labeling Scheme for Dynamically Updating XML Data[C].In:Proc. of the 16th Australasian Database Conference,Newcaslel, Auslralia,2005.

[24]Abiteboul S,Quass D,McHudh J,Widom J,Wiener J.The Lorel Query Language for Semistructured Data[J].Int'l Journal on Digital Libraries,1997.1(1).68-88.

[25]Buneman P,Femandez M,Sueiu D.UnQL: A Query Language and Algebra for Semistructed data based on recursion[J].The VLDB Journal,2000.9(1):76-110.

[26]Melnik S,Gareia-Molina H,Rahm E.Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching[C].In:Proc. of the 18th Int'l Conf. on Data Engineering(ICDE),San Jose CA,2002.117-128.

[27]Deutsch A,Fernandez M,Suciu D.Storing Semistructured Data with STORED[C]. In:Haas LM,Tiwary A,eds.Proc. of the ACM SIGMOD Int'l Conf. on Management of Data(SIGMOD),Philadelphia,1999.431-442.

[28]Deutsch A,Fernandez M,Floreseu D,Levy A,Sueiu D.A Query Language for XML[J].Computer Networks(Amsterdam.Netherlands:1999),1999.31(16):1155-1169.

[29]Bosak J,Bray T,Connolly D,et al.W3C XML Specification("XMLspec") DTD Version2.1 15 February 2000.http://www.w3c.org/XML/1998/06/xmlspec-report-v21.htm

[30]World Wide Web Consortium.XML Schema Part 0(Part 1、Part 2):Primer.W3C Recommendation,28 October 2004.http://www.w3.org/TR/xmlschema-0/

[31]Chamberlin d,Robie J,Florescu D.Quilt:An XML Query Language for Heterogeneous Data Source[C].In:Suciu D,Vossen G,eds.Proc. of the Int'l Workshop on the Web and Databases(WebDB2000),Dalas,2000.53-62.

[32]Wirth N.Type Extensions[J].ACM Transactions on Programming Languages and Systems,1988.10(2):204-214.

[33]Dietz PF.Maintaining Order in A Linked List[C].In:Proc. of the 14th Annual ACM Symposium on Theory of Computing,SanFranciseo,Californla,l982.122-127

[34]Wang W,Jiang HF,Lu HJ,Jeffrey XY.PBiTree Coding and Efficient Processing of Containment Joins[C].In:Proc. of the 19th International Conference on Data Engineering(ICDE),Los Alamitos,2003.391-402.

[35]李英俊,宗金良,孙志胜.基于 EXN-Tree 编码的 XML 结构连接算法研究[J].计

算机应用,2006.26(10):2405-2407.

[36]Tatarinov I,Viglas SD.Storing and Querying Ordered XML Using a Relational Database System[C].In:Franklin MJ,Moon B,Ailamaki A,eds.Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD),Madison,2002. 204–215.

[37]S.Flesca,F.Furfaro,E.Masciari.On the Minimization of XPath Queries[C].In: Proc. of the 29th Int'l Conf. on Very Large Data Bases(VLDB),Berlin,Germany,2003. 241-250.

[38]Cooper BF,Sample N,Franklin MJ,Hjaltason GR,Shadmon M.A Fast Index for Semistructured Data.In:Apers PMG,Atzeni P,Ceri S,Paraboschi S,Ramamohanarao K,Snodgrass RT,eds.Proc. of the 27th Int'l Conf. on Very Large Data Bases (VLDB),Roma, 2001.341-350.

[39]O'Neil P,O'Neil E,Pal S,Cseri I,Schaller G.ORDPATHs:Insert-friendly XML Node Labels[C].In:Weikum G,König AC,Deßloch S,eds.Proc. of the ACM SIGMOD Int'l Conf. on Management of Data(SIGMOD),Paries,2004. 903-908.

[40]Busse R,Carey M,Florescu D,Kersten M,Manolescu I,Schmidt A,Waas F.XMark an XML Benchmark Project. http://monetdb.cwi.nl/xml/index.html

[41]Bressan S,Lee M,Li YG,Lacroix Z,Nambiar U.The XOO7 XML Management System Benchmark.[J]Technical Report TR21/00,National University of Singapore,CS Department,2001.

[42]Bohme T,Rahm E.XMach-1:A Benchmark for XML Data Management[C].In: Proc. of German Database Conference(BTW),2001.264-273.

[43]Apache Xindice.http://xml.apache.org/xindice/.

[44]The Apache Xerces Project.http://xerces.apache.org/.

[45]Tatarinov I,Zachary G.Ives.Updating XML[C].In:Aref WG,ed.Proc. of 2001 ACM SIGMOD Int'l Conf. on Management of Data(SIGMOD),Santa Barbara,2001. 413-424.

[46]陶世群,富丽贞.一种高效非归并的 XML 小枝模式匹配算法[J].软件学报,2009. 20(4):795-803.

致谢

值此论文完成之际，我首先要衷心地感谢我的导师高集荣副教授。两年来，在他的悉心教导下，我的知识水平和科研能力得到了很大的提升，是他严谨的治学态度、渊博的学识和勤勉的工作作风，一直鞭策着我在探索求知的道路上不断前行。论文的如期完成离不开高老师的谆谆教导，从论文的选题到理论的逐步成型和完善再到最后的终稿，每一阶段都包含了老师的心血和汗水。高老师为人处事的作风以及治学的态度深深地影响了我，将是我终生学习的榜样。

中山大学信息科学与技术学院给我们创造了良好的科研和学习环境，在此，谨向各位老师和工作人员致以衷心的感谢和由衷的敬意。

感谢计算机软件与理论班的同班同学，感谢同门曾楚伟、黄启辉同学，与你们一同学习和交流，使我获益良多。感谢所有帮助和支持我的同学老师和朋友们。

我论文中的研究工作并不是孤立存在的，是在大量前人的研究之上完成的，论文想法的由来、理论的形成和完善都离不开国内外的技术资料，在此对参考文献的作者、译者以及出版单位表示感谢！

我要特别感谢我的家人，多年来是他们始终支持我完成学业，给予我无私的关爱和帮助。

最后，感谢所有的论文审稿老师和答辩评委对我的建议和指正。