东北大学

硕士学位论文

XML复杂路径表达式查询处理技术研究

姓名：周博

申请学位级别：硕士

专业：计算机软件与理论

指导教师：于戈

20031201

XML 复杂路径表达式查询处理技术研究

摘 要

XML 是可扩展标记语言（Extensible Markup Language）的简称，它为 Web 上半结构化文档和数据提供了通用格式。XML 是一种元语言，通过对一组标签设定特定的意义，可以创建出其它的标记语言。随着 Internet 的发展尤其是 Web 技术的广泛应用，越来越多的应用采用了 XML 技术作为信息表示和数据交换的标准，这使得通过数据库技术对 XML 数据进行存储、查询等操作变得越来越重要。

在众多 XML 查询语言中，路径查询是最重要、使用最频繁的组成部分。目前提出的大多数查询方法都是在实例空间中进行的，也就是说，使用这些方法查询时，直接面对的操作对象是 XML 文档。这类方法中比较有代表性的是 XML 文档树遍历的方法和包含连接的方法。根据自动机技术，我们提出了一种通过用查询自动机匹配 XML 模式树来计算查询路径表达式的方法，称为自动机匹配算法(Automata Match,AM)，来解决 XML 正则路径和复杂路径查询表达式的策略。自动机匹配算法可以在模式空间内高效地计算路径表达式，因此这种方法可以适应在海量数据上执行复杂查询的需要。

本文提出了如何将具有各种运算符的正则表达式转化为查询自动机的方法。针对 XPath 规范中规定的“//”操作符，即祖先一后代关系操作符，我们提出了一个称为模式自动机(Schema Automata)的数据结构，模式自动机可以接收所有可能出现在 XML 文档中的片断，也就是说，它可以匹配任何可能出现在 XML 文档中的路径模式；而传统的自动机要想支持包含连接这一类非正则运算符是非常困难的。为了进一步提高模式自动机的性能，本文还提出了两种优化方法 PSA 和 RWS。前者将模式自动机作为索引的一部分存储在磁盘上，避免了每次计算都要生成模式自动机的开销，后者则是通过 following 集合和 preceding 集合来过滤掉模式自动机中多余的状态和转换函数来达到提高查询效率的目的。为了支持自动机匹配算法，本文还提出了高效地支持自动机匹配算法的数据结构：路径模式树和路径实例树。通过与结构连接算法进行性能测试对比，我们发现自动机匹配算法的效率远远高于结构连接算法，PSA 和 RWS 对自动机匹配算法的优化也很显著。

与传统的关系数据库中的查询不同，针对半结构化数据的查询更多的是要找到满足某些特定模式的节点。近来，在简单路径查询的问题得到较好解决的基础上，人们将注意力转移到 Twig 查询中来。本文提出了如何利用索引技术来更好地解决 Twig 查询的问题。根据路径模式树索引，我们给出了利用自动机匹配路径模式树索引解决这一问题的方法，围绕这一方法，本文对 Twig 查询自动机的构建，

Twig 查询自动机与路径模式树的匹配等算法进行了讨论，并与用传统的结构连接方法解决 Twig 查询进行了实验对比，结果证明，基于自动机的方法在性能上具有较大优势。

关键词：可扩展标记语言，自动机，模式自动机，路径表达式查询，Twig 查询

Study on XML Query Processing Techniques of Complex Path Expressions

Abstract

XML (Extensible Markup Language) provides general format for semistructured documents and data on the Web. It is a simple and flexible format and was originally designed for electronic publishing. Nowadays, XML plays more and more important role in Web based applications. In XML, tags are used to describe the structure informations of documents. XML is widely used as a standard language for representing and exchanging data in Web site construction, distributed application platforms and other systems. Therefore, it becomes more and more important to manage XML data through databases. Recently a lot of research work has been done for managing XML data.

Path query is one of the most frequently used components by the various XML query languages. Most of the proposed methods compute path queries in instance spaces, i.e. the XML data, such as XML tree traversal and containment join ways. As a query method based on automata technique, *Automata Matching (AM)* can evaluate path expression queries in schema space so that it allows efficient computation of complex queries on massive XML data.

This thesis first introduces how to construct query automata in order to evaluate various regular expression queries including those with wildcards. Furthermore, a data structure named *schema automata* is proposed to evaluate containment queries that are very difficult to handle from the conventional automata point of view. To improve the efficiency of schema automata, we proposed two optimization strategies, PSA and RWA. For PSA, we store the schema automata on the disk as a part of the index. The schema automata need not to be constructed dynamically every time, which thereby reduces the response time. With respect to RWA, we reduce the schema automata by filtering the useless statuses and the transform functions according to the following set and the preceding set. In this thesis, two data structures, *path instance tree* and *path schema tree*, were proposed to support the automata match algorithm efficiently. The experimental results show that the AM performs much better than the containment join, and PSA and RWS are

effective optimizations on AM.

Different from the conventional queries in RDBMS, XML queries typically specify twig patterns of selection predicates on multiple elements that have some specified structural relationships. Finding all occurrences of such a twig pattern in an XML database is a core operation for XML query processing. A twig query can be transformed to an automaton. Performance of conventional evaluation approaches based on structural joins declines as increasing of data and query complexity. In this thesis, a novel approach is proposed to compute twig queries by matching twig query automata and path schema trees. Moreover, we give the performance evaluation to demonstrate the high performance of our methods.

Key words: XML, Automata, Schema Automata, Path Expression Query, Twig Query

声　明

本人声明所呈交的学位论文是在导师的指导下完成的。论文中取得的研究成果除加以标注和致谢的地方外，不包含其他人已经发表或撰写过的研究成果，也不包括本人为获得其他学位而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

本人签名：　周博

日　　期：　2004年1月4日

第一章 前言

随着 Web 技术的飞速发展，基于网络技术的应用领域日渐增多，网络信息量也变得越来越大。目前主要用于表示 Web 文档和数据的 HTML 语言着重于数据的外部表现形式，而不是内部的数据结构，已经不能满足 Web 上信息表示和数据交换的要求。XML（eXtensible Markup Language）是一种简单的、自描述的、内容与表现相分离的语言，已经被大家广泛接受作为 Web 上信息表示与数据交换的新标准。随着 Microsoft 在其.NET 体系中全面使用 XML 作为其各部件间信息载体，以及 Oracle，IBM DB2 等大型商业数据库产品纷纷将 XML 的存储和查询功能纳入其核心部分，越来越多的网站以及 Web 应用使用了 XML 进行信息的发布，Web 上用 XML 表示的数据越来越多，XML 已经成为目前 Web 上最受瞩目以及被认为是极具发展前景的技术。然而，由于 XML 的半结构化特性，使得传统的数据管理技术不能完全满足对 XML 数据的存储以及查询管理的需要。因此，研究针对 XML 数据的数据库管理技术势在必行。目前，XML 数据库的主要工作集中在查询处理及优化技术，如路径表达式查询优化技术，Twig 查询优化技术，包含连接查询技术研究，XML 索引技术等，但是由于 XML 具有十分复杂的半结构化特性，目前的研究成果还不能完全支持 XML 标准查询语言 XQuery，因此在世界范围内，无论是学术界还是工业界，对 XML 查询处理和优化技术都正在进行更加深入的研究。

1.1 XML 相关技术

1.1.1 一个 XML 数据实例

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="sample.xsl"?>
<!DOCTYPE site SYSTEM "sample.dtd">
<Act id='1'>
    <Prologue>
        <Act>
            Act in Prologue.
            <Speech><Line>Start the act 1.</Line></Speech>
        </Act>
    </Prologue>
    <Epilogue>
      <Act>
          Act in Epilogue.
        <Speech><Line>Finish the act 1.</Line></Speech>
      </Act>
    </Epilogue>
</Act>
```

图 1.1 XML 实例文档

Figure 1.1 XML sample document

图 1.1 是一个 XML 文档实例，文档的第一行是一个 XML 声明，给出了 XML 文档采用的 XML 版本号，是否和外部 DTD 配合使用，以及数据所采用的编码方式等信息。文档的第 2 行指出了当显示该 XML 文档时所应使用的样式文件。文档第 3 行是文档的外部 DTD 使用说明。从第 4 行起是文档所表示的数据。XML 通过元素来组织 XML 数据，元素是 XML 文档内容的基本单元，XML 元素包括标记和字符数据。从语法角度上看，一个元素包含了一个起始标记、一个结束标记以及标记之间的数据内容。XML 中有元素、属性、文本等几种基本的数据类型，因为本文主要探讨 XML 文档路径查询，因此我们主要研究这几种数据类型。每个 XML 文档有且仅有一个根元素，图 1.1 中的根元素的标签为 Act，它有两个子元素，其标签分别是 Prologue 和 Epilogue。

1.1.2 DOM 模型及 DOM 标准

图 1.2 是与图 1.1 中的 XML 文档相对应的 DOM 树。DOM 模型以有层次的结点对象树的形式来表现文档。这些结点中有的可以有孩子结点，有的只能作为叶子结点存在。通过用图形表示的 DOM 树，我们可以很清楚地了解 XML 文档的数据之间的结构。用带箭头的线段表示实例之间的实际的父子关系或者元素与属性之间的关系。图形中的数字代表这一结点实例的唯一标识（Object ID），如果 DOM 树在内存中实现，它可能是结点的指针或句柄，如果 DOM 被持久化在数据库中，它可能是记录的主键或者对象的 ID。符号下面的字符串标明了元素结点或属性结点的名字。如标记为“9”的结点就对应于前面示例文档中“Act in Prologue”这段文本，它是 Act 元素（编号为 5）节点的子节点。

DOM(Document Object Model)是 W3C 推荐的一组应用于 XML 和 HTML 的编程接口。DOM 定义了文档的逻辑结构以及文档的访问和修改方法。当 XML 文档以 DOM 的方式呈现时，可以被看作数据而不单纯是文件。DOM 规范为程序开发

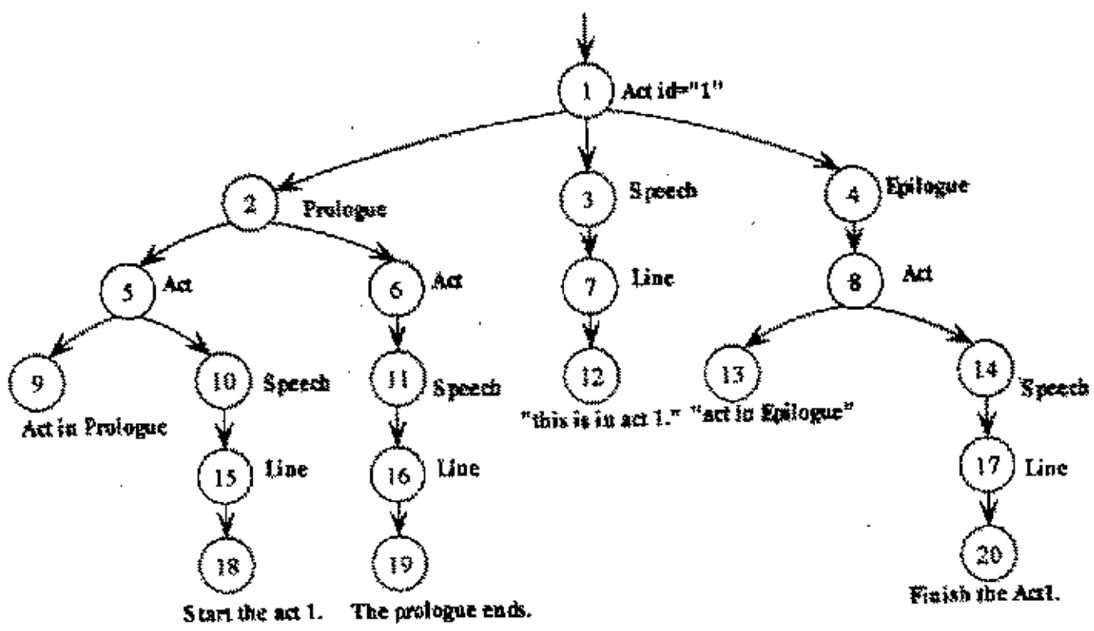

图 1.2 示例文档的 DOM 树

Figure 1.2 DOM tree of sample document

者提供了一系列的应用程序接口的描述，通过 DOM，程序开发者可以创建一个 XML 文档，按文档的结构进行遍历、增加、删除或修改 XML 文档中的元素和内容。W3C 在制定 DOM 接口时即要求该标准具有平台无关性，使它能够在任何平台上以任何语言实现，所以 DOM 标准使用 OMG IDL 编写。DOM 标准定义的所有的 API 都是接口（interfaces）定义而不是类（class）定义。这意味着标准的任何实现都只需要按照名字和定义好的操作来实现该方法。

1.1.3 DTD 简介

图 1.3 给出了一个与图 1.1 中的 XML 文档对应的 DTD 图。

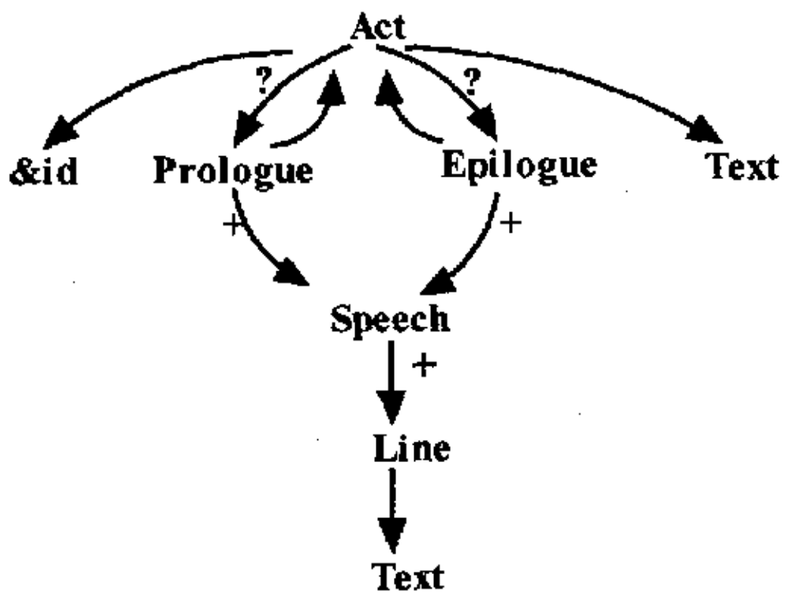

图 1.3 示例文档的 DTD 图

Figure1.3 DTD Graph of sample document

DTD 是 W3C 推荐的针对 HTML 和 XML 文档的类型定义标准。它对 XML 文档的层次结构和内容做了描述。XML 解析器利用 DTD 来验证 XML 文档的正确性。虽然 DTD 提供了一定的模式信息，但是还十分不全面，为了解决这一问题，W3C 又提出了 XML Schema 标准。另外，DTD 本身是可以动态改变的。这就为 XML 的扩展提供了方便。

XML 的本质意义是通过提供给文档的编写者以制定基于信息描述、体现数据之间的逻辑关系的能力来保证文档的语义清晰和易检索性。因此，一个 XML 文档不仅要是“格式良好的”，而且还应该是使用了一个自定义标记的“有效的”XML 文档，也就是说，它必须遵守文档类型定义 DTD 中已经声明的规定和限制。DTD 描述了标识语言的语法和词汇表，也就是定义了文档的整体结构以及文档的语法。也就是说，DTD 实际上规定了一个语法分析器为了解释一个“有效的”XML 文档所需要了解的全部细节。一个 DTD 可以是内部的，包含在 XML 文档的前面说明部分；也可以是外部的，作为一个外部文档被引用。外部 DTD 的好处是可以被多

个 XML 文档方便地共享。表 1.1 给出了图 1.3 对应的 DTD。

表 1.1 实例文档的 DTD

Table 1.1 DTD of sample document

```
1  <!ELEMENT ACT       (PROLOGUE?, EPILOGUE?)>
2  <!ELEMENT PROLOGUE  (SPEECH+, ACT*)>
3  <!ELEMENT EPILOGUE  (SPEECH+, ACT*)>
4
5  <!ELEMENT SPEECH    (LINE+)>
6  <!ELEMENT LINE      (#PCDATA)>
```

从表 1.1 中可以看到：ACT 元素包含两个子元素 EPILOGUE 和 PROLOGUE 它们后面各自的一个“?”称为元字符。表 1.2 列出了 DTD 的正则表达式中可能出现的元字符及其意义。

表 1.2 DTD 中的元字符

Table 1.2 Meta-characters in DTD

元 字 符	含 义
元素 A 元素 B 元素 C	元素列表，无须遵从顺序要求
,	并（AND），要求严格遵从顺序要求
+	出现一次或多次
*	出现零次或多次
()	一组要共同匹配的表达式
|	或（OR）
?	可选，不出现或出现一次

1.2 XML 数据的存储策略

上面给出的图 1.2 中的 DOM 树只给出了 XML 数据在数据库中存储的逻辑模型，而 XML 数据在数据库中的物理存储却是因数据库的不同而差别很大，归纳起来，主要有以下四种方式：文本文档格式，关系表格式，对象类模式和专门为 XML 设计的存储模式。对应于不同的存储模式，也有不同的数据管理技术。以上四种方法由于出发点不同，在存储，查询的效率上各有所长。

除了最近出现的XML数据流，XML数据的表现形式基本都是XML文本文件，因此管理 XML 数据的最直接的方法就是将其存储于文件系统中。目前有三种方式是基于文本的存储方式的：基于信息检索技术（Information Retrieval，IR）的关键词查询和用户定制的基于 DOM 或 SAX 接口的用户定制查询。这种存储方式的优点是：存储方便，成熟的 IR 技术可以直接得到应用，可以使用基于 DOM 或 SAX

的工具。缺点有很多：IR 方法仅仅是对关键词的查询，而忽视了结构信息，也就是忽视了 XML 数据的本质。因为 DOM 占用的内存将是原文档大小的 10 倍左右，所以将文档以 DOM 的形式解析到内存进行查询的方式只适合于小文档，而且每一次查询都需要重新解析 XML 数据。

基于关系数据库的管理方式由于关系数据库产品本身在数据库市场的统治性地位而成为 XML 数据管理技术中最重要的一种，因此也有许多研究注重于这一方面。当我们将 XML 数据存入关系数据库中时，必须根据 XML 数据的模式为其设计若干子模式做为存储 XML 数据的表模式。使用关系数据库管理 XML 数据主要包含以下几个步骤：

1. 为 XML 数据定义合适的表模式，并将其存储到关系表中。
2. 利用文档 DTD 信息和文档本身的信息将 XPath(XQuery)查询语句转化为 SQL 语句。
3. 查询优化，查询执行。
4. 将查询结构以 XML 的格式提交给用户。

使用关系数据库技术来管理 XML 有两个比较明显的缺点，一是 XML 数据是半结构化的，而关系表是二维表结构，存储和查询都不可避免的要在两种格式之间相互转化。另外，XML 查询语言支持许多复杂的语义，比如正则表达式中的闭包操作符，这在 SQL 中是找不到与之对应的部分的，要想解决这个问题，避免不了要做多次表连接操作，查询效率会受到很大的影响。

ODMG 标准为面向对象数据库定义了对象模型(Object Model)，对象定义语言(ODL)和对象查询语言(OQL)。因为 XML 在描述数据时采用在不同层次上的嵌套关系，理论上将这种树形结构数据映射为嵌套对象模型比映射为关系表更为自然，而且对象路径查询也与 XML 数据的遍历策略很相似，因此一些系统采用了对象技术来管理 XML 数据。由于对象数据库在高效地处理大规模数据方面相对于关系库来说还不成熟，因此还有很多需要完善的地方。

XML Native 数据库技术以斯坦福大学的 LORE 系统[2]为代表，其它比较著名的还有 Tamino,TIMBER[20]等。这些系统专门为 XML 这一特殊的数据类型设计存储模型与索引技术。对于 XML 路径表达式的查询处理，比较典型的是三种基于 DataGuides 的查询算法，自顶向下(Top-down)、自底向上(Bottom-up)和混合策略(Hybrid)。自顶向下方法是从一个 XML 文档树的根节点开始遍历直到找到正确的结果；自底向上则是借助值索引的帮助从文档树底部向上遍历直到根节点；而混合策略则是首先把一个长路径分割成若干子路径，对每个子路径可以使用自顶向下或自底向上计算，最后对所有子路径结果做基于元素父子关系的连接操作，以得到最终结果。由于 XML Native 数据库采用了针对 XML 本身特点而设计的存储方案和查询策略，因此，虽然它现在还不算成熟，但被认为是四种方案中最有前

途的解决方案。

1.3 XML 查询语言

随着利用 XML 格式存储和表示的信息日益增多，在各种各样的 XML 数据源中查询 XML 数据的需求也越来越迫切。XML 的提出是为了灵活的表示不同数据源的不同信息。为了使这种灵活性得到最大程度的发挥，研究者设计了许多 XML 查询语言来从 XML 中查询数据。其中最有代表性的是 W3C 发布的 XML 路径语言 XPath 和 XML 查询语言 XQuery，它们首先为 XML 文档提供一个数据模型，并且提供了一组基于这个模型的查询操作。XPath 是 XQuery 的一个子集。

1.3.1 XPath 规范

XPath[4]提供了关于路径导航(path navigation),节点定位(node location)，谓词选择等方面的一份精确的规范，并且被作为 XQuery 的核心部分。XPath 中最重要的组成部分是定位路径(location path)，具体分为绝对定位路径(absolute location path)和相对定位路径(relative location path)两种。前者是从根节点到某个节点的路径，后者是从当前选定的节点的路径到某个节点的路径。在 XPath 中，路径表达式是其核心组成部分。对于一个路径表达式，可以写成 Path:=(step)*的形式，而每个 step 又可以定义为 step:=Axis::NodeTest[Predicate]的形式。在每个 step 中，轴(Axis)专门指定了文档定位的“方向”。XPath 支持 11 种定位轴(不考虑 attribute axis 和 namespace axis)。节点测试 NodeTest 指选择 XML 文档中满足该模式的节点。谓词 Predicate 进一步限制了节点测试的选择范围。XPath 中的轴分成两类：前向轴(ForwardAxis)和反向轴(BackwardAxis)，其中前向轴包括 child, descendant, self , descendant-or-self , following-sibling , following 共 6 个轴；反向轴包括 parent , ancestor , preceding-sibling , preceding , ancestor-or-self 。本文以讨论前向轴为主。

XPath 和 XQuery 中最基本的构筑部分是表达式。它们提供了若干种表达式，这些表达式由关键字、符号和操作数组成。通常情况下，一个表达式的操作数是另外一个的表达式。XPath 和 XQuery 是函数性的查询语言，它们允许多种表达式任意层次的嵌套。同时，它们也是强类型的语言，其表达式中的操作数、操作符和函数等等必须对应有特定的类型。

在 XPath 中，有以下类型的表达式：

1. 主表达式（Primary Expressions）。主表达式是 XPath 语言的基本原语，包括变量、字面量、函数调用等等。
2. 路径表达式（Path Expressions）。路径表达式是 XPath 语言的核心，它用来在

XML 树上定位结点，按照文档序返回一个不重复的结点序列。路径表达式的计算必须依靠表达式上下文。

3. 序列表达式（Sequence Expressions）。序列表达式用来在 XPath 语言中构筑和合成序列，序列是一个包含零个或更多元素的有序集合，序列中的元素可以是一个简单的值或者是一个结点。
4. 算术表达式（Arithmetic Expressions）。算术表达式提供了 XPath 语言中的加、减、乘、除等运算。
5. 比较表达式（Comparison Expressions）。比较表达式用来支持“大于”、“小于”以及判等等一系列的比较运算。在 XPath 中，比较运算包括值的比较、序列的比较、文档结点的比较以及文档序的比较等等。
6. 逻辑表达式（Logical Expressions）。逻辑表达式表示逻辑上的与、或、非等运算，返回值是一个逻辑值。
7. 迭代表达式（For Expressions）。XPath 提供了迭代表达式用来执行迭代运算，这种表达式经常用来计算两个或者更多文档之间的连接以及数据的重构。
8. 条件表达式（Conditional Expressions）。条件表达式允许根据不同的条件而使表达式返回不同的值。
9. 量词表达式（Quantified Expressions）。量词表达式允许通过全称量词或存在量词测试表达式的值，其返回结果是一个逻辑值。

1.3.2 XQuery 简介

XQuery 被设计成用来实现 W3C 的标准“XML 查询要求”（XML Query 1.0 Requirements）。XQuery 是一种小的、易于实现的语言，用它来表示的查询简单并且容易理解。XQuery 十分灵活，可以在包括数据库和文档的多种 XML 数据源之上进行查询。XQuery 的 1.0 版本包含了 XPath 的 2.0 版本作为它的一个子集。所有在 XQuery1.0 和 XPath2.0 中语法上均有效而且均可成功执行的表达式，在这两种语言中必然得到相同的结果。由于这两种语言有着如此紧密的联系，它们在语法和语言的描述以及数据模型的表示上都是统一的。

XQuery 中包含了上面描述的 XPath 中的所有表达式，除此之外还包括 FLWR 表达式和排序表达式等等。

1.3.3 正则路径表达式

路径表达式查询是 XML 查询最基本的要求，几乎所有的 XML 查询语言的核心都是基于路径表达式的查询，而且很多关于 XML 的查询处理器都是针对路径表

达式的。各种查询语言及查询处理器使用的路径表达式基本上很相似，其中最常使用的就是使用正则路径表达式(RPE Regular Path Expression)，正则路径表达式是正则表达式在路径查询中的应用，其形式可以如表 1.3 定义：

从表 1.3 中我们可以看出，一个 RPE 可能是一个绝对的 RPE（AbstRPE）也可能是一个相对的 RPE（RltvRPE）。一个绝对的 RPE 表示对文档查询要从 XML 文档的根元素开始，而一个相对的 RPE 可能从 XML 文档中的任一位置开始查询，查询的具体位置要根据查询时的上下文动态决定。在表示形式上，绝对的 RPE 由表示根元素的符号“/”和一个相对的 RPE 表示。无论是绝对的 RPE 还是相对的 RPE，都是由 RPEStep 和 RPE 操作符构成的。RPEStep 表示了 RPE 查询中的一个单一操作，如取得某个元素结点的孩子，或者取得这个结点的指定名字的属性结点等。一个 RPEStep 可能带有限定部分，限定部分表示了一些比较复杂的操作，其中最常见的就是谓词。在查询处理中，谓词意味着对阶段性的查询结果根据制定的条件执行过滤操作。RPE 操作符用来把 RPEStep 合成复杂的 RPE，这些操作符所代表的运算可以任意嵌套，是完全正交的。RPE 中常见的操作符和它们所代表的意义如表 1.4 所示。

表 1.3 RPE 的 BNF 定义

Table 1.3 BNF syntax of RPE

```
RPE ::=        AbstRPE | RltvRPE
AbstRPE ::=    '/' RltvRPE
RltvRPE ::=    RltvRPE '/' RltvRPE
               | RltvRPE '|' RltvRPE
               | RltvRPE '*'
               | RltvRPE '+'
               | '(' RltvRPE ')'
               | RPEStep
RPEStep ::=    SmplRPEStep
    { NodeQualifier }
SmplRPEStep ::=  Name
                 | '@' Name
                 | 'text()'
NodeQualifier ::=  '['Predicate']'
```

W3C 的标准 XPath 也是一种基于路径表达式的查询语言，XPath 的路径表示非常类似于上面定义的 RPE，最明显的差别是 XPath 中没有闭包运算（没有“*”和“+”这两个操作符），取代它们的是 ancestor-descendant 操作符“//”，它代表了 XML 文档中结点的祖先后代关系，如“/site//item”就表示所有具有“site”祖先的名为“item”的结点。

表 1.4 RPE 操作符

Table 1.4 RPE operators

操作符	操作符的意义
/	连接两条路径
|	合并两条路径（或操作）
*	闭包运算（零次或多次）
+	非空闭包运算（一次或多次）
()	改变表达式优先级

利用 XML 文档的 DTD，任何查询表达式中的“//”操作符都可以重写成由表 1.3 中的 RPE，从而消除路径中的“//”操作符。这种重写的策略是一种解决“//”操作符比较常用的方法，本文中提出了利

用模式自动机来解决“//”和在模式空间中遍历路径模式树来解决“//”的两种方法，这两种方法都不需要人为去统计文档信息。

1.4 XML 路径查询概述

1.4.1 包含连接技术简介

在处理路径查询的各种方法中，XML 文档树遍历是最基本，最简单的一种。它以某种特定的顺序遍历并检查 XML 文档树中的每个节点，看其是否满足给出的查询表达式。不难看出，XML 文档树遍历的方法查询效率是很低的。另外一种比较流行的处理方法是包含连接（Containment Join）的方法，人们通常也称其为结构连接。采用包含连接方法的基本思想如下：

1. 先对 XML 文档树中的每个节点进行编码并将编码存储在数据库中。
2. 将一条查询语句分解成若干个二元关系查询（即父子关系和祖先－后代关系），对每一个二元关系进行计算，得到查询的结果。
3. 将 2 中得到的结果连接起来得到最终结果。

为了解决上面的第 2 环节，在文献[31]中提出了一种传统的关系数据库中的归并连接算法的变体，称为多谓词归并连接算法(MPMGJN)。该算法依赖于将 XML 文档中的每个元素按(startPos,endPos,level)三元组编码得到的索引结构，实验证明，在关系数据库中，MPMGJN 算法比传统的归并连接快一个数量级左右。

在执行每个二元关系查询时通过索引对节点的编码进行访问，并按照特定的连接算法（如 MPMGJN）进行连接。通常将参与连接的两个集合分别称为祖先集合(Alist)和后代集合(Dlist)。参考文献[23]中提出了一种遍历两个集合各一次就可以找到连接全部结果的高效 StackTree 算法。后来研究者们进一步提出了一些索引结构如 B＋树[11]和 XR 树[21]来进一步提高 StackTree 算法的效率，其基本思想是在查询过程中通过索引来跳过一些一定不会产生结果的元组来加快查询速度。

在上述第 3 个步骤中，(2)中的每个二元连接都得到一个中间结果参与最后的连接。在连接过程中，有些情况下，即使每个二元连接的结果和最终的查询结果并不是非常大的时候，也会产生存储代价巨大的，难以处理的中间结果。为了解决这个问题，人们提出了不同的解决方案，从传统的查询优化角度来看，有的研究者[17,29]通过估计每一种连接顺序的代价来找到一种最优的连接次序。而文献[6]中提出了利用若干个堆栈来来消除中间结果的 PathStack 算法。包含连接系列算法的出现很好地解决了过去比较难解决的祖先－后代关系问题（XPath 规范中的“//”操作符）。包含连接算法的优点是很容易结合在关系数据库中，其效率也比遍历文档树的方法有较大提高。但是，包含连接算法并不是十全十美，比如，包含连接算法并不支持正则路径表达式中的闭包运算符（“*”，“+”等）。

1.4.2 XML 路径查询中所采用的编码策略

为了支持包含连接系列算法，在索引的建立过程中，人们采用了不同的编码策略。

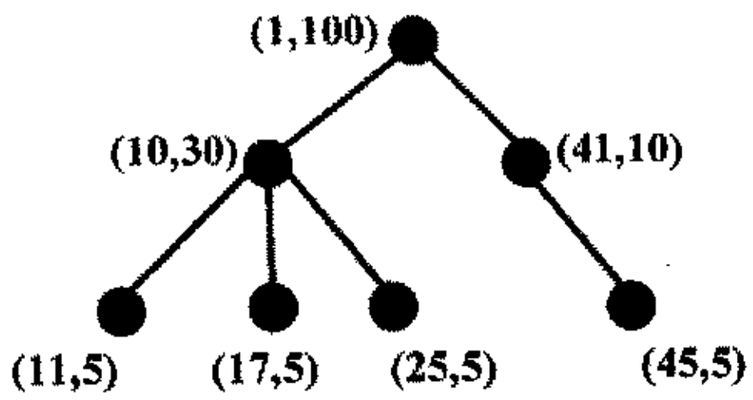

图 1.4 扩展前序编码

Figure1.4 Extended preorder numbering scheme

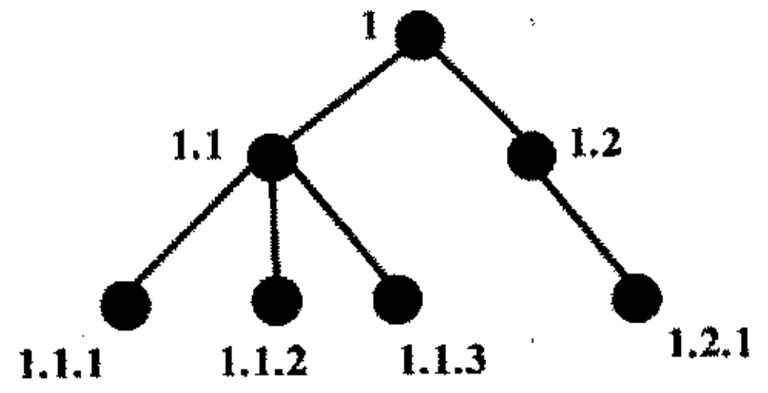

图 1.5 DEWEY 编码

Figure1.5 DEWEY numbering scheme

如图 1.4 所示，扩展前序编码由一个二元组（start,interval）组成，其中 start 对应一个节点在树中唯一的一个前序编码，而 interval 则决定了该节点的后代节点的 start 和 interval 的取值范围。对于用图 1.4 中的方法进行编码的 XML 文档中的两个节点 X，Y。我们定义 X 是 Y 的祖先，当且仅当：

1. start(X) < start(Y)
2. start(X)+internal(X) >= start(Y)+internal(Y)

定义 X 是 Y 的父亲，当且仅当：

1. start(X) < start (Y)
2. start(X) +internal(X) >= start(Y)+internal(Y)
3. depth(X) = depth(Y) –1

其中，depth 为该节点所在的层数，也就是说，为了确定节点间的父子关系，我们需要将二元组(start,interval)扩展为(start,interval,depth)。

图 1.5 中采用的编码策略一般称为 DEWEY 编码，它所体现的按层次编码的思想被广泛地应用到日常生活中，比如图书馆中书籍的索书号就是 DEWEY 编码的实例。如图所示：每个节点被赋予一个向量 v，该向量的值唯一标识了一条从根到该节点的路径，通过向量 v，我们也可以很容易的确定节点 X,Y 之间的关系：

1. X 是 Y 的祖先 :left(v(Y) , length(v(X))) = v(X)
2. X 是 Y 的父亲：left(v(Y) , length(v(Y)) –1) = v(X)
3. 类似地，我们还可以很容易地得到判断两个节点间是否有兄弟关系的条件表达式。

DEWEY 编码比扩展前序编码的优势在于，DOWEY 编码可以很容易的判断兄弟关系和某个节点是其父亲的第几个孩子。DOWEY 编码的缺点是其编码长度随着树层数的加深而变长，不适合对大规模的 XML 文档进行编码；而且当有新的节点插入时，新节点的一部分兄弟节点和兄弟节点的后代节点的编码需要更新，而

扩展前序编码由于其 interval 的设计，插入节点需要更新的可能性则低的多。

1.4.3 自动机技术在 XML 处理中的应用

由于自动机技术处理数据的结构和数据的模式十分方便，因此也经常应用到 XML 处理系统中，其中具有代表性的是 XFilter 系统[1]和 YFilter 系统[12]。

XFilter 和 YFilter 系统都属于信息过滤系统的实例。信息过滤系统产生的背景是如何使网络上数量极为巨大并且还在以飞快的速度增加的信息更好的为用户所用的问题。为了解决这个问题，用户需要这样一种机制：当且仅当用户感兴趣的文档到达时，用户会得到系统的通知和他所感兴趣的文档，也就是说，实现一种“个性化”的信息分发模式。提供这种功能的系统在日常生活中有很多现实应用：比如将股市信息，交通信息等发送给大量的对其感兴趣的用户。这种选择分发信息系统的模型是：实时地从数据源接收数据，将这些数据通过用户的 Profile 文件进行过滤，然后系统把相关数据传递给对其感兴趣的用户。为了有效地将正确的信息传递到正确的用户那里，系统依赖于用户将其感兴趣的信息模式存储在用户 Profile 文件中。目前的大多数过滤系统都是依赖于关键词检索，这种方式表达用户需求的能力十分有限，而且其效率也跟不上 Web 上信息的更新速度。，因此 XFilter 系统采用 XPath 形式的 Profile 文件，这些 XPath 表达式相对于 Web 上千变万化的信息来说变化是很小的，而且采用 XPath 既可以体现出用户的感兴趣的关键词，又可以包含结构信息。

在 XFilter 系统中，Altmel 和 Franklin 将自动机技术应用于大型信息发布系统中，用来进行 XML 文档的过滤。XFilter 系统设计了一个名为 Query Index 的倒排索引，利用基于 XML Parser 的 SAX 接口驱动过滤引擎，根据 XPath 语言输入的查询条件对大量的 XML 文档进行过滤。在他们的设计中，自动机用来表示用户的配置文件，并且利用一个基于 SAX 接口的解析器来解析 XML 文档同时激活自动机，进行文档的过滤。XFilter 系统主要由 4 部分构成，分别是：用来解析 XML 文档的基于事件的解析器，用来解析用户的 Profile 文件的一个 XPath 的解析器，过滤引擎和分发组件。YFilter 系统与 XFilter 一样，也是用自动机形式的配置文件来达到对 XML 文档的过滤，分发目的，与 XFilter 相比，YFilter 最大的改进在于：在 XFilter 系统中，每一个查询语句对应一个自动机，这种方法没有充分利用到相同的正则表达式片断转化成的自动机片断可以合并的特点。YFilter 采用的方法是把所有的查询语句转化为一个自动机，它利用了大量路径表达式中可能出现许多相同的前缀这一特点将表达式中相同的部分合并到一起，这样，这个共同的前缀最多只会被匹配一次。这种优化给一个大规模的信息过滤系统带来的效率提升是很明显的，因为系统支持的用户越多，这些用户之间出现对某些东西共同感兴趣的可能性越大，通过对这一部分可以合并减少自动机的状态，提高过滤分发的效

率。除此之外，通过合并表达式中相同的部分，减少了最终结果中自动机的状态，也为将自动机存储在磁盘上节省了空间。

1.5 XMark 测试标准

为了更好地评价 XML 数据库系统中相关算法的功能和效率，人们提出了很多针对 XML 数据处理的测试标准与数据集，例如 XMark，Xmach-1，DBLP，Shakes 等。本文在性能测试的环节中，除了选用自行定义的 DTD 生成的 XML 文档外，还采用了 XMark 这个测试集。

XMark 用于评估 XML 数据库在实际应用中对不同查询的处理能力，XMark 提供了一系列的查询，每个查询都可以测试查询处理器和存储引擎的某个方面性能的优劣，有的查询侧重于文本信息的查找，有的查询着重于结构信息的分析。文档的元素类型主要有：annotation, person, open_auction, closed_auction, item, category 等。

XMark 中提出了几个比较特殊的测试方向：第一，测试了针对 XML 文档中元素顺序的查询：例如，查询 XML 文档中某元素的第 4 个孩子。第二，XMark 以字符串作为最基本的数据类型：因为字符串的长度差异很大，因而可能出现额外的存储问题。第三，包含层次结构的复杂路径表达式的查询，其是 1:n 的关系而且关系间有序。在关系数据库系统中，这种查询需要昂贵的连接和集合操作。

第二章 自动机匹配查询处理技术

2.1 自动机匹配查询处理技术的提出

随着 XML 越来越普及，许多针对 XML 数据的查询语言被提出，这些语言的一个共同的特点是：路径查询是其核心组成部分。通过研究，我们发现：大多数支持 XML 路径查询的方法都是在实例空间中进行，也就是说，使用这些方法进行查询时面对的是 XML 数据本身，这类方法中，具有代表性的是 XML 文档树遍历的方法和包含连接的方法。

XML 是一种半结构化的语言，它不仅支持基于内容的查询，而且还支持数据的结构查询。因此，大多数的 XML 查询语言都把结构查询作为它们的核心组成部分。在 Xpath 标准中，路径表达式由定位阶（Location Step）组成，阶与阶之间由路径操作符相连。不同的路径操作符具有不同的属性，我们的目标就是为各个路径操作符的计算找到正确有效的方法。

我们知道，XML 文档是通过 DTD 或者 XML Schema 进行定义的，当 XML 文档变得很大时，DTD 和 XML Schema 本身并不会改变很多。因此，如果找到一种查询方法可以运行在模式空间（该查询方法只跟 DTD 或 Schema 相关，与 XML 数据本身无关），它的效率就不会随着 XML 文档规模的增大而下降。基于以上出发点，我们提出了运行在模式空间（schema space）中的自动机匹配(automaton match)的方法。它的大致思路如下：首先通过遍历 XML 文档建立一棵路径模式树作为索引，并把查询表达式转化为查询自动机；令路径模式树中的模式节点和查询自动机的状态相匹配，查询的结果可以从自动机中止状态对应的模式树上的节点得到。可以看出，除了在最后取结果阶段需要访问实例节点之外，自动机匹配算法不需要访问实例节点。实验证明：自动机匹配算法在大规模的 XML 文档查询上性能优势明显。

大多数的 XML 路径查询系统允许路径表达式中支持嵌套结构。我们发现用普通的方法解决某些特定的路径操作符是非常低效的，因为查询处理的效率与路径表达式的复杂度密切相关。正则表达式中定义的闭包操作符在查询嵌套结构的 XML 查询时非常有用，而闭包操作符的计算用通常的方法很难很好地支持。有些人提出的将闭包操作符重写为多次的连接的方法可行但效率欠佳。根据经典的自动机理论,每个正则表达式都对应着一个与之等价的自动机，而闭包操作符很自然可以转化为自动机的一部分，所以自动机匹配的方法可以很方便地用来计算 RPE 中的闭包操作符。

2.2 自动机相关技术简介

作为计算机科学的一种基本工具，自动机技术广泛应用在程序编译、模式匹配以及数据库查询处理等许多领域，它是一种十分有用而且相当成熟的技术。这一节介绍了自动机的基本概念、自动机技术在 XML 处理中的应用以及输入的 RPE 与自动机之间的转换。

2.2.1 查询自动机基本概念

定义 1 一个确定的有限自动机 DFA（Deterministic Finite Automaton）M 是一个 5 元组：$M = (K, \Sigma, \delta, q_0, F)$。其中 K 为自动机的状态集合，$\Sigma$ 为字母表，$q_0 \in K$ 为自动机的开始状态，$F \subseteq K$ 是终止状态集。δ 为状态转移函数，$\delta: K \times \Sigma \rightarrow K$。自动机 $M = (K, \Sigma, \delta, q_0, F)$按照如下方式工作：

1. 一开始，自动机处于初始状态 q_0。
2. 自动机读取一个符号 $a \in \Sigma$，假设自动机的当前状态为 q，并且 $\delta(q,a) = p$ $(p, q \in K)$，则自动机的当前状态变为 p。如果 $\delta(q, a)$无意义，则转 5。
3. 如果还有未读取的符号，则转 2，否则如果自动机的当前状态为终止状态，即 $q \in F$，则转 4，否则转 5。
4. 自动机成功接收句子。
5. 自动机接收句子失败。

以上定义的状态转移函数用来说明自动机从某个状态读取某一符号而到达的状态，更一般的，状态转移函数 δ 可以扩充到读取一个句子。我们定义 $\hat{\delta}: K \times \Sigma^* \rightarrow K$，$\hat{\delta}(q, \varepsilon) = q$，$\hat{\delta}(q, \omega a) = \delta(\hat{\delta}(q, \omega), a)$。为了表示上的方便，我们不加区分的使用 δ 和 $\hat{\delta}$。

接下来是正则表达式的概念，一个正则表达式代表一个集合，如果这个集合等于一个自动机所接收的语言，说明这个正则表达式与这个自动机等价。

定义 2 设 Σ 为字母表，Σ 上的正则表达式 r 代表一个集合，这个集合记做 L (r)，且 $L(r) \in seq(\Sigma)$，正则表达式及其所代表的集合递归定义如下：

1. φ 是一个正则表达式，它代表空集。
2. ε 是一个正则表达式，它代表集合$\{\varepsilon\}$。
3. 对任意 $a \in \Sigma$，a 是个正则表达式，它代表集合$\{[a]\}$
4. 若 r，s 分别是代表集合 R，S 的正则表达式，则 r|s，r/s，r*分别是代表 RS，RS，R*的正规表达式。

定理 设 r 是 Σ 上的正则表达式，都可以找到 DFA M，使得 $T(M) = L(r)$，在此情况下，称正则表达式 r 与 DFA M 等价。

这个定理说明任何正则表达式，都可以找到与之等价的 DFA。定理的证明涉及到许多其它的概念，如非确定有限自动机等，可以在[19]上找到，而且这个定理的证明不是这里的重点，我们只是直接利用这一结论，把用户查询输入的 RPE（也就是正则表达式）转换成与之等价的自动机。

自动机尤其是自动机的状态转移函数的表现方法有很多种，其中最常用直观的就是状态转移图。图 2.1 就是一个自动机的状态转移图。假设图 2.1 的状态转移图代表自动机 M = (K, Σ, δ, q_0, F)，下面用这个例子说明状态转移图的意义。图中的圆形表示自动机的状态，用圆形中的数字表示，同一状态在图中只出现一次，所以自动机的状态集合 K = {0, 1, 2}。连接状态的带箭头的线段表示状态的转移，读取的符号标注在线段上，因此字母表 Σ = {a, b}。线段的意义是从一个状态出发，读取某一符号后，转移到另一个状态，用这种形式表示状态转移函数，因此在此例中，δ (0, a) = 1，δ (1, b) = 2，δ (2, a) = 0。不从任何状态出发的线段标识了自动机的初始状态，即 q_0 = 0。边界线为双线的圆形为自动机的结束状态，本例中 F = {2}。

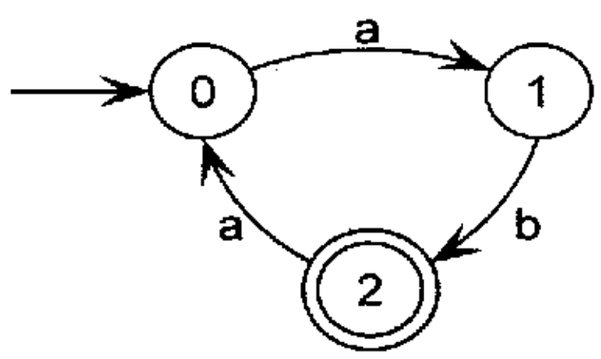

图 2.1 自动机状态转移图

Figure 2.1 Status transaction graph of automaton

图 2.1 所表示的自动机可以接收的句子有无穷多个，如[a, b]，[a, b, a, a, b]等。该自动机还与一个正则表达式等价：a / b / (a / a / b)*。

2.2.2 路径模式与路径模式树

在下面即将提到的路径表达式查询的自动机匹配算法 AM 中，XML 路径模式被当作查询的目标—使用路径表达式去匹配查询自动机，被接受的路径模式对应的路径实例的最后一个元素为查询结果。然而，在没有有效的路径模式索引结构的情况下，逐一地为 XML 文档中每一条路径进行模式匹配的效率非常低。

一个 XML 文档中的所有结点构成了一个集合。如果用变量 *D* 表示一个 XML 文档，那么我们用 Node(*D*)表示文档 *D* 中所有结点构成的集合，其中包括元素结点、属性结点和文本结点等。根据这些结点所表示的数据，可以把这些结点分为不同的类型，通常称为节点的模式。对于元素结点来说，每个元素结点都有名字，或者称为标签，具有相同元素名的 XML 结点表示同一类型的数据，如元素名为 name 的结点可能用来表示人、组织或者物品的名字，元素结点的模式就用该元素结点的名字或者标签表示。类似的，对于属性结点来说，属性结点的名字就是该结点的模式，与元素结点不同的是，属性结点并不是独立存在的，都是依附于某一元素结点的，不同名的元素结点的同名属性有不同的逻辑意义，属于不同的属

性。文本结点用来存储数据，没有名字或标签，我们可以认为它们属于同一类，具有相同的模式。因此，字符串可以用来标识结点的模式，唯一需要做的是用加前缀或者后缀的方法区分以上三种结点。每个 XML 文档中的节点都对应着一个节点模式(node schema)，节点模式描述了节点的类型。举例来看：一个标签为“book”的 XML 节点和一个标签为“name”的节点被认为有不同的类型，所以它们的节点模式也不同。为了方便，下面我们将节点模式简称为模式(schema)。定义 3 给出了 XML 文档结点的模式(schema)的定义，定义 4 定义了 XML 文档的结点模式集合，这个集合包含了 XML 文档中所有结点的模式。定义 5 将模式的定义扩展到一个序列，因为一个结点的模式是一个字符串，所以一个结点序列的模式是一个字符串的序列。由于 XML 数据以 DOM 树的形式加以组织，DOM 树上任一条路径均是 XML 结点的序列。DOM 树上的任一条绝对路径的模式被称作路径模式，定义 6 给出了该定义。最后，定义 7 定义了一个 XML 文档的路径模式集。

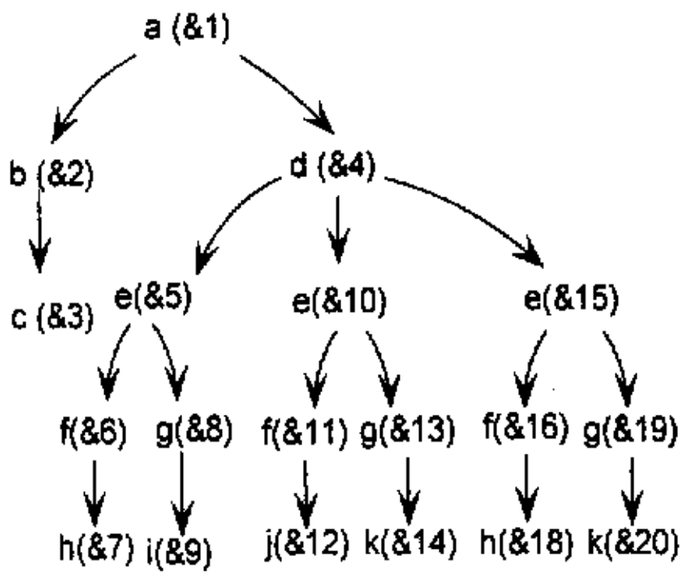

图 2.2　XML 数据的 DOM 存储结构

Figure2.2 DOM tree of sample document

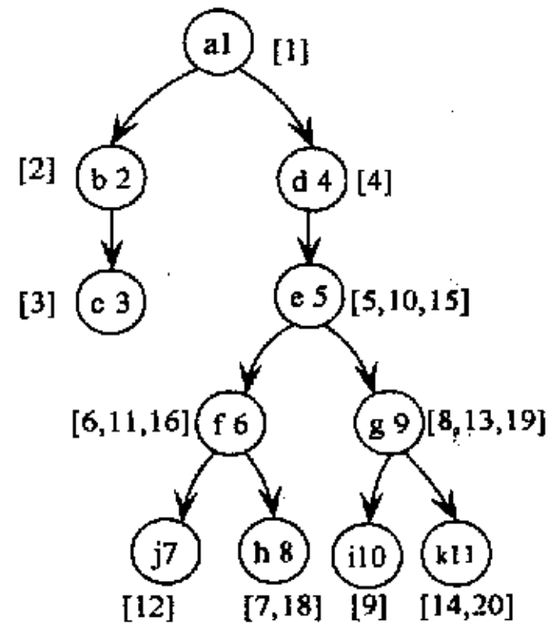

图 2.3　实例文档的路径模式树

Figure2.3 path schema tree of sample document

定义 **3** 给定 XML 文档 *D*，对文档中的任意结点 m ∈ Node (*D*)，其模式用 schema 表示，其值是一个字符串。如果 m 是一个元素结点，则 schema (m)的值是该元素结点的名字；若 m 是一个属性结点，则 schema (m)的值是该属性结点的名字加上前缀“@”；若 m 是一个文本结点，则 schema (m)的值是一个固定的字符串“text()”。

定义 **4** 给定 XML 文档 *D*，该文档的结点模式集为该文档中的所有结点的模式集合，即 schema (*D*) = {schema (m) | m ∈ Node(*D*)}。

例如，在图 2.2 中 schema (&1) = a，schema (&8) = g。文档结点模式集 schema (*D*) = {a,b,c,d,e,f,g,h,i,j,k}。

定义 **5** 给定 XML 文档 *D*，对于文档中的结点 m ∈ Node (*D*)，仅包含一个 m 的

单元素序列的模式为仅包含 m 的模式的单元素序列，即 schema ([m]) = [schema (m)]；对于任意两个由 XML 文档中结点所构成的序列 α，β ∈ seq (Node (*D*))，它们之间的连接运算的模式为它们各自的模式连接之后的结果，即 schema (α + β) = schema (α) + schema (β)。

定义 **6** 给定 XML 文档，对于文档中的结点 m ∈ Node (*D*)，其对应的路径模式为到达 m 的绝对路径的模式，即 pathschema (m) = schema (abspath (m))。

例如，图 2.2 中&12 的路径模式是[a.d.e.f.j] 。显然，每个 XML 数据树节点对应一条绝对路径，即对应一个路径模式。不同的路径可能对应相同的路径模式，而同一路径模式可能对应多条路径实例。举例说明，图 2.2 中节点&7 对应的路径实例[&1, &4, &5, &6, &7]和节点&18 对应的路径实例[&1, &4, &15, &16, &18] 拥有相同的路径模式[a, d, e, f, h]。

定义 **7** 给定 XML 文档 *D*，文档的路径模式集为文档中所有结点对应的路径模式的集合，即 Ψ_D= {pathschema (m) | m ∈ Node (*D*)}

很显然，Ψ_D 是 Σ_D^*的子集。本文用小写字母 α，β 来表示路径模式；*Inst*(α)表示所有路径模式为 α 的节点实例组成的集合。每一个 XML 节点对应一个路径模式，反之，每个路径模式又对应一个节点集合，这个集合称为该路径模式的实例节点集。对于一个路径模式 α ∈ Ψ_D，*Inst*(α)代表了 α 的实例集合。

我们采用持久化 DOM 模型来存储 XML 文档。图 2.2 给出了一个以 DOM 为存储模型的 XML 文档片断示意图。为了集中说明自动机匹配算法，我们假设该文档片断中只有元素类型的结点。图 2.2 中的数字（“&”后面）代表这一结点实例在数据库中的唯一标识，字母表示了元素结点的名字。

路径模式树用来索引所有在 XML 文档中出现的路径模式。模式树中的每个结点唯一对应着一个路径模式。图 2.2 对应的路径模式树如图 2.3 所示：圆圈中的数字和字母分别代表模式树结点的序号和元素名，方括号中的结点代表该模式树结点对应的实例结点。从模式树的根到某个模式结点的路径上的结点的标签序列唯一确定了该结点的路径模式。在实例树中所有满足该模式的实例结点的集合构成了该模式结点的外延。

定义 **8** 给定路径模式树中的一个节点 PS，它对应的路径模式设为 α，则路径实例树中所有满足模式 α 的实例节点组成的集合构成了 PS 的外延，即 *ext*(PS) ={m| m ∈Node(*D*) && schema(m)= α }

定义 **9** 给定路径模式树中的两个模式结点 PS_1,PS_2，如果 PS_1 的外延中的某个结点是 PS_2 的外延中的某结点的父亲，则在模式树中，我们定义 PS_1 为 PS_2 的父亲。

在大规模的 XML 文档中，如果我们直接把文档本身当作与查询自动机匹配的句子的话，这种查询过程是非常低效的。因此，我们定义了路径模式树这一索引结构，通过路径模式树，我们可以有效地减少查询的计算时间。在简单路径查询中，路径模式树的作用相当于用来和自动机作匹配的句子，自动机到达终止状态时所对应的模式树结点的外延即为查询结果。例如，对图 2.2 所对应的文档提交查询“/a/d/e/f/h”的查询结果为（&7，&18），即路径模式树中 8 号节点对应的外延。

2.2.3 自动机查询相关算法

许多 XML 查询语言使用路径表达式来表示查询，如 XPath，XQuery 等，一种典型的路径表达式是正则路径表达式，正则路径表达式中包含了连接，选择和闭包操作。祖先－后代操作符“//”不是一个正则操作符，对它的支持将在后面讨论。

因为自动机匹配算法是通过转化为自动机的查询进行计算的，所以查询处理的第一步工作就是把路径表达式转化为查询自动机。一个查询自动机是一个有限状态机，具有 $FSA_Q = (K_A, \Sigma_A, \delta_A, s_A, F_A)$的形式。因为查询自动机代表了一个查询，所以五元组中的 Σ_A 除了包含 XML 文档中全部的节点模式外，还包含了代表空模式的符号 ε 和代表“//”操作符的符号 ξ。当“//”操作符被重写前，在查询自动机中我们使用通配符“ξ”来表示它。我们将形如“E1//E2”这样的表达式写作“E1/ξ/E2”。其中，ξ 是一个可以代表任何模式与模式组合的符号。通过以上扩展，我们得到了 $\Sigma_A \subseteq \Sigma_D \cup \{\varepsilon, \xi\}$，$K_A$ 是状态集合，s_A 是开始状态，F_A 是终止状态集，$s_A \in K_A$，$F_A \subseteq K_A$，$\delta_A: K_A \times \Sigma \rightarrow K_A$ 是转换函数。

对于XML 文档树来说，它的路径实例集合PathInstance(*D*)可以代表整个文档的数据，因此对于XML 文档树的查询可以转化为对于对应的路径实例集合的查询。而根据定义6，每一个路径实例都有其对应的路径模式，因此在一个XML 文档树对应的路径实例集合PathInstance(*D*)与路径模式集合PathSchema(*D*)之间存在着对应关系。故对于XML 数据的查询操作可以由在路径模式集合上的查询操作替代。自动机匹配路径查询技术（Automata Match）就是基于这样的理论的XML 路径表达式查询处理技术。

在用路径模式树来匹配查询自动机的过程中，路径模式被看作模式的句子用来与自动机匹配。如果一个路径模式 α 可以被一个查询自动机 FSA_Q 所接受，那么所有在 Inst(α)中的节点都是查询的结果。为了找到所有可以被 FSA_Q 接收的路径模式，这里不是简单地对所有的路径模式进行测试，而是提出了一个匹配算法。为了说明这个算法，先提出一个称为“匹配关系”的概念。

定义 **10** 匹配关系 $R \subseteq \Psi_D \times K_A$ 定义如下：

1. (“/”,s_A) ∈ R
2. 如果有α，β∈Ψ_D,有 p,q ∈K_A，(α，p) ∈R，α=parent(β)，d(p，β)=q，那么就有(β，q)∈ R

下面是应用“匹配关系 R”设计的自动机匹配算法概要

算法 2.1：自动机匹配算法（AM）

Algorithm 2.1: automata matching

AUTOMATAMATCH(D,Q)

INPUT:XML 文档 D,路径表达式 Q

OUTPUT:查询结果

1：将路径表达式 Q 转化为查询自动机 FSA_Q

2：计算匹配关系 R

3：通过匹配关系 R 找到所有能被 FSA_Q 接收的路径模式

4：从路径实例树中取出结果节点

下面主要讨论算法 2.1 中的(1)部分，即查询自动机的构建：

在自动机匹配算法中，路径表达式首先被转化成查询自动机。首先，通过一个小型的解析器将表达式转化成一棵表达式树，在表达式树当中，祖先一后代操作符被当作一个普通的操作符ξ计算。如果表达式中还有闭包操作符“+，？”，我们可以通过下面的规则对其进行替换：$E^+ \Leftrightarrow EE^* \Leftrightarrow E^*E$，$E? \Leftrightarrow E|\varepsilon$，替换结束后，在表达式树中只能出现三种操作符，即“/,|,*”。

一个表达式树是一棵二叉树，我们可以通过遍历它来建立自动机。二叉树的每一个叶子节点都被转化为一个自动机片断（它本身就是一个自动机）；在每个分支节点处将其各个子节点对应的自动机组合成一个新自动机。这里用 CONAUT(n)来表示将分支节点 n 转化为自动机的过程。很显然，对路径表达式查询 Q，FSA_Q= CONAUT(Q)。下面具体说明 CONAUT 的计算过程：

$S_A \subseteq S_D \cup \{\varepsilon, \xi\}$代表表达式树叶子节点的集合。函数 NEW(q)表示创建一个新的自动机状态 q，并返回 q。

对于简单的叶子节点 a∈S_D，很显然，CONAUT(a) = ({NEW(q_0), NEW(q_1)} , {a} , {q_0,a,q_1} , q_0 , {q_1})。空模式节点的自动机转化 CONAUT(ε) = (NEW(q_0), Ø, Ø,q,{q})。“//”操作符的转换方法将在后面讲到。接下来我们定义两个自动机 M_1，M_2 之间的空转移函数 $d_{(M1,M2)}$: $F_{M1} \times \{\varepsilon\} \rightarrow \{s_{M2}\}$是从所有的 M_1 的终止状态到 M_2 的开始状态的一系列转换的集合。通过 $d_{(M1,M2)}$的定义，就可以很方便地定义分支节点的转换了。

令 Q_1，Q_2 是两个路径表达式，如果有 M_1=CONAUT(Q1),M_2 =CONAUT(Q_2),

那么 CONAUT(Q1,Q2)=(K_{M1}UK_{M2}, Σ_{M1}UΣ_{M2}, δ_{M1}Uδ_{M2}U$\delta_{(M1,M2)}$, s_{M1}, F_{M2})。为了计算正则操作符“|”，定义了两个单状态的自动机 I_1= CONAUT(ε)，I_2= CONAUT(ε)。注意这里 I_1 和 I_2 并不相同，因为它们不具有相同的状态。CONAUT(Q_1|Q_2) = (K_{M1}UK_{M2}UK_{I1} UK_{I2}, Σ_{M1}UΣ_{M2}, δ_{M1}Uδ_{M2}U$\delta_{(I1,M1)}$U$\delta_{(I1,M2)}$U$\delta_{(M1,I2)}$U$\delta_{(M2,I2)}$, s_{I1}, F_{I2})。类似地，对于闭包运算符“*”，有 CONAUT(Q_1^*) = (K_{M1}UK_{I1}UK_{I2} , Σ_{M1} , δ_{M1}U$\delta_{(M1,M1)}$U$\delta_{(I1,M1)}$U$\delta_{(M1,I2)}$U$\delta_{(I1,I2)}$,s_{I1}, F_{I2})。

通过上述算法 CONAUT(Q)生成的自动机是有限的但不是确定的。通过[19]中的方法对 CONAUT(Q)进行正规化和化简就可以得到一个 DFA。图 2.4 给出了一个通过查询表达式 a/(b|c)//g*构造而成的查询自动机。

图 2.4 查询自动机举例

Figure2.4 sample query automata

在匹配过程中，每个路径模式树的结点都要与自动机的状态相匹配，匹配的状态标记在结点的 status 域。算法从根结点开始，将根结点与自动机的初始状态相匹配，并将根结点加入等待对列。等待队列存放那些需要检查状态和匹配孩子结点的结点。检查结点匹配状态有两部分工作，其一是判断该状态上是否有谓词，如果有，要标记实例树上不满足谓词的子树，其二是判断该状态是否为终止状态，如果是终止状态，则找出所有该模式树结点所对应的实例树结点，如果该实例树结点未被标记，则说明满足谓词要求，是查询需要的结果。在标记孩子结点的状态时，要判断自动机在父结点的状态读取孩子结点的标签后转移到哪个状态，如果该状态存在，则用此状态标记孩子结点，并把孩子结点加入等待队列。

算法 2.2 给出了匹配关系的计算。

算法 2.2：匹配关系的计算

Algorithm 2.2: Evaluation of matching relation

输入：

RPE 自动机 fsa

路径模式树 pst

实例树 pit

输出：

结果集合 rs

中间结构：

匹配等待队列 waitlist

算法描述：

```
pst.root.status <- fsa.startstatus // 用模式树根匹配自动机的开始状态
waitlist <- {pst.root}; // 模式树根进等待队列
while waitlist is not empty
  curnode <- shift (waitlist); // 从等待队列中取出并删除一个结点
  if has pred on curnode then
    mark the exclusive sub-instancetree // 如果自动机状态上有谓词则验证谓词并标记子树
  endif
  if curnode.status is in finalstatus then // 如果当前结点的匹配状态是自动机的终止状态
    for each inst in curnode.instset do // 找出所有模式结点对应的实例结点
      if instance has not be marked then // 验证是否为查询结果
        rs <- rs U {instance} // 放入结果集合
      endif
    done
  endif
  for each child of currentnode cc do
    if fsa has a transition from state labeled curnode.status to a status ts activated by cc then
      cc.status <- ts; // 匹配孩子结点与相应的自动机状态
      waitlist <- waitlist U {cc}; // 将孩子结点加入等待队列
    endif
  done
endwhile
```

2.3 自动机匹配引擎系统结构

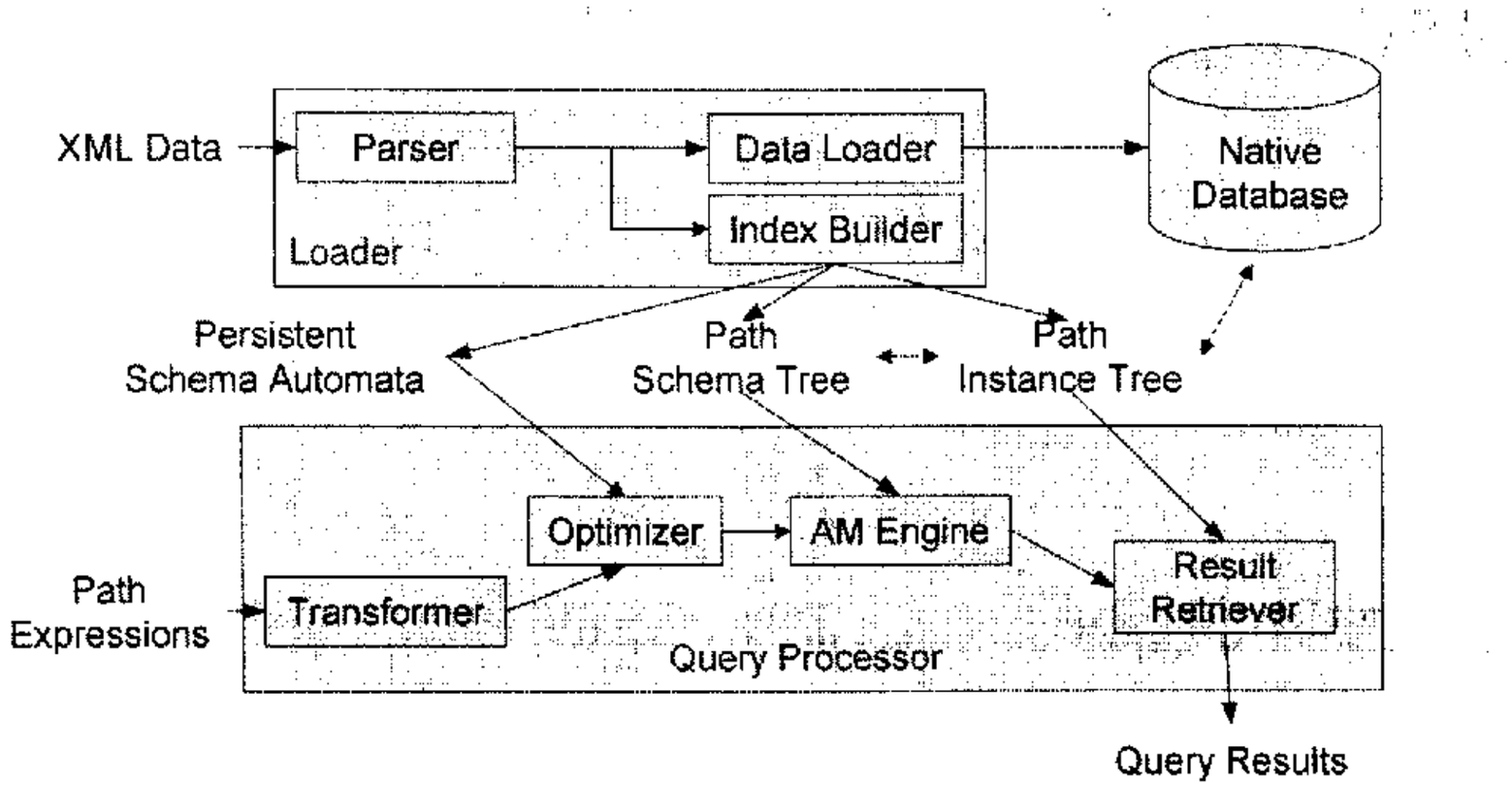

图 2.5 系统结构图（AM-Engine）

Figure 2.5 System Architechture (AM-Engine)

AM-Engine 是一个 XML 路径表达式查询自动机匹配查询处理器，它的目标是使用自动机匹配技术支持基于路径表达式查询处理。AM-Engine 的基本体系结构如上图 2.5 所示。

AM-Engine 系统进行查询处理的过程主要分为 3 步：

1. 首先将 XML 文档通过解析器进行解析，将这一步的输入作为 Data Loader 和 index Builder 两个模块的输入。分别将解析后的 XML 文档转化为路径实例树和路径模式树存入数据库中。
2. 路径表达式查询需要进行必要的处理操作，将路径表达式转换查询自动机结构。
3. 使用存储在索引数据中的 XML 数据对应的路径模式信息匹配查询自动机，如果路径模式被查询自动机接受，则根据路径模式与路径实例间的对应关系得到符合条件的路径实例。
4. 最后，对路径实例集合进行取最后节点处理，得到路径表达式查询最后查询结果。

第三章 XML 路径查询中祖先—后代关系查询的解决

在 XPath 规范中，祖先—后代操作符“//”是另一个频繁出现在查询表达式中的操作符，与上面讨论的“*”不同，“//”是一个非正则运算符。这个操作符的意义是找到当前节点的所有后代节点，我们也可以把“//”看作是路径表达式中的通配符，也就是说，“//”出现的位置可以对应任何路径表达式。针对“//”的处理，目前主要有以下几种方法：第一种是根据文档中的统计信息找到“//”对应的所有可能的路径，据此，对查询路径表达式进行重写。这种方法的缺点是当 XML 文档的模式信息非常复杂时，重写表达式的工作量会变得非常庞大甚至根本不能完成。另一类解决“//”的方法是以包含连接为代表的一系列算法。包含连接算法的基本思想是：祖先和后代节点被分别放入祖先列表和后代列表中，然后使用特定的算法对这两个表进行连接。包含连接的方法在某些索引的支持下通常是比较有效的，但是其效率依然依赖于参与连接的两个集合的大小。因此，当参与连接的集合非常大时（比如查询 a // *），包含连接的效率就不能满足算法的需要了。因为“//”不是正则运算符，所以自动机匹配的方法不能像支持“*，+”那样对其直接支持。本文提出了一个称为模式自动机（Schema Automata）的数据结构来支持“//”的计算。模式自动机可以通过 XML 文档本身或文档的模式信息来构建。它既可以在查询处理时动态生成，也可以作为索引的一部分存储在数据库中。

3.1 模式自动机

正如第 2 章中描述的那样，除了“//”以外，表达式中的所有节点都可以转化为一个自动机。因为“//” 不是一个正则操作符，这里采用将其重写成模式自动机(Schema Automata)的方法来解决包含“//”的查询。

对于一个 XML 文档 D，它的模式自动机 SA_D 是一个可以接收 D 中所有路径模式子序列的自动机。很显然，当 SA 重写完成以后，可以将 SA 直接插入到查询表达式转化成的查询自动机中“//”对应的位置，也就是说 $\text{CONAUT}(//) = SA_D$。定义 11 以推导的方式给出了模式自动机的定义。

定义 11 对任意的一个 XML 文档 D，L(FSA)表示 FSA 所接收的语言，一个自动机 $SA_D = (K_S, \Sigma s, \delta s, s_S, F_S)$是模式自动机当且仅当：

1. $\Sigma s \subseteq \Sigma_D$；
2. $\varepsilon \in L(SA_D)$，也就是说，$s_S \in F_S$；

3. $\Psi_D \subseteq L(SA_D)$;
4. 对任意一个 $a \in \Psi_D$ 并且 $|a|>1$,有 $left(a,|a|-1) \in L(SA_D)$ 且 $right(a,|a|-1) \in L(SA_D)$。

模式自动机可以通过 XML 文档本身或 XML DTD，XML Schema 的信息来构建。算法 3.1 给出了如何根据 XML 文档来构建一个模式自动机。结构自动机是一个可以接收文档中所有路径模式的自动机，它的转换图形式与文档的 DTD 图相同。

下面的图 3.1 和图 3.2 分别给出了一个 DTD 图和其对应的模式自动机

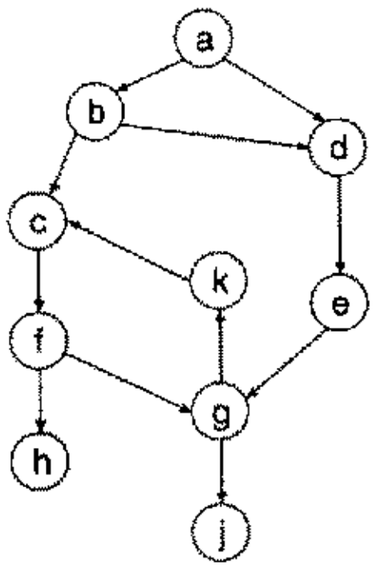

图 3.1 DTD 图

Figure3.1:DTD graph

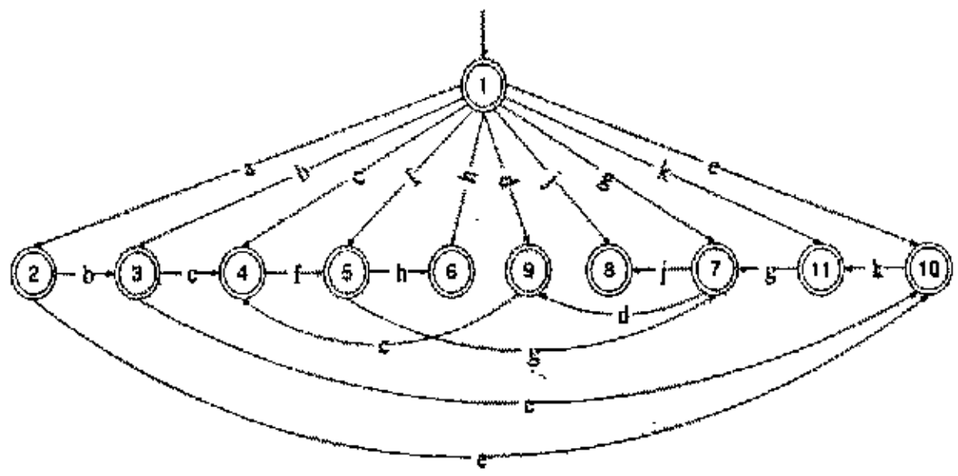

图 3.2 模式自动机

Figure3.2 Schema automata

算法 3.1：模式自动机的构建

Algorithm 3.1: Construction of schema automata

CONSTRUCTSCHEMAAUTOMATA(D)

输入：XML 文档 D

输出：模式自动机 SA_D

1. 通过 XML 文档构建一个结构自动机 $TA_D=(K_T, S_T, d_T, s_T, F_T)$
2. NEW(s_S) , NEW(f_S)
3. $ds \leftarrow d_T$
4. **for all** q $\in K_T$ **do**
5. $ds \leftarrow ds \cup \{(s_S, \varepsilon, q),(q, \varepsilon, f_S)\}$;
6. **end for**
7. $SA_D = (K_T \cup \{s_S, f_S\}, S_T \cup \{\varepsilon\}, ds, s_S, \{f_S\})$;
8. 对 SA_D 进行正规化和最小化
9. **return** SA_D.

由图 3.2 可知，即便已经经过最小化，一个比较复杂的 XML 文档对应的模式自动机还是比较复杂的。

3.2 模式自动机的优化

模式自动机可以用来重写祖先一后代操作符从而正确地计算查询表达式。然而，由图 3.2 可知，XML 文档对应的模式自动机还是比较复杂的，如果要进一步提高查询性能，就需要对其进一步化简。由算法 3.1 可知，模式自动机的构建只与 XML 文档相关，而与查询表达式无关。通过对查询表达式的结构进行分析发现：模式自动机可以根据查询表达式做进一步化简。这里提出了一个通过 preceding 和 following 集合来化简的方法(RWS)来提高性能。

给定一个路径表达式查询 Q，对 Q 中的每个符号 ω，ω 的前驱集合(preceding set) $PS_Q(\omega) \subseteq S_D$ 代表了在所有的路径模式中恰好在 ω 前的模式集合；类似地，ω 的后继集合(following set)，$FS_Q(\omega) \subseteq S_D$ 定义为所有恰好出现在 ω 后的模式集合。举例来看：在路径表达式 PE: a/(b|d) / ξ /g，ξ 的前驱集合 $PS_Q(\xi)$为{b,d}，ξ 的后继集合 $FS_Q(\xi)$为{g}。

因此，不难看出：在某个路径表达式中，ξ 并不代表任意的“通配符”。它必然出现在其 $PS_Q(\xi)$中的某个模式后，同时也一定出现在其 $FS_Q(\xi)$中的某个模式前。通过这条规律可知，在将 SA_D“嵌入”到某个路径表达式查询 Q 对应的自动机之前，我们可以先计算 SA_D 的前驱集合和后继集合，然后将 SA_D 中不在前驱集合和后继集合中的部分剪掉。这一策略对性能的提高效果显著，这一点，可以通过后面的例子和性能测试中看出。图 3.3 给出了一个图 3.2 中的 SA_D 的通过 RWS 算法化简后的版本。RWS 算法在下面给出，该算法遍历 SA_D 的状态两次，第一次得到所有通过前驱集合中的模式可能到达的状态，第二次找到所有可以到达后继集合中模式的状态。

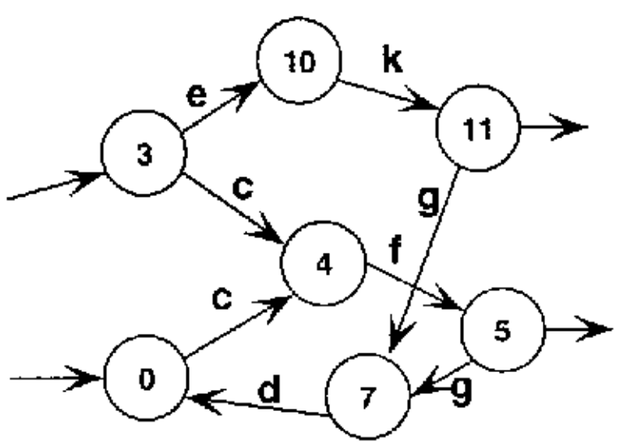

图 3.3 通过 RWA 化简后的 SA_D

Figure3.3 The SA_D that after simplification by RWA

算法 3.2：模式自动机的化简

Algorithm 3.2: Reduce schema automata

REDUCESCHEMAAUTOMATA(SA_D,ps,fs)

输入：模式自动机 SA_D,前驱集合 ps,后继集合 fs

输出：化简版本的模式自动机 RSA_D

1. **if** $\xi \in ps$ **then**
2. waitset $\leftarrow K_S$;
3. **else**
4. *waitset* $\leftarrow$ {};
5. **for all** schema a$\in ps$ **do**
6. waitset $\leftarrow$ *waitset* $\cup$ $\{d_S(s_S, a)\}$;
7. **end for**
8. **end if**
9. $S_S \leftarrow$ *waitset*, *visitedset* $\leftarrow$ {},*reducedset* $\leftarrow$ {};
10. **while** *watiset* not empty **do**
11. p = pop(*waitset*),visitedset $\leftarrow$ *visitedset* $\cup \{p\}$;
12. **for all** b that $d_S(p,b)=q$ exists **do**
13. **if** b$\in f_S$ then
14. *reducedset* $\leftarrow$ *reducedset* $\cup \{p\}$;
15. **end if**
16. **if** $q \notin$ *visitedset* then
17. *waitset* $\leftarrow$ *waitset* $\cup$ {q};
18. **end if**
19. **end for**
20. **end while**
21. $F_R \leftarrow$ *reducedset*,*visitedset* $\leftarrow$ *visitedset* $\cup$ {r};
22. **while** *reducedset* not empty **do**
23. r= pop(*reducedset*), *visitedset* $\leftarrow$ *visitedset* $\cup \{r\}$;
24. **for all** c that $d_S(\omega,c)=r$ exists **do**
25. **if** $\omega \notin$ *visitedset* **then**
26. *reducedset* $\leftarrow$ *reducedset* $\cup \{\omega\}$;
27. **end if**
28. **end for**
29. **end while**
30. $K_R \leftarrow$ *visitedset*, $I \leftarrow$ CONAUT(ε);
31. $RSA_D \leftarrow (K_R \cup K_I, S_S, d_S \cap (K_R \times S_S \times K_R) \cup (\{(s_I, \varepsilon)\} \times S_S), s_I, F_R)$;
32. **return** RSA_D

模式自动机可以在查询处理时动态地被创建出来，另一种可以改善性能的方法是将创建好的模式自动机存储在磁盘上。我们将存储在磁盘上的自动机称为持久化的模式自动机(persistent schema automata,*PSA*)。在 PSA 上执行最频繁的操作是对转换函数 δ 的查找，因此我们为转换函数设计了类似于 B+树的索引结构来加快访问速度。在为 PSA 建索引的过程中，每个转换函数在 B+树的节点中有其对应的入口，节点在满了以后采用 B+树中的分裂算法进行分裂。

在实验中，我们采用的底层数据库是 Native XML 数据库，然而，自动机匹配算法也可以在其他的数据库平台，比如关系数据库中很方便地实现。在关系数据库系统中，PSA 可以用一个转换函数表来表示，表中有 3 列分别是(From,Schema,To)，这样，转换函数可以通过 *SELECT* 语句很容易的得到。

3.3 性能评测

为了评价本章提出的自动机匹配算法(AM)，我们在一个 Native XML 数据库系统 XBase 上实现了 AM 查询系统。而且，为了更好地说明 AM 算法性能上的优势，我们也在 XBase 上实现了一个典型的包含连接算法(stack tree join,STJ)[23]。同样，我们对上一小节提出的两种优化策略 *RWS* 和 *PSA* 的效果也做了测试，这样我们一共评测了四种自动机匹配算法(及其优化算法)的性能，他们是：*AM*, *AM+RWS*, *AM+PSA* , *AM+RWS+PSA*。下面具体说明这四种方法和 STJ 方法比较的实验结果。

以下提到的测试均在一台主频为 933MHz，内存为 128M 的 PC 上进行，我们采用一个本地化的 XML 数据管理系统（XBase）来存储数据，它通过 ODMG 绑定的 DOM 接口将 XML 文档存入一个面向对象的数据库系统（FISH）中，采用 Windows2000 作为操作系统平台。所有的程序都在 VC6 下编写并运行。为了使测试更加全面，我们采用了两个数据集：XMark，是一个用来评测不同长度的路径查询性能的标准数据集，另一个是我们为了更好地测试 AM 在对“//”运算符的计算上对 STJ 方法的性能优势而设计的数据集 CMark。XMark 中共有 20 条查询，为了讨论方便，我们从中选取了 4 条有代表性的不同类型的查询，它们被列在表 3.1 的 Q_1-Q_4 中。我们采用了 100M 的 XMark 文档。我们采用图 3.1 所示的 DTD 来生成 XML 文档，为了测试文档大小对算法的影响，我们用 IBM 的 XML Generator 生成了 5 个大小分别为 20M,40M,60M,80M,100M 的文档。我们为 CMark 特地设计了可以很好地验证算法在包含连接查询上的性能的 3 个查询 Q_5~Q_7。

表 3.1：XMark 和 CMark 查询表达式

Table 3.1: XMark and CMark queries

Q_1：	/site/people/person
Q_2：	/site/regions/Australia/item/description
Q_3：	/site/closed_auctions/closed_aution/annotation/description/text/site/closed_auctins/closed_auction/annotation
Q_4：	/description/parlist/listitem/parlist/listitem/text/emph/keyword
Q_5：	a/b/c//g
Q_6：	a//e//j
Q_7：	a/(b|d)/(g|h)

图 3.4 给出了 STJ 和 AM 两种方法在 XMark 数据集上的比较，因为这几个查询中没有“//”操作符，所以我们并没有在 XMark 数据集上研究模式自动机的性能。通过图 3.4 可以看出，在查询短路径时，AM 算法并不比 STJ 算法有明显的性能优势；这是因为路径比较短，一般情况下查询结果数目也比较少，STJ 需要进行的连接运算的工作量也很少，而自动机匹配在这种情况下，就成了相对比较耗时的操作，所以性能上比 STJ 算法没有明显优势，在结果数很少的时候，甚至会出现性能不如 STJ 算法的极端情况，如 Q2。然而，在长路径查询中，AM 算法的性能绝对优于 STJ。例如 Q_4，其 STJ 的响应时间是 AM 响应时间的 10 倍，这说明，对于这种很长的路径，自动机匹配算法彻底的摆脱了 STJ 算法对 XML 实际数据的大量的复杂操作，而是集中处理与模式有关的问题，把解决问题的复杂度限制在模式层面，所以可以获得很好的查询性能。

AM 算法的时间消耗主要分为两个阶段，第一是匹配阶段，第二是结果的提取阶段。因为第一个阶段除了处理谓词以外不需要真正访问数据库中的实例节点，这就大大的减少了查询时间，所以在处理比较长的路径的时候，与其他方法（如 STJ）相比，节省的时间就比较多，但处理较短的路径时，其节省的时间接近甚至于少于因处理匹配操作而消耗的时间，所以查询效果并不理想。通过这些结果可以看出，总的来看，AM 是一种非常有效的路径查询方法，特别是对长路径查询。

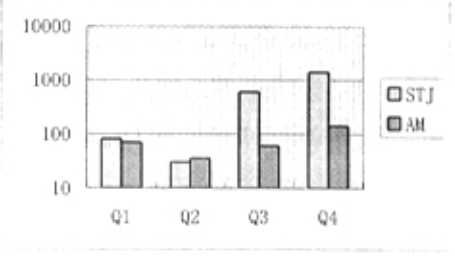

图 3.4 AM 与 STJ 在 XMark 上的性能

Figure3.4 Performance of XMark(100M)

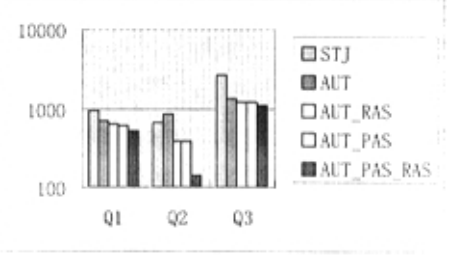

图 3.5 AM 与 STJ 在 CMark 上的性能

Figure 3.5 Performance of CMark

通过图 3.5 给出了 AM 及其优化算法与 STJ 在 CMark 数据集上的性能比较，从图中可以很清楚地看出，全优化的自动机匹配算法(AM+RWS+PSA)的性能总是远远超过 STJ。从图 3.5 中除了可以看到 AM 算法的性能优势外，我们注意到 RWS 和 PSA 优化对 *AM* 本身性能的提高所起的作用也是很显著的。通常情况下，这两种优化本身性能接近。不使用 RWS 和 PSA 优化的 AM 算法在 Q_5 和 Q_7 的性能上均优于 STJ，但在 Q_6 上稍逊于 STJ，这是因为在 Q_6 中有两个“//”操作符，在我们的实现中，在查询处理时两个模式自动机需要被构造出来，整个查询自动机变得非常复杂，因而影响了查询性能。这个问题可以通过 RWS 和 PSA 两种优化策略解决。同样，恰恰是这个查询体现了我们设计的 RWS 和 PSA 优化所带来的效率的巨大提升，从图 3.5 可以看出，AM+RWS+PSA 的响应时间远远小于其他几种方法，原因是这个查询里有两个“//”操作符，两种优化策略所带来的效果相当于得到了双倍的体现。

3.4 本章小结

本章中，我们提出了一个新的数据结构－模式自动机来计算路径表达式中的“//”操作符，提出了两种对自动机匹配算法进行优化的方法：RWS 和 PSA。AM 是一种计算路径查询表达式的很有效的方法，他在模式空间中运行，除了在取最终的查询结果之外不需要访问实例节点。通过实验，我们发现：AM 算法对长路径查询尤其有效。模式自动机的提出，扩展了 AM 算法在计算包含连接方面的能力，RWS 和 PSA 两种优化方法进一步加快了这一过程。通过实验结果可以得出这样的结论：AM 算法的响应时间随着文档规模的增大只有少量的增加，因此，相对于其他方法，AM 算法在处理大数据量查询时优势更加明显。

第四章　XML Twig 路径查询

4.1 Twig 路径查询的技术概述

近来，在简单路径查询的问题得到较好解决的基础上，人们将注意力转移到复杂路径查询即 Twig 查询中来。在 XPath 标准[4]中，谓词操作是定位路径的一部分，谓词操作的实质是对查询的中间结果进行过滤的过程，包含谓词的 XPath 查询可以表示成一棵带有“分叉”的查询树形式。相对于不带有“分叉”的简单路径查询，这种树形的查询通常被称作“Twig”查询。在 XML 数据库中找到所有符合某种特定 Twig 模式的结点集合是 XML 查询处理领域中的核心操作之一[6]。

处理 Twig 查询最简单直接的方法是：先将 Twig 查询分解为若干基本的二元关系查询，然后将每一个二元关系查询的结果进行结构连接得到需要的结果。虽然一些对结构连接优化的算法不断被提出，但使用结构连接解决 Twig 查询有以下几个问题：(1)由于需要对多个子查询的结果进行连接，中间结果的存储代价巨大；(2)在计算结构连接过程中，需要对每个实例结点进行遍历和比较；(3)采用结构连接算法实现正则操作符（如闭包运算）需要根据文档模式信息(DTD 或 XML Schema)和文档的统计信息进行重写。

本文采用自动机技术处理 Twig 查询，自动机是根据查询要求动态生成的，而不是以索引的形式作为用户的配置文件存在数据库中。在查询过程中，通过用路径模式树索引来匹配查询自动机，可以充分利用文档中的模式信息，从而大大减少对文档实例结点的访问次数，提高查询的性能。与结构连接算法每次连接都要存储中间结果相比，用自动机来解决 Twig 查询需要存储中间结果的次数只与该查询中“Twig”的数目有关，而与路径表达式中路径的长短无关。此外，由于自动机本身与正则表达式的等价性，使用自动机来实现正则操作符可以省去不必要的重写操作，对于那些非正则运算符“//”，我们可以用重写的方法解决。

4.1.1 Twig 路径表达式

在 XPath 标准中，查询语句“[”和“]”中间的部分为谓词。例如：在图 2.2 中的 XML 文档中计算查询 Q1：/a/d/e[g/i]/f/h 的结果是&7。表 4.1 给出 Twig 绝对路径表达式(TAPE)的 BNF 范式，这里我们只考虑结构谓词，并没有包含值谓词的情况。

如上所述：在只考虑‘/’和‘//’轴的情况下，一个 Twig 路径表达式在去掉‘[,]’后可以看作一个单路径查询，我们把去掉‘[,]’后的单路径查询称为主路径查询。而‘[,]’中的谓词表达式可以看作是对主路径查询的某些结点上加入路径条件限制。因此，计算一个 Twig 表达式的值就是在主路径求解的结果中找到满足路径条件限制的结点。

4.1.2 Twig 查询自动机

一个 DFA 的定义通常为一个五元组（K, S, d, q_0, F），其中 K 为状态集合，S 为字母表，d 为状态转移函数，q_0 为开始状态，F 是终止状态集[19]。对于一般的 RPE 查询自动机，字母表为文档中元素名称的集合。为了解决 Twig 查询，我们在字母表中加入了一些特殊的字母，使自动机在读到它们时执行特殊的动作，新的字母表为

S ’=S U {‘//’, ‘CONSTRUCT’, ‘FILTERUP’, ‘FILTERDOWN’, ‘PUSH’, ‘POP’, ‘RESTORE’}。

表 4.1Twig 表达式的 BNF 语法

Table 4.1: BNF of Twig expression

```
TAPE::=’ /’ TPE
TPE::=TPE Bi-Op TPE
   | TPE Un-Op
   | step
Bi-Op::=  ‘/’
   |  ‘//’
Un-Op::=  ‘*’
   |  ‘+’
step::=name
   |name  ‘[’ predicate ‘]’
      predicate::= TPE
```

4.2 自动机匹配 Twig 查询的相关算法

4.2.1 在模式空间内计算祖先－后代关系

为了支持 XPath 中的后代关系轴，现在比较通用的方法是对整棵文档树的每个结点按（StartPos, EndPos）进行编码并使用 STJ 算法来得到查询结果，在此编码的基础上，出现了一系列相关的优化算法。前面我们提出了一种将文档的模式重写成模式自动机的方法，当查询中出现“//”操作时就将此模式自动机插入到查询自动机中与路径模式树进行匹配。这里，我们提出一种新的基于模式树匹配的祖先－后代关系计算的方法：在计算“Pre/a//b”形式的查询时（Pre 是从根到元素 a 的一条路径），我们将一个特殊字母‘//’引入字母表，上面的查询转化成的自动机如图 4.1 所示：每当读到“//”这个特殊字符时，自动机查询引擎被通知下面要进行一个

图 4.1 ：查询片断 Pre/a//b 转化成的查询自动机

Figure4.1:Query automata transformed from pre/a//b

祖先一后代关系计算，我们假设读 a 之后，状态 n+1 与模式树集合 S_1 相匹配，算法 4.1 得到与读 b 之后的状态 n+3 匹配的模式树集合 S_2。取 S_2 中每个结点对应的外延的并集就是查询结果。下面给出通过遍历模式树计算“//”操作符的算法。

算法 4.1：利用模式树计算后代

Algorithm 4.1: Evaluate descendants in schema tree

输入：匹配自动机状态 n+1 的路径模式树结点集 S_1
后代的元素名称 name
路径模式树 T
输出：匹配自动机状态 n+3 的路径模式树结点集 S_2
算法描述：

```
FIND-DESCENDANT(S1, name, T)
1   for each schemanode  ni ∈ S1  do
2       TRAVERSE-SCHEMA-TREE(ni,name);//对 S1 集合中的每个元素，遍历以其为根的子树
3   TRAVERSE-SCHEMA-TREE(root,name)
4   if(root=T.null()) then//如果当前结点为空，返回
5       return;
6   child=GET-CHILD(root,name);//找到当前结点的元素名为 name 的子结点
7   if (child≠T.null())
8       S2 ←S2 U {child};
9   for each (a in S) do //对字母表中的每个字母
10      child2=GET-CHILD(root,a);//找到当前结点的元素名为 a 的子结点
11      if (child2≠T.null()) then
12          TRAVERSE-SCHEMa-TREE(child2,name);//遍历每一个不为空的子树查找 name
```

4.2.2 Twig 查询自动机的构建

为了支持 Twig 查询，我们需要在自动机的原有的字母表中加入一些特定字母，使之在被读到时执行其定义的“动作”，下表列出了主要的动作

名称和他们各自的功能。

表 4.2：新增字母及其相应动作

Table4.2 New alphabet and the corresponding operation

CONSTRUCT	将与当前状态匹配的模式树结点集 SchemaSet 对应的实例结点以四元组的形式放入当前实例结点表(InstanceSet)中，四元组(StartPos,EndPos,Level,flag)中前三项通过将文档结点通过编码得到，flag 为表示该结点是否满足条件限制的标志位，flag 初始值置 0
PUSH	将与当前状态匹配的模式树结点集 SchemaSet 和当前的 InstanceSet 分别压入栈 Sk_schema 和栈 Sk_instance 中
POP	将 Sk_schema 和 Sk_instance 弹栈
RESTORE	把 Sk_schema 的栈顶作为与当前的状态匹配的模式结点集
FILTERUP	用当前的 InstanceSet 来过滤 Sk_instance 的栈顶
FILTERDOWN	用 Sk_instance 的栈顶来过滤当前的 InstanceSet

我们引入了两个栈 Sk_schema 和 Sk_instance 的目的是：当自动机匹配进行到关键结点时(为了叙述方便，我们将有谓词出现的结点称为关键结点，如 Q1 中的 e 结点)，以 Q1：/a/d/e[g/i]/f/h 为例，有两条路径需要匹配，即 g/i 和 f/h,如果先匹配谓词，也就是 g/i，在匹配了 i 后，为了使自动机回到关键结点并继续匹配另一条路径（f/h），需要在自动机匹配了关键结点后且匹配 g 之前，把关键结点 e 对应的模式树结点集 SchemaSet 存储在 Sk_schema 中，对应的实例树结点集 InstanceSet 存储在 Sk_instance 中。这样，在匹配了 g/i 之后，自动机就可以回溯到关键结点，依据前面存储的模式结点集与另一条分支路径 f/h 继续匹配。在匹配了 g/i 路径之后，根据 i 对应的实例结点过滤掉 Sk_instance 中不满足条件的 c 结点。在匹配了另一条路径 f/h 之后，根据 Sk_instance 中的 e 结点过滤掉不满足条件的 h 结点。

上表中的 FILTERUP，FILTERDOWN 两个操作的实质是在祖先和后代两个集合中找到满足祖先—后代或父子关系的元素对，可以采用包含连接或其改进算法来实现。FILTERUP 是用后代来过滤祖先：每当找到一对满足条件的元素后，如果祖先元组中的标志位为 0，则将其置 1。FILTERDOWN 是用祖先来过滤后代，每当找到一对满足条件的元素后，如果祖先元组中的标志位为 1，则将后代元组的标志位置 1。下面给出将任意一个 Twig 路径表达式转化为 Twig 查询自动机的算法：

算法 4.2 ：Twig 自动机构建算法

Algorithm4.2: Contruction of twig automata

输入：Twig 路径表达式解析成的树 *pet*

输出：由 *pet* 转化成的查询自动机 *ta*

算法描述：

```
TWIG-TO-AUTO (pet)
1   if CHILDREN-COUNT (pet) = 0 then
2       ta ←CONVERT-TO-AUTO (pet); //将 pet 结点转化为自动机
3       return;
4   else if CHILDREN-COUNT (pet) = 1 then
5       ta ← CONNECT (CONVERT-TO-AUTO (pet), TWIG-TO-AUTO (FIRST-CHILD
    (pet))) ;
6       return; // CONNECT 将参数 1 和参数 2 用空转移连接
7   else
8       ta ← CONVERT-TO-AUTO (pet);
9       ta[K] ← ta[K] ∪ {NEW (n1), NEW (n2)}; //在 ta 的字母表中加入 n1,n2
10      for each f ∈ta[F] do
11          ta[d] ← ta[d] ∪ {(f, CONSTRUCT, n1)} //向 ta 中增加转换函数
12      ta[d] ←ta[d] ∪ {(n1, PUSH, n2)}
13      fc ← First-Child (pet)
14      fa ← TWIG-TO-AUTO (fc)
15      pc ← NEXT-SIBLING (fc)
16      while pc≠NULL do
17          for each pp ∈ ROOT-TO-LEAF-PATH(pc) do
18          // pp 是一条从 pc 到以 pc 为根的子树的叶子的路径
19              ta ← CONNECT (ta, RPE-To-Auto (pp))
20              //采用[19]中的方法将 pp 转化为自动机
21              ta[K] ← ta[K] ∪ {NEW (n1), NEW (n2),NEW(n3)}
22              for each f ∈ta[F] do
23                  ta[d] ← ta[d] ∪ {(f, CONSTRUCT, n1)}
24              ta[d] ← ta[d] ∪ {( n1, FILTERUP, n2),(n2, RESTORE, n3)}
25              pc ← NEXT-SIBLING (pc)
26      ta ← CONNECT (ta, fc)
27      ta[K] ← ta[K] ∪ {NEW (n1), NEW (n2),NEW(n3)}
28      for each f ∈ta[F] do
29          ta[d] ← ta[d] ∪ {(f, CONSTRUCT, n1)}
30      ta[d] ← ta[d] ∪ {( n1, FILTERDOWN, n2),(n2,POP, n3)}
```

根据表 4.2 中所述的动作和 TWIG-TO-AUTO 算法，下面以 Q1 为例来说明 Twig 查询自动机(图 4.2)的构建过程：

1. 将 Q1 分成关键结点前的主路径(a/d/e)，谓词路径 (g/i)，关键结点后的主路径(f/h)三部分。
2. 根据[19]中的方法将上述三个正则路径表达式分别转化成与之对应的自动机 A1,A2,A3(如图 4.2 所示)，其开始状态和终止状态集分别为 q_{A1}，F_{A1}，q_{A2}，F_{A2}，q_{A3}，F_{A3}，开始状态用一个圆圈表示，终止状态集用同心圆表示，为了突出重点：我们省略了除开始状态和终止状态外的其它状态。

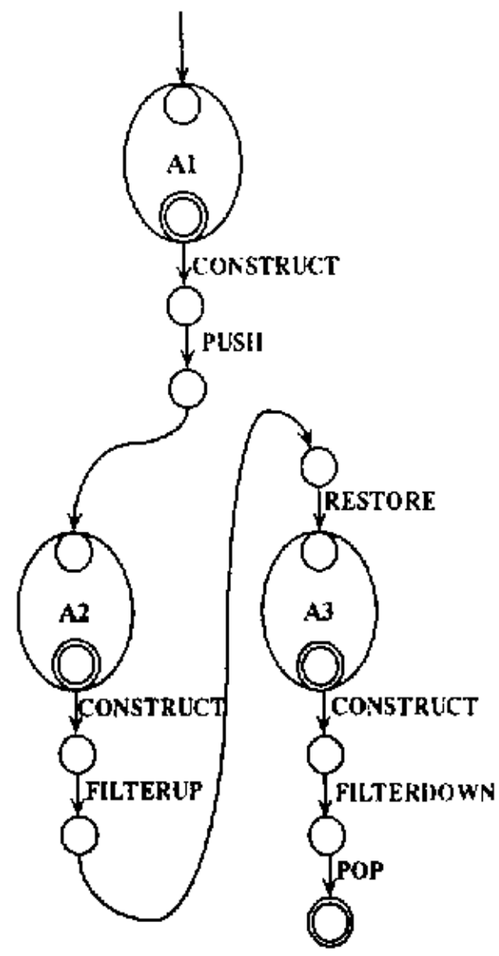

图 4.2：Q1 对应的查询自动机

Figure4.2: Automaton of Q1

3. 在原有基础上增加状态 n_1~ n_8 及增加 $d(F_{A1},CONSTRUCT)=n_1$，$d(n_1,PUSH)=n_2$，$d(F_{A2},CONSTRUCT)=n_3$，$d(n_3,FILTERUP)=n_4$，$d(n_5,RESTORE)=q_{A3}$，$d(F_{A3},CONSTRUCT)=n_6$，$d(n_6,FILTERDOWN)=n_7$，$d(n_7,POP)=n_8$ 共 8 个转换函数。

4. 将第 3 步后形成的 3 个自动机用空转移连接起来，并对 A2,A3 的状态作相应的调整，经过化简，我们就得到了查询 Q1 对应的查询自动机(如图 4.3)。若关键结点后的主路径本身又是一个 Twig 路径，可以递归调用以上过程来得到查询自动机。

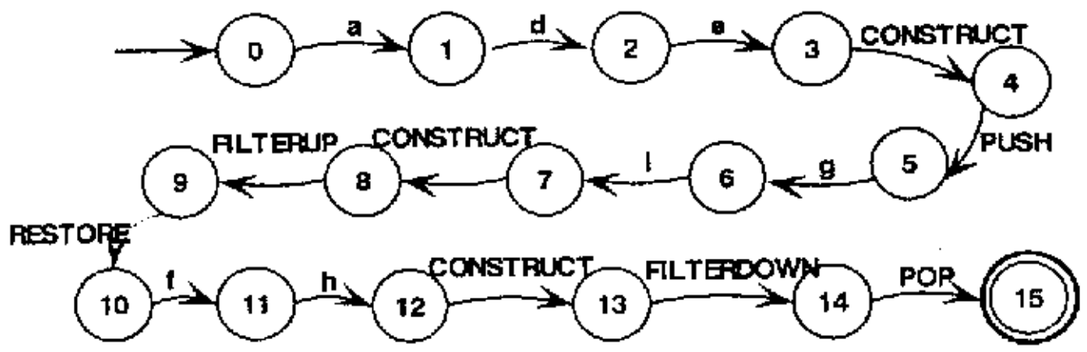

图 4.3：Q1 对应的查询自动机

Figure 4.3 : Query automata corresponding to Q1

4.2.3 Twig 查询自动机的匹配算法

我们的自动机匹配算法的思想是：首先将输入的 Twig 路径表达式按上述方法转化为一个带有特定“操作”的查询自动机。然后，令自动机的

开始状态与模式树的“虚根”结点匹配，依次通过模式树结点的元素名来驱动查询自动机，当遇到表示“操作”的字母时，即去执行相应的操作。当自动机到达终止状态后，当前 InstanceSet 中 flag 为 1 的元组为查询结果。

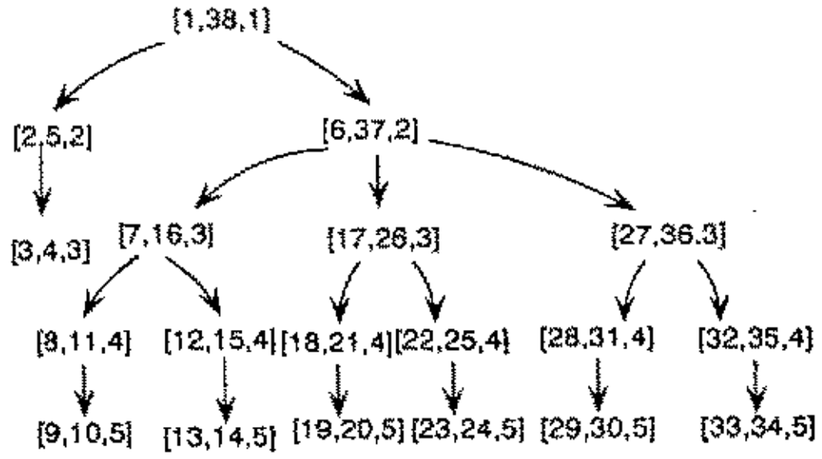

图 4.4：经过三元组编码后的文档树

Figure4.4: DOM tree after numbering

为了对结点进行过滤，我们对文档中的元素采用{StartPos,EndPos,Level}三元组进行编码，如图 4.4 显示了一个通过三元组编码后的 XML 文档。下表模拟了查询 Q1 执行中的一些关键步骤中自动机状态与模式树结点集的匹配过程及相关数据结构的变化：

表 4.3：Twig 自动机匹配过程模拟

Table 4.3:Sample procedure of twig automata match algorithm

字母	状态	模式结点	其它
	0	null	
a	1	{1}	
e	3	{5}	
CONSTRUCT	4	{5}	InstanceSet={<7,16,3,0>　<17,26,3,0>　<27,36,3,0> }
PUSH	5	{5}	将模式结点集{5}和 InstanceSet 压入各自的栈
CONSTRUCT	8	{10}	InstanceSet={ <13,14,5,0>　}
FILTERUP	9	{10}	Sk_instance.top() 中 [7,16,3] 是 [13,14,5] 的祖先，将 [7,16,3]对应的值 f 置 1
RESTORE	10	{5}	将 Sk_schema 的栈顶作为当前的模式结点集
CONSTRUCT	13	{8}	InstanceSet={<9,10,5,0>　<29,30,5,0> }
FILTERDOWN	14	{8}	<9,10,5,0>和 <29,30,5,0>中只有<9,10,5,0>对应的祖先<7,16,3,1>的标志为 1，&7 为查询结果
POP	15	{8}	将 sk_instance 和 sk_schema 弹栈

下面给出自动机与模式树匹配的详细算法：

算法 4.3：Twig 自动机与路径模式树匹配算法

Algorithm 4.3：Twig automaton match

输入：Twig 自动机 *ta*；路径模式树 *pst*；

输出：最终结果对应的实例结点集 *InstanceSet*；

算法描述：

```
TWIG-AUTO-MATCH(ta, pst)
1   matchstate ← (pst.root, ta[q0]);
2   while (true)
3       schemanodeset←matchstate.nodeset., from ← matchstate.status;
4   if d(from , name) ≠ NULL then
5       to ← GET-TRANSITION (from,name) ;
6       if to ∈ta[F] then
7           return InstanceSet;
8       if name ∈{'CONSTRUCT','FILTERUP','FILTERDOWN','PUSH', 'POP' , 'RESTORE'}
    then
9           operation(name); // 调用表 1 中对应的操作
10          matchstate.status← to;
11      if name ='//' then
12          new_schemaset←FIND-DESCENDANT (schemanodeset, name, pst);
13          matchstate ← (new_schemaset, to);
14      else
15          for each schemanode ni ∈ schemanodeset do
16              child ← GET-CHILD (pst, ni, name) ;
17          new_schemaset← new_schemaset ∪ {child};
18          matchstate ← (new_schemaset, to);
```

4.3 AMTwig 算法查询代价分析

我们通过对自动机匹配 Twig 方法(AMTwig)和改进的结构连接(StackTreeJoin)两种方法来解决 Twig 查询进行比较来分析 AMTwig 的性能。因为两者都要用到 STJ 算法，我们称第二种方法为 STJ-Twig 方法。AMTwig 方法的 CPU 时间主要由两部分组成，即查询自动机与模式树进行匹配（AutoMatch）的过程和使用 STJ 算法对中间结果进行过滤的过程。由于模式树的大小远远小于实例树，因此不难看出：对大数据量查询，调用 STJ 算法占用了大部分 CPU 时间。对于一个给定的 Twig 查询，影响其

CPU 时间的主要有以下两个因素：（1）调用 STJ 算法的次数；（2）每次调用中参与连接的两个集合的大小。为了更好的比较因素 1 对 AMTwig 和 STJ-Twig 两种方法的影响，这里我们假设两种方法中每一次调用 STJ 的时间都等于 T_j，这样，查询的 CPU 时间只与调用连接的次数和在模式树中匹配的次数有关。我们设模式树中每个结点匹配占用的平均时间为 T_a，对于一棵查询树，AMTwig 和 STJ-Twig 两种方法的 CPU 时间可以用表 4.4 来估算，其中的 N_{leaf}，N_{key}，N_{total}，N_{schema} 分别为查询树中的叶子结点数，主路径上的关键结点数，结点总数和路径模式树中的结点总数。以查询 Q1:/a/d/e[g/i]/f/h 为例，查询树中结点总数为 7，所以 STJ-Twig 调用 STJ 的次数为 7-1=6 次，因为主路径为 a/d/e/f/h，其上的关键结点是 e，关键结点数为 1，叶子数为 2，所以 AMTwig 方法只调用两次 STJ。

表 4.4：AMTwig 与 STJ-Twig 两种方法时间比较

Table 4.4 : Comparation of AMTwig and STJ-Twig

T_{AMTwig}	$=(N_{leaf}-1)*T_j + N_{key} * T_j + N_{schema}*T_a$
$T_{STJ\text{-}Twig}$	$=(N_{total}-1)* T_j$
TN_{total}	$= N_{leaf}+ N_{key} +N_{key'} +N_{pre}$

表 4.4 式(3)中的后两项：$N_{key'}$ 和 N_{pre} 分别代表主路径上度为 1 的结点数和谓词路径上度大于等于 1 的结点数。根据表 4.4 可知：给定一个查询 Q，AMTwig 方法比 STJ-Twig 方法少做 $N_{key'}+N_{pre}$ 次连接。在最坏的情况下：$N_{key'}+N_{pre}=0$(形如 a[b]/c,a[b]/c[d]/e 这样的查询)，因为 AMTwig 方法的 CPU 时间包括 $N_{schema}*T_a$，所以在此最坏的情况下(并且不考虑下面讨论的因素 2)，AMTwig 方法稍慢于 STJ-Twig 方法。我们用 $(N_{key'}+N_{pre})/N_{total}$ 来表示 AMTwig 方法对 STJ-Twig 方法在连接次数上的优势。该比值越大，两种方法调用连接次数之比越小。

因为 XML 文档中元素分布的不均匀性，影响 Twig 查询 CPU 时间的除了调用 STJ 的次数外，还有每次调用 STJ 时参与连接的两个集合的大小（因素 2）。在构建参与连接的两个集合的过程中，使用路径模式树索引可以避免大量一定不会产生匹配结果的元素参与 STJ 操作。在每一步连接操作时：与 STJ 方法将当前查询结果与整个文档中所有名为 X 的元素组成的集合进行连接相比，AMTwig 算法取出的只是模式树中某个名为 X 的结点对应的外延。因此，一般来说 AMTwig 算法中调用 STJ 时访问的实例结点少于 STJ-Twig 方法。在最坏的情况下：查询树中的任意一个结点 q，其在模式树中只对应一个结点；或者虽然 q 对应模式树中的多个结点 $q_1,q_2\ldots\ldots q_n$,但在 q 参与连接时，$q_1,q_2\ldots\ldots q_n$ 对应的外延全部被取出(在进行'//'操作时 q 作为后代结点可能出现这种情况)，在以上两种极端情况下，

模式树并没有起到过滤一部分不满足条件的结果的作用，就因素 2 来讲：AMTwig 算法退化成与 STJ-Twig 一样了。

4.4 性能评测

为了测试自动机实现 Twig 的方法的性能，我们采用在同样平台上实现自动机匹配 Twig 算法（AMTwig）和改进的结构连接（StackTreeJoin）两种方法进行比较。结构连接的方法是通过将 Twig 路径分成查询的片断分别用 STJ 算法进行查询，然后将各个片断的查询结果相连接，得到最终的查询结果。

表 4.5：测试用的查询语句

Table4.5:Query expression

查询	路径表达式
Q1	a/b/c[f/h and g/j]/d
Q2	a/b[//f]//e//j
Q3	a/b/c/d/e/g[k/c]/j
Q4	a/b//c[d/e]/f

我们为了更好地检验本方法在复杂 DTD 数据集上的性能，使用了 IBM XML Generator 来生成文档，通过指定文档元素的最大重复次数和文档的最大深度等特性生成了大小不同的 5 个数据集（TMark）（5M,10M,20M,40M,80M），其中 80M 的文档大约包含 200 万个元素。我们设计了四个查询(Q1～Q4)来测试路径长度，结果数量，文档大小三方面对算法的影响。

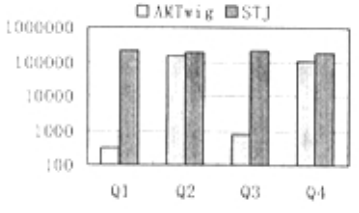

图 4.5：　80M 文档下的性能比较

Figure4.5: Performance of AMTwig and STJ

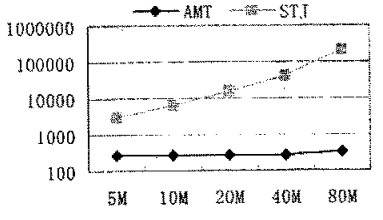

图 4.6：文档大小对长路径查询 Q1 的影响

Figure4.6: Performance of long path query Q1

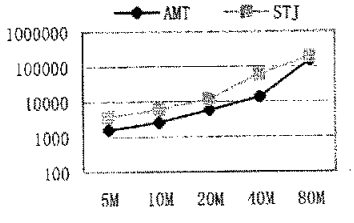

图 4.7：文档大小对短路径查询 Q2 的影响

Figure4.7: Performance of short lenth path query (Q2)

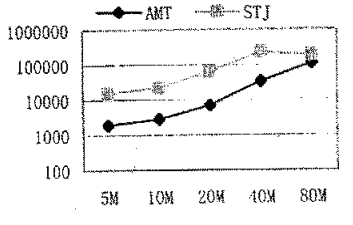

图 4.8：文档大小对中等路径查询 Q4 的影响

Figure4.8: Performance of average lenth query(Q4)

根据测试结果，AMTwig 方法的性能在上述各个方面均优于 STJ，这是因为，STJ 方法在每次连接时，都需要对实例文档中的元素进行操作，而 AMTwig 算法访问实例结点次数远远少于 STJ，以查询 Q3 为例：除了匹配结束时访问实例结点外，中间步骤中只进行了两次连接，访问了三组实例结点即：满足路径模式 a/b/c/d/e/g 的 g，满足模式 a/b/c/d/e/g/k/c 的 c，满足模式 a/b/c/d/e/g/j 的 j。可以看出，在 AMTwig 方法中，由于需要访问实例结点的组数仅与该查询中 Twig 的数量有关，而 STJ 算法访问实例结点的组数与路径长度成正比，因此 AMTwig 算法访问的实例结点数大大小于 STJ 算法。Q1~Q4 中，AMTwig 对 STJ-Twig 的连接次数上的优势：$(N_{key}+N_{pre})/ N_{total}$ 分别为 1/2,2/5,2/3,1/2。因此，如果只考虑连接次数对查询的影响：Q3 最好，Q1,Q4 其次，Q2 最差。由于模式树过滤的效果对 Q1 更为显著，所以 Q1 的效果好于 Q3。从图 4.8 可以看出：由于满足 Q1 这类长路径查询模式的实例结点相对文档中的全部结点较少，因此，文档大小对 Q1 的查询时间影响不大，ATMTwig 算法保持了自动机匹配算法中一贯的长路径优势，与 STJ 方法的加速比在数倍到数十倍之间。对于 Q2 查询（图 4.9），我们主要检验的是短路径查询及通过模式树计算后代轴的算法，AMTwig 的性能是 STJ 的 2 倍左右，由于此查询结果的数量较大(约为整个文档全部元素数量的 10%)，因此查询时间随文档的大小成比例增长。

4.5 本章小结

本文中，我们提出了一个新的自动机匹配算法来支持 Twig 路径查询，该方法在自动机的字母表中增加了一些表示动作的字母，通过将 Twig 路径表达式改写成带有动作的查询自动机并使之与路径模式树进行匹配来计算 Twig 查询。本文还设计了一个基于通过遍历模式树来解决 XPath 中的后代轴的算法，并对上述两个算法在模拟数据集上进行了测试。测试证明，用上述两种方法的结合来解决 Twig 查询问题比传统结构连接方法在性能上有较大提高。

第五章 结论

本文针对 XML 复杂路径表达式查询的特点，提出了一种新颖的基于 XML 数据的路径表达式查询算法——自动机模式匹配算法。这种算法以实例树的方式存储 XML 数据，并根据 XML 数据建立路径模式树作为索引，并在模式树与实例树之间建立对应关系。用户输入的查询路径表达式被转化成自动机，与模式树相匹配，得到查询结果。对于 XPath 中的“//”操作符，我们可以用将“//”重写成模式自动机的方法来解决。对于更复杂的 twig 路径查询，我们可以先将 Twig 查询表达式转化成 Twig 查询自动机，再将 Twig 查询自动机与路径模式树索引进行匹配，得到查询结果。

本文的主要贡献和毕业设计的主要工作如下：

1. 实现了将一个正则路径表达式转化成一个查询自动机并将其正规化，最小化的算法。
2. 实现了用路径模式树匹配查询自动机的算法。
3. 提出并实现了通过用模式自动机来重写“//”实现包含连接查询的算法
4. 实现了模式自动机的构造算法
5. 提出并实现了 Twig 查询自动机的构造算法
6. 提出并实现了用 Twig 查询自动机匹配路径模式树的算法。

通过性能分析，对自动机匹配查询方法的评价和有关结论如下：

1. 对于复杂路径，自动机匹配方法可以得到很好的查询性能。
2. 自动机匹配查询算法的相对性能优势随着路径的增长愈发明显。
3. 自动机匹配查询算法的性能受查询的结果数目的影响。
4. 模式自动机在效率上优于结构连接算法。
5. PSA 和 RWS 是有效的优化策略。
6. Twig 自动机匹配路径模式树比传统的结构连接算法有较大的性能优势。

在此查询方法的基础上，我们还可以考虑在模式空间中采用查询自动机匹配模式树的方法来解决 XPath 中其他轴的计算。

参考文献

1. Altmel M. and Franklin M. Efficient filtering of XML documents for selective dissemination of information. In Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000,53-63.
2. Abiteboul S., Quass D., McHugh J., Widom J., and Wiener J.The Lorel Query Language for Semistructured Data. International Journal on Digital Libraries, 1997, 1(1): 68-88.
3. Abiteboul S., and Vianu V. Querying the Web. In Proceedings of the ICDT, 1997,54-67.
4. Berglund, Baog S., Chamberlin D, et al. XML Path Language (XPath), ver. 2.0. W3C Working Draft 20 December 2001, Tech. Report WD-xpath20-20011220, W3C, 2001. http://www.w3.org/TR/WD-xpath20-20011220.
5. Bruno N., Gravano L., Kouda N., Srivastava D. Navigation vs. Index Based XML Multi Query Processing. In Proceedings of ICDE, Bangalore, India, 2003,139-150.
6. Bruno N., Koudas N., and Srivastava D., Holistic Twig Joins: Optimal XML pattern Matching. In Proceedings of the ACM SIGMOD Conference, Madison, Wisconsin, USA, 2002,310-321.
7. Chan C., Felber P., Garofalakis M., Rastogi R. Efficient Filtering of XML Documents with XPath Expression. In Proceedings of the 28th VLDB Conference, Hong Kong SAR, China, 2002 354-379.
8. Calvanese D., Giacomo G. D., Lenzerini M. and Vardi M. Y. Rewriting of Regular Expressions and Regular Path Queries. In Proceedings of PODS, Philadephia, Pennsylvania, USA, 1999,194-204.
9. Chung C., Min J., and Shim K. APEX: An Adaptive Path Index for XML Data. In Proceedings of the ACM SIGMOD Conference, Madison, Wisconsin, USA, 2002,121-132
10. Cooper B.F., Sample N., Franklin M.J., Hjaltason G.R. and M.Shadmon. A Fast Index for Semistructured Data. In Proceedings of the 26th VLDB Conference, Roma, Italy, 2001,341-350.
11. Chien S. Y., Vagena Z., Zhang D., Tsotras V.J. and Zaniolo C.Efficient Structural Joins on Indexed XML Documents. In Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002,263-274.
12. Diao Y., Franklin M. J. High Performance XML Filtering: An Overview of Yfilter. In Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2003,41-48.
13. Diao Y., Fischer P., Franklin M., and To R. YFilter: Efficient and scalabel filtering of XML documents. In Proceedings of ICDE, San Jose, California, USA, 2002,341-350.
14. Deutsch, Fernandez M., Florescu D., and et al. XML-QL: A Query Language for XML. W3C Note, 1998. Http://www.w3.org/TR/1998/NOTE xml ql 19989819/.

15. Florescu D. and Kossmann D.Storing and Querying XML Data using an RDBMS. IEEE Data Engineering Bulletin, 1999, 22(3): 27-34.
16. Goldman R. and Widom J.. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997,436-445.
17. Guha S., Jagadish H. V., Koudas N., Srivastava D., Yu. T. Approximate XML Joins. In Proceedings of the ACM SIGMOD Conference, San Diego, California, USA, 2003,287-298.
18. Grust T. Accelerating XPath Location Steps, In Proceedings of the ACM SIGMOD Conference, Madison, Wisconsin, USA, 2002,109-120.
19. Hopcroft J.E., Motwani R., Ullman J.D., Introduction to Automata Theory, Languages, and Computation, 2nd ed. Addison-Wesley, 2001.
20. Jagadish H. V., Al-Khalifa S., Chapman A., Lakshmanan L. V. S., Nierman A, Paparizos S., Patel J.M., Srivastava D., Wiwatwattana N., Wu Y., Yu C. TIMBER a Native XML database. The VLDB Journal (2002) 11,2002,274-291.
21. Jiang H., Wang W., Lu H. Holistic Twig Joins on Indexed XML Documents. In Proceedings of the 29th VLDB Conference, Berlin, German, 2003,273-284.
22. Kaushik R., Bohannon P., Naughton J.F., Korth H. F. Covering Indexes for Branching Path Queries. In Proceedings of the ACM SIGMOD Conference, Madison, Wisconsin, USA, 2002,133-144.
23. Al Khalifa S., Jagadish H. V., Koudas N., Patel J. M., Srivastava D. and Wu. Y. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In Proceedings of ICDE, San Jose, California, USA, 2002,141-152.
24. Kaushik R., Krishnamurthy R., Naughton J. F., Ramakrishnan R. On integration of structured indexs and inverted lists. In Proceedings of the ACM SIGMOD Conference, San Diego, California, USA, 2003.
25. Li Q. and Moon. B. Indexing and querying XML Data for regular path expressions. In Proceedings of the 27th VLDB Conference, Roma, Italy, 2001,361-370.
26. Lv J., Wang G., Yu G. Storage, Indexing and Query optimization in A High Performance XML Database System. Proc. of the 2002 PYIWIT Conf. 2002. Japan.
27. Lu H., Wang G., Yu G., Bao Y., Lv J. and Yu Y. XBASE: Making your gigabyte disk queriable. In Proceedings of the ACM SIGMOD Conference, Madison, Wisconsin, USA, 2002,630.
28. Tatarinov, Viglas S., Shanmugasundaram J., Beyer K., Shekita E. Storing and Querying Ordered XML Using a Relational Database System. In Proceedings of the ACM SIGMOD Conference, Madison, Wisconsin, USA, 2002,204-215.
29. Wang W., Jiang H., Lu H., X. Yu J. Containment Join Size Estimation Models and Methods. In

Proceedings of the ACM SIGMOD Conference, San Diego, California, USA, 2003,145-156.

30. Wang W., Jiang H., Lu, H. Yu. J. X. Pbitree Coding and Efficient Processing Containment Join. In Proceedings of ICDE, Bangalore, India, 2003,391-340.

31. Zhang C., Naughton J.F., Dewitt D. J., Luo Q., and Lohman G. M. On supporting containment queries in relational database management systems. In Proceedings of the ACM SIGMOD Conference, Santa Barbara, CA, USA, 2001.

32. 王国仁, 于戈. 分布并行的对象数据库系统[M]. 东北大学出版社. 2001

致　谢

在这里，我首先要感谢的是我的指导老师王国仁教授和我的导师于戈教授，王老师用他丰富的专业知识耐心指导我，不仅教给我了做研究的方法，更教会了我做人的道理。王老师敏锐的思维，开阔的思路，严谨的科研作风，乐观积极的生活态度对我产生了极大的影响。王老师的言传身教使我在研究生的学习期间对编程的看法，对项目的理解以及对学术研究的观念都有了本质的转变和提高。我在实验室学习生活的近两年的时光中，于戈老师一直给予我很大的帮助，从他身上，我不仅学到了知识，更重要的是，我学到了很多正确的看问题的方法，这一点将使我终生受益。于老师虽然平时公务较多，但还能经常抽出时间询问我课题的进展情况，对我进行细致的指导，使我的研究课题得以顺利地进行下去。我还要感谢实验室中的其它老师，单吉弟老师、鲍玉斌老师、王大玲老师、宋宝燕老师、王丹老师、申德荣老师、于亚新老师，这些老师都曾经给过我许多无私的帮助。在技术上使我受益最多的是我的师兄孙冰，有机会与他合作我感到无比的幸运。他有宽阔的知识面，深厚的编程功力，特别是他的乐于助人是我永远难以忘记的。给予我很多指导的还有已经获得博士学位的吕建华师兄，他在学术上取得的杰出成就是我们实验室每一个人学习的榜样。同时，我还要感谢与我与我一起工作的贾福林和王钊，与他们在一起工作，我感到非常愉快。还有实验室中的其它同学：葛键，汤南，王镝和刘欣阳，平时通过同他们的讨论和交往使我受益良多。其实，我在这里更应该感谢的是我的父亲和母亲，在我即将离开学校，走向工作岗位之际，回忆起近 20 年的学习生涯中他们为我付出的心血让我感慨万千，他们给予我的支持和爱护是任何其它东西都不能代替的，是我前进路上最大的动力，感谢他们为我做的一切。我还要感谢所有研究所中的其它同学和我所有的朋友。谢谢大家。

祝愿我们的研究所越来越出色，永远朝气蓬勃。

XML复杂路径表达式查询处理技术研究

作者： 周博
学位授予单位： 东北大学

相似文献(9条)

1.学位论文 付兴宏 基于XML Schema的文档验证技术研究 2004

XML是可扩展标记语言的简称，它为Web上的结构化文档和数据提供了通用的格式。随着Internet的发展尤其是Web技术的广泛应用，越来越多的应用采用了XML技术作为信息表示和数据交换的标准，这使得通过数据库技术对XML数据进行管理变得越来越重要。

在关于XML的数据管理技术中，数据验证是比较重要、且使用比较频繁的组成部分，在维护数据安全和有效性方面扮演着十分重要的角色。XMLSchema作为描述XML的新的W3C推荐标准，以其丰富的数据类型和灵活的结构描述等优点，被许多系统所使用，越来越多的人开始研究基于XMLSchema的数据验证技术。

针对XMLSchema规范中规定的复杂数据类型的结构描述，本文提出了一种称为模式自动机(SchemaAutomaton)的数据结构，讨论了将XML模式结构转换成模式自动机的方法，设计并实现了一种自动机验证算法来验证实例XML文档的有效性，以解决XML结构正则表达式验证的问题。自动机验证算法可以在模式空间内高效地验证每一个获得的XML数据，因此具有很高的效率。

2.期刊论文 余双.曹冬磊.戴蓓洁.金蓓弘.YU Shuang.CAO Dong-lei.DAI Beio-jie.JIN Bei-hong 高效XML验证技术的实现 -计算机工程与设计2008,29(4)

XML解析器是分析、处理XML文档的基础软件.对XML解析器的高效验证技术进行了研究,实现了支持StAX接口的验证型解析器OnceStAXParser2.0.该解析器采用了多项性能优化措施,包括属性验证的高效实现、元素验证自动机的优化、基于统计的预测算法等.性能测试表明,在进行验证的条件下,OnceStAXParser2.0具有出色的解析性能.

3.期刊论文 胡孔法.刘海东.陈峻.达庆利.HU Kong-fa.LIU Hai-dong.CHEN Ling.DA Qing-li 一种改进的可扩展标记语言查询增量维护算法 -计算机集成制造系统2008,14(11)

为降低可扩展标记数据查询执行器重新构建的代价,提出了一种基于树型结构的可扩展标记语言查询增量维护算法.该算法利用树型结构进行可扩展标记语言数据流查询执行器增量维护,利用自动机来表示状态转换,从而实现了对可扩展标记语言树型结构的动态维护,避免了在没有文档类型定义情况下出现的环形结构的复杂操作,减少了维护时间和状态转换数量.实验表明,基于树型结构的可扩展标记语言查询增量维护算法能够以有限转换路径为代价,有效地完成可扩展标记语言数据流持续查询执行器的动态维护,减少了增量维护时间和状态转换数量.

4.学位论文 孙冰 基于自动机的XML路径查询处理技术研究 2002

该文提出了一种新的XML路径查询技术——自动机匹配查询技术.这一方法使用了两个主要的索引结构:路径模式树和路径实例树.路径实例树存储XML数据和数据之间的结构关系以及一些其它信息.路径模式树以树型结构组织XML数据中所有实例的路径模式,路径模式树结点与实例树结点之间有一对多的对应关系.在查询时,首先将输入的查询表达式转化成查询自动机,然后将自动机与路径模式树相匹配,在匹配成功的模式树结点上找到XML实例的入口点,在实例树上得到查询结果. 围绕着这一方法,该文设计和定义了相应的数据模型;设计并实现了支持自动机匹配方法所需的索引结构和其它相关数据结构;而且设计并实现了新的利用键树来匹配自动机的算法,扩展了自动机的功能;同时提出了自动机匹配算法中路径表达式谓词问题的计算方法和优化方法;该文最后给出了自动机匹配算法与其它两种查询算法在查询性能上的比较并加以简要分析.

5.学位论文 柳娜 统一资源管理系统中查询组件的设计与实现 2006

为了能有效地进行资源共享和交换，提高资源建设的效率和水平。本文提出了统一资源库的概念——即建立一个统一的资源平台，通过资源类型注册机制，对各种不同标准类型的资源提供统一的存取、检索和管理。

由于统一资源管理平台数据量大、实时操作频繁、数据类型多样化，因此本文采用了目前比较先进的XML(ExtensibleMarkupLanguage，可扩展标记语言)半结构化数据库模型来进行数据库的管理，对于其中的数据查询功能利用XML文档查询技术来实现。

通过对国内外XML文档查询算法的分析发现，大多数算法的实现是把被查询文档全部载入内存之后再进行处理，因此要消耗大量内存，尤其是在XML文档很大以致无法全部载入内存的情况下，这些算法就无能为力了。

针对这一问题，本文在统一资源管理平台查询组件中设计并实现了一种新的查询算法。该算法的设计思路是：首先根据XPath查询表达式，生成一个查询自动机，并将查询条件隐含在查询自动机的结构和状态中；然后通过SAX(SimpleAPIsforXML，XML简单应用程序接口)解析程序将XML文本数据流转化为SAX事件流，并将这些事件作为查询自动机的输入，来触发查询自动机的状态转换；查询自动机依据不同的输入事件，在各个状态之间进行转换，一旦确认某一部分文档完全匹配查询表达式，查询引擎程序就输出查询结果。

文中详细地介绍了由查询表达式构造查询自动机的步骤；实现了一个基于流的XML文档查询系统的原型，它可以在对XML流的一次单向读取过程中处理XPath，输出查询结果。文中还对基于内存的XML查询算法和基于流的XML查询算法进行测试、比较，并对结果进行了分析。

基于流的XML查询算法是为了满足一些数据密集型应用对数据查询处理的需求而引入的，这类应用处理的数据不宜用持久稳定的关系建模，而应采用数据流建模。这类应用的领域包括金融服务，网络监控，电信数据管理，生产制造，传感检测等。本论文的研究对这类实际应用将具有一定的理论意义和使用价值。

6.会议论文 吕建华.周巍.孙冰.王国仁.于戈 XML查询中RPE索引技术研究 2001

本文针对正则路径达表达式的特点提出了一个基于自动机的正则路径表达式索引方法,它把正则路径表达式转换成自动机,利用其相似性准确而有效地索引正则路径表达式,从而提高正则路径表达式的查询处理效率.

7.学位论文 许翼 XML树模式查询研究 2008

XML是可扩展标记语言的简称，为Web上半结构化文档和数据提供了通用格式。随着Internet的发展尤其是Web技术的广泛应用，越来越多的应用采用了XML技术作为信息表示和数据交换的标准，这使得通过数据库技术对XML数据进行存储、查询等操作变得越来越重要。由于XML文档可看作树型，对XML的查询可以看成是基于值谓词的子树Twig匹配，也就是从XML文档找到和给定查询条件相匹配的子树，因此能够高效找到子树匹配成为XML中的一个核心问题。

本文在介绍了XML查询的研究现状和研究成果的基础上，针对Twig Stack算法在多层嵌套的XML文档下性能不高的情况，提出了改进算法Nest Twig，并证明行之有效。 Nest Twig算法和Twig Stack算法都是基于分解—匹配—合并的步骤来进行查询匹配处理，容易产生大量无用的中间结果或者会对一些子模式树进行重复匹配。为了解决这个问题，本文引入自动机思想，对Twig查询自动机与路径模式树的匹配进行了讨论，提出了改进算法Pattern Match。对Pattern Match算法和Nest Twig算法进行了实验对比证明基于自动机的方法在性能上具有较大优势。

8.期刊论文 方峻.刘长毅.徐诚 枪械方案设计系统中的基于实例推理框架研究 -兵工学报2003,24(1)

本文提出了一种基于实例推理(CBR)的面向对象框架原型,阐述了其结构组成和实现形式,研究了其在构建枪械方案设计系统中的应用,同时建立了基于可扩展标记语言(XML)的实例表达形式和一套面向对象的实例推理模型,在此基础上开发了一套枪械自动机方案设计原型系统,并以一个设计实例对系统进行了验证.

9.学位论文 包小源 压缩XML的索引与查询方法研究 2005

XML由于为其中所保存内容的每个语义单元引入一个标记而使整个文件的内容迅速膨胀，这种情况下，在对数量庞大的XML数据进行管理时，进行压缩就成为一种有效的基本解决方法。其实，XML作为一种数据交换标准，由于其频繁在网络上进行传输，为了节省宝贵的网络资源，压缩将是一种必然的选择。

对压缩数据进行查询的基本方法是首先解压缩，然后再实施查询。这种方法在数据量小的情形下尚可，对于所管理数据量庞大的情形下(如IR中的海量数据管理、大规模企业的海量XML客户个性化数据等)，每次查询需要事先解压缩所带来的性能下降是无法承受的。于是，直接基于压缩数据进行索引、查询等就变得非常重要而且必要。这一点对压缩XML数据是同样适用的。但目前已有的压缩方法大多没有兼顾结构保持、解压缩效率、以及压缩比等三个方面的要求，不能支持直接基于压缩的查询；同时，基于压缩的XML索引、查询(包括静态、流两方面的查询)很少，而已有研究不能直接、或者根本不能应用于压缩数据的索引和查询。

围绕上述问题，本文提出了XML结构保持压缩方法、基于压缩的XML索引方法、基于压缩XML静态数据以及压缩XML流的查询及分发方法。主要的创新性成果包括：

1.提出了基于Byte-Huffman的XML结构保持压缩思想和方法

目前已有的XML结构保持压缩方法基本上自成系统，无法有效与其他应用系统集成，且其实现复杂。还有很重要的一点，在网络环境下，对压缩和解压缩的速度要求是同等重要的，但目前的压缩方法(无论是非结构保持压缩或是结构保持压缩)因主要集中于实际应用中XML传输及存储效率的提高，从而对解压缩的效率不够重视。经过我们对现有的实验系统测试，其解压缩的速度一般都很慢。

本文提出了一种支持查询的XML压缩方法：QueXComp(QuerableXMLCompression)。它以XByteHuff为基本压缩编码方法，通过保留以“＜”、“＞”，“/”，及“=”等定义的XML结构，以Word为基本压缩统计单位对元素的Tag及元素值实施压缩，可实现高效率的压缩和解压缩。

为了能实现从压缩XML数据中任意位置开始的查询及解压缩，我们对ByteHuffman编码进行了改进，保留其MSB作为标志位以指示一个Word的开始，形成了XbyteHuff。

同时，为了实现基于压缩的高效查询，我们提出了字典索引结构P-Tries，并提出了基于ByteHuffman树中附加Mark域信息以支持所提出的基于NFA(非有限自动机)的压缩XML内容查询。

2.提出了基于压缩的XML结构索引S-Index、ArithRegion、XML内容索引P-Index

索引作为一种经典的、被广泛证明有效的查询辅助方法及技术，当然在压缩的XML数据环境下，有深入研究的必要性。但目前所提出的很多索引结构大多基于XML源数据(非压缩)，不能直接应用于压缩XML数据，甚至对于某些压缩(如算术压缩)方法，这些索引结构根本不适用。尤其对于实际中频繁使用的查询Q=//element1/element2/…/elementm[textO=string]，现有索引结构不能对其进行高效处理。

本文首先提出了适用于字典压缩的XML结构索引：S-Index。S-Index是一种基于后缀树(SuffixTree)的XML索引结构。S-Index的构造通过只对XML数据树遍历一次以及在S-Index中引入后缀链(Sufflink)的方法，从而达到较低的构造代价。S-Index中所有节点利用Hash表保存到其所有子节点的指针，可以证明，查询//element1/element2/…/elementm的处理代价仅为O(m)，优于其他索引结构。

接下来，针对基于XML结构的算术压缩，提出了索引结构ArithRegion，该索引结构以B+树为基本结构，以压缩的左边界值为关键字，可实现对算术压缩XML结构的有效索引。

最后，提出了压缩XML内容索引结构P-Index。P-Index以Patricia&Tries树为基本结构。在P-Index的每个叶结点，有一个倒排表(InvertedList)与之关联，以表示该结点对应的Word在整个XML文档中的位置信息。利用P-Index可处理查询Q的内容过滤部分(/elementm[text()=string])。结合利用结构索引所得到的结构查询部分的结果，我们可以代价O(m+|string|+c*logB(|L|))实现对Q的查询处理。

3.提出了压缩XML查询代数ACX

利用代数来表达查询并以此为基础进行逻辑查询优化是一种基本方法。我们研究并提出了压缩XML查询代数ACX，它有利于准确表达查询语言的语义，对查询优化的讨论可不依赖具体的物理实现。具体内容包括：

①提出了基本的数据类型及其逻辑操作符，并给出了一系列逻辑优化的规则；

②针对我们所提出的压缩方法(QueXComp及反向算术压缩)、压缩XML的存储方法(XCompStore)以及索引结构(S-Index、ArithRegion以及P-Index)给出了基本查询的物理操作符及其实现；

③给出了逻辑操作符物理实现的一般方法。

4.提出了压缩XML流的高效查询及分发方法

我们提出了针对压缩XML流进行高速查询及分发机制BloomRouter：①以保持结构的反向算术压缩为基本压缩方法，提出了针对XML压缩流的查询及分发方法；②提出了基于BloomFilter的压缩流非查询相关内容过滤方法，以减轻分发中心的实际查询处理及分发负荷；③提出了基于区间树的针对大量查询语句的索引结构，以使查询相关内容可快速与对应查询匹配并实现分发。