利用自动机高效处理 XML 路径表达式查询

王国仁 于勇前 孙 冰
（东北大学信息科学与工程学院 沈阳 110004）

摘 要 在 XML 查询处理中，应用于绝大多数 XML 查询语言中的路径表达式在定位和查询 XML 数据和数据的结构关系方面具有极强的表达能力，并且由于 XML 数据的半结构化性，使得 XML 路径表达式查询的查询处理技术的研究与传统的数据库查询处理技术相比有着全新的特点和挑战. 一些目前已有的查询处理技术可以用来处理路径表达式，但是查询处理中产生的大量中间结果导致了这些方法应用在大规模 XML 文档和复杂的路径表达式查询中时查询效率急剧下降. 文中利用自动机技术设计了一个处理 XML 路径表达式查询的高效方法——SAM. SAM 的基本思想是将路径表达式查询转化成一个与之完全等价的自动机，然后将其与从 XML 文档中抽象出来的模式路径相匹配. 文中同时也给出了基于 SAM 方法的针对路径表达式中"//"操作符计算的有效解决方案. 实验证明：SAM 是一种非常有效的查询方法，在计算大数据量复杂路径表达式查询时具有非常高的效率，是一种实用的 XML 路径表达式查询方法.

关键词 XML 路径表达式；自动机；查询处理
中图法分类号 TP311

Efficiently Processing XML Path Queries Using Automata

WANG Guo-Ren YU Yong-Qian SUN Bing
(*School of Information Science and Engineering, Northeastern University, Shenyang* 110004)

Abstract In XML query processing, path expressions applied in most XML query languages have a powerful ability in locating and querying XML data as well as structural relationships. Due to the semi-structured feature of XML data, the query processing techniques of XML path expression query has new characteristics and challenges compared to traditional database query processing techniques. So far, some techniques have been proposed for processing path queries. However, when they are applied to large scale of XML documents and complicated path expressions, their performance degrade dramatically. This paper proposes a high efficient XML path expression query processing method——SAM based on the automata technique. Its basic idea is transforming a path expression query to an equivalent automata and matching the automata with the schema paths abstracting from the XML document. This paper also presents an approach to computing the "//" operation based on the SAM method. The experimental results show that SAM is an efficient and practical method for computing complicated path expression queries on large scale XML documents.

Keywords XML path expression; automata; query processing

收稿日期：2004-10-25；最终修改稿收到日期：2007-07-05. 本课题得到国家自然科学基金（60573089）和国家"九七三"重点基础研究发展规划项目基金（2006CB303103）资助. **王国仁**，男，1966 年生，博士生导师，主要研究领域包括 XML 数据管理、P2P 数据管理、生物信息学等. E-mail：wanggr@mail.neu.edu.cn. **于勇前**，男，1966 年生，博士研究生，主要研究方向包括 XML 数据管理、数据挖掘等. **孙 冰**，男，1978 年生，博士研究生，主要研究方向为 XML 数据管理.

1 引　言

随着 Internet 技术的发展，越来越多的应用系统采用了 XML 语言[1]作为数据表示和信息交换的标准，这使得通过数据库技术对 XML 数据进行存储、查询等操作变得日益重要. 由于众多的 XML 数据查询语言（如 XQuery[2]，XPath[3]和 Lorel[4]等），都采用路径表达式作为它们的主要组成部分之一，XML 路径表达式查询在目前有关 XML 数据管理技术的研究中处于非常重要的地位.

目前对于路径表达式查询的处理方法主要有查询重写、XML 数据树遍历以及结构连接等等. 查询重写的方法主要是针对将 XML 数据存储在关系数据库内的情况，通常手段是将 RPE 重写成若干个简单路径表达式（SPE）的集合，然后使用关系之间的连接来计算 SPE. 例如 VXMLR[5]系统中，包含“//”和“*”的 RPE 查询被重写为简单的路径表达式，再根据关系数据库模式翻译成 SQL 语句进行处理. 由于 XML 的数据模型与关系模型并不兼容，在查询路径表达式时需要大量的转换操作，查询效率相对比较低. 通过对 XML 数据树进行遍历是一种简单有效的计算路径表达式的方法. Lore[6]系统提出了三种 SPE 查询策略：自顶向下（Top-down）、自底向上（Bottom-up）和混合策略（Hybrid）. 由于这种方法在查询时遍历数据树，所以查询效率受 XML 数据量的影响相当大，如果数据量很大，那么遍历方法的查询效率就会非常低.

很多研究都集中在使用结构连接方法解决路径表达式查询问题. XISS[7]系统提出了路径连接（path join）的方法用来计算正则路径表达式查询，并给出了 $\mathcal{EE}$-Join 算法计算“//”以及 $\mathcal{KC}$-Join 算法来计算“*”. 文献[8]以路径连接方法为基础提出了一个索引结构 APEX，根据查询负载情况建立自适应性的路径索引. 在以前的工作中[9]，我们详细定义了元素外延等概念，并设计了若干的索引结构，包括元素索引、属性索引、引用索引和路径索引等等以支持外延连接算法. 文献[10]提出了结构连接的概念并给出两个算法 tree-merge 和 stack-tree 用以计算路径表达式中的“//”操作符. 最近还有一些研究致力于在结构连接算法基础上使用索引以加快其操作，如使用基于 B+树的索引[11]以及 XR-Tree[12]等. 从效率上来讲，结构连接方法较之 XML 数据树遍历的方法有很大的改进，对于比较简单的查询，结构连接的速度非常快. 然而，对于比较复杂的查询，如长路径查询，XPath 中带有“//”操作的查询以及带有正则路径表达式中闭包操作的查询等，结构连接方法的效率并不十分令人满意. 尤其是闭包运算，到目前为止还没有效率很高的解决方法.

本文拟采用有限状态自动机的相关技术解决以上问题. 由于路径表达式查询可以表示成一个有限自动机，而 XML 数据可以被抽象成为若干路径的集合. 因此路径表达式的查询问题就可以转化为有限自动机的匹配问题. 文献[13]使用 SAX 接口的事件函数来驱动查询自动机的状态变迁，从而完成将 XML 文档分发给相应用户的任务. 然而，在 XML 数据库查询中，却不可能使用 SAX 接口驱动. 同时，XFilter 系统的自动机匹配过程结果得到的是某一文档是否符合某一用户以查询形式给出的要求，而不是该查询在该文档中的查询结果.

本文利用自动机技术设计了一个处理 XML 路径表达式查询的高效方法——SAM. SAM 的基本思想是将路径表达式查询转化成一个与之完全等价的自动机，然后将其与从 XML 文档中抽象出来的模式路径相匹配. 本文同时也给出了基于 SAM 方法的针对路径表达式中“//”操作符计算的有效解决方案以及实用的优化方法. 本文在后面给出了对 SAM 方法进行性能测试的实验结果，并加以分析.

本文第 2 节给出与 SAM 方法相关的基本概念和定义，包括查询自动机、模式路径和路径表达式等；第 3 节介绍如何将路径表达式查询转化为查询自动机；第 4 节介绍模式自动机的概念以及用法，并给出两种优化策略；第 5 节详细讨论 SAM 算法并给出相关的具体算法和数据结构；第 6 节给出测试结果并进行性能分析；最后第 7 节总结全文.

2 基本概念及定义

本节简要介绍本文中涉及到的与 XML 相关的一些概念以及 SAM 方法中所用到的基本概念的相关定义. 2.1 节介绍与 XML 查询相关的一些基础知识，包括 XML 文档，XML 文档类型定义和 XML 文档对象模型等，同时，为了说明这些概念而提出的相关示例将会在全文范围内作为例子使用；2.2 节介绍实例路径与模式路径的概念；2.3 节说明 SAM 方法所针对的路径表达式查询；2.4 节给出查询自动机的概念.

2.1 XML 数据模型

作为一种半结构化的标记语言，XML 使用元素和属性以及元素之间的嵌套包含关系来表示 XML 数据间的层次关系，图 1 给出了一个 XML 文档的例子. 在这个示例文档中，一个 Document 元素有一个名为 Doc_id 的属性以及两个子元素 Title 和 Chapter；而 Chapter 元素有一个名为 Id 的属性和 3 个子元素 Author、Head 和 Section. 一个 Document（文章）可能包含多个 Chapter（章），而一个 Chapter（章）又可能包含多个 Section（节）. 类似的，一个 Section 元素也有一个名为 Id 的属性和 3 个子元素 Head、Paragraph 和 Section，一个 Section（节）可能包含多个 Paragraph（段落）和 Section（节，实际上是小节）.

```
〈Document doc_id = "D1"〉
  〈Title〉T1〈/Title〉
  〈Chapter Id = "C1"〉
    〈Author〉A1〈/Author〉
    〈Head〉CH1〈/Head〉
    〈Section Id = "S1"〉
      〈Head〉SH1〈/Head〉
      〈Paragraph〉P1〈/Paragraph〉
      〈Section Id = "S2"〉
        〈Head〉SH2〈/Head〉
        〈Paragraph〉P2〈/Paragraph〉
        〈Paragraph〉P3〈/Paragraph〉
      〈/Section〉
    〈/Section〉
    〈Section Id = "S3"〉
      〈Head〉SH3〈/Head〉
      〈Paragraph〉P4〈/Paragraph〉
    〈/Section〉
  〈/Chapter〉
〈/Document〉
```

图 1 XML 文档示例

应用程序可以使用 DOM（Document Object Model，文档对象模型）作为编程接口访问 XML 数据，同 XML 一样，DOM 也是 W3C 的推荐标准，它不仅说明了应用程序访问 XML 数据的方法，同时也定义了 XML 文档的逻辑结构和数据模型. 在 DOM 中，XML 数据被抽象成为有元素/子元素联系和元素/属性联系的实体的集合，这些实体可能是元素，也可能是属性. 一个 XML 文档在 DOM 中被组织成一棵数据树，前面所说的实体即为这棵数据树中的节点，而其间的联系正对应于连接树中节点的边.

图 2 给出了图 1 中的示例文档所对应的 DOM 数据树. 在图中，圆形表示元素节点或属性，而矩形表示文本节点，带箭头的边表示实例之间的元素/子元素关系或者元素/属性关系. 由于在本系统中，所有实例节点均存储在数据库中，每个节点在数据库中都用一个唯一的标识（Object ID）表示，这个标识在图形中用以“&”开头的数字代表. 每个元素节点或属性节点都有一个名字，这个名字在图中用符号下面的字符串表示.

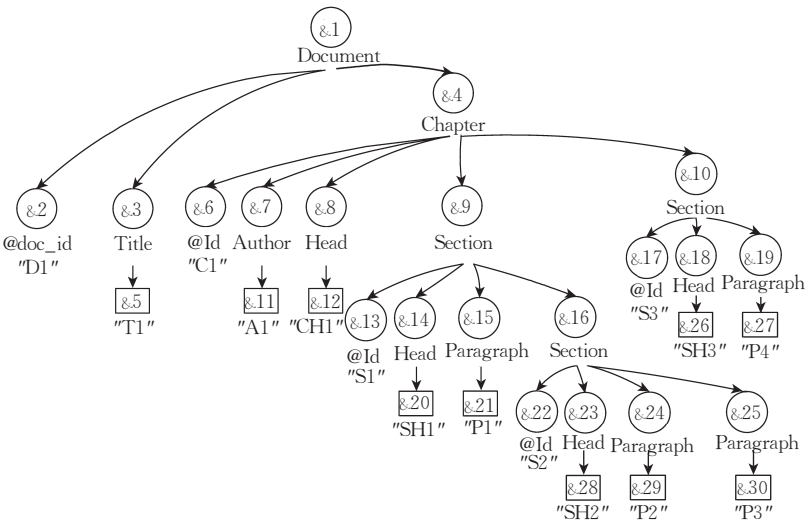

图 2 示例 XML 文档的数据树

一个 XML 文档在 DOM 中被组织成一棵数据树，前面所说的实体即为这棵数据树中的节点，而其间的联系正对应于连接树中节点的边. 举例来说，节点 &9 是一个名为 Section 的元素节点，&4 节点是它的父亲节点，而节点 &15 是它的几个孩子节点之一. 在本文中，我们用符号 D 表示一个 XML 文档以及该文档对应的数据树，同时，D 也被看成是该文档中所有节点的集合.

XML 是一种自描述语言，仅凭 XML 文档自身就可以完整地定义出数据的结构而不需要其它任何的辅助定义，但是为了使 XML 数据符合应用所定义的某些完整性约束并且更高效地处理 XML 数

据，通常使用 DTD(Document Type Definition，文档类型定义）来约束文档的结构，同时对 XML 文档进行有效性验证. 在 DTD 中定义了可能出现在一个 XML 文档内的所有元素、这些元素之间的嵌套方式以及这些元素可能带有的属性等等. 出于直观的考虑，类似 DOM 树，DTD 也可以用图形的形式来表示. 但是不同于 DOM，DTD 在逻辑上是一个图而非一棵树. 图 3 给出了图 1 所示的示例文档对应的 DTD 图.

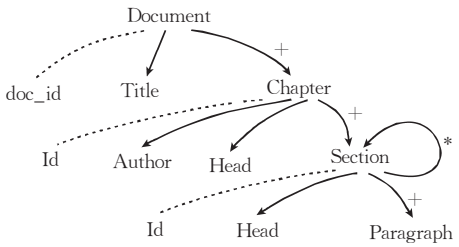

图 3　示例 XML 文档的 DTD 数据树

2.2　实例路径与模式路径

XML 文档以树的形式组织，这样，每一个文档树上的 XML 节点都对应一个从根节点到它自身的路径，这一路径被称作实例路径，以区分将要提到的模式路径. 定义 1 给出了这一概念的定义.

定义 1.　对于一个 XML 文档 D 和其中的任一节点 i，i 对应的实例路径是一个由 D 中的节点组成的序列，表示为 $\mathcal{P}_I(i)=\langle n_1,n_2,\cdots,n_k\rangle$，其中，$n_1=root(D)$，$n_k=i$ 且 $n_l=parent(n_{l+1})$，这里 $(l=1,2,\cdots,k-1)$.

在定义中，$root$ 是计算文档根节点的函数，而 $parent$ 是计算当前节点的父亲节点的函数，类似这样的函数在 DOM 标准中都有相应的定义，本文仅在使用的时候加以说明，而不进行重复的定义. 根据图 2 可以直观地给出实例路径的例子，其中节点 &5 和 &23 的实例路径分别为 ⟨&1，&3，&5⟩ 和 ⟨&1，&4，&9，&16，&23⟩. 显然，一棵 XML 树上的节点与实例路径有一一对应的关系.

在 XML 数据树中，每一个节点都被赋予了一个类型，如元素节点或属性节点的名字，本文使用“节点模式”以确切定义上述类型的概念.

定义 2.　对于一个 XML 文档 D，对于所有的 XML 节点 $i\in D$，则节点 i 的节点模式的值是一个字符串，用 $\varphi(i)$ 表示，根据节点类型的不同分别定义为：

(1) 若 i 是一个文档节点，则 $\varphi(i)$ 的形式定义为 $doc(\langle url\rangle)$，其中 $\langle url\rangle$ 代表该文档在 Internet 上的地址；

(2) 若 i 是一个元素节点，则 $\varphi(i)$ 定义为 i 的名字；

(3) 若 i 是一个属性节点，则 $\varphi(i)$ 定义为前缀“@”加上 i 的名字；

(4) 若 i 是一个文本节点，则 $\varphi(i)$ 定义为常量字符串“text()”.

表 1 总结了定义 2 中的各种情况，并给出了相应的例子. 从定义 2 可以看出，所有的文本节点具有相同的节点模式，而对于元素节点和属性节点来说，两个节点具有相同的节点模式当且仅当他们的名字相同. 文档节点的节点模式值是一个与文档位置和名字有关的函数，然而由于在本文中通常都是在同一个文档内部进行讨论，因此在不产生混淆的时候使用另外一个常量字符串“/”表示文档节点的节点模式. 对于一个 XML 文档 D，其中所有节点的节点模式的集合用 Σ_D 表示，即 $\Sigma_D=\{\varphi(i)\mid i\in D\}$. 在本文中我们使用拉丁字母如 a，b 等表示节点模式. 我们另外使用一个函数 $Inst(a)$ 表示节点模式为 a 的所有 XML 实例节点，即 $Inst(a)=\{i\mid i\in D\wedge\varphi(i)=a\}$.

表 1　节点模式的意义及表示

节点类型	节点模式定义	示例
Document	以描述文档位置的URI作为参数的函数	doc("e1.xml")
Element	元素名字	Chapter
Attribute	属性名字加上“@”前缀	@Id
Text	常量字符串	text()

与实例路径相对应，在一条路径上的所有实例的节点模式同样构成了一个序列，这一序列称为模式路径，其定义由定义 3 给出.

定义 3.　对于一个 XML 文档 D 和其中的任一节点 i，i 对应的模式路径是一个由 i 所对应的实例路径上的节点的模式组成的序列，用 $\mathcal{P}_S(i)$ 表示. 若 $\mathcal{P}_I(i)=\langle n_1,n_2,\cdots,n_k\rangle$，则节点 i 对应的模式路径 $\mathcal{P}_S(i)=\langle\varphi(n_1),\varphi(n_2),\cdots,\varphi(n_k)\rangle$.

对于一个 XML 文档 D，其中所有节点对应的模式路径的集合用 Ψ_D 表示，即 $\Psi_D=\{\mathcal{P}_S(i)\mid i\in D\}$. 显然，$\Psi_D\subset\Sigma_D^*$. 本文中使用希腊字母 α 和 β 等表示模式路径. 为了简化模式路径的表示方法，我们把路径模式 $a_1,a_2,\cdots,a_k$ 表示为 $/a_1/a_2/\ldots/a_k$. 在一个 XML 文档中，每个节点都对应唯一的一条模式路径，但是一条模式路径可能对应多个节点. 例如在图 2 中，节点 &9 对应唯一一条模式路径为 Document/Chapter/Section，而节点 &10 也对应了同样一条路径. 我们使用 $Inst(\alpha)$ 表示模式路径 α 对

应的所有 XML 实例节点的集合.

2.3 路径表达式查询

目前,可以用于查询 XML 数据的查询语言有很多,虽然 W3C 将 XPath 和 XQuery 列为推荐标准,但是随着 XML 相关技术的不断发展和完善,所有的 XML 查询语言仍然在继续发展,还不能断言哪一种语言最适合 XML 数据,最具生命力.但是,由于 XML 数据的半结构化特性,这些查询语言都不约而同地采用路径表达式作为语言的核心部分.路径表达式用一种极其简洁的方式表述查询要求,可以很方便地用来在 XML 数据树上定位节点.各种查询语言中所使用的路径表达式的形式也并不统一,其中一种最典型的路径表达式是正则路径表达式.正则路径表达式采用正则表达式的形式定位查询路径,使用正则操作符如"/"、"|"、"?"、"*"和"+"等连接基本的查询表达式.为了与大部分的查询语言中的路径表达式写法相统一,本文使用"/"表示连接操作,而其它一些文献中可能使用"."等符号.XPath 和 XQuery 中还使用了另外一个操作符"//",用来表示 XML 数据中祖先/后代关系的连接运算,这一操作符非常重要而且如果采用通常的方法很难计算.本文提出的 SAM 方法可以支持包含上述所有的操作符的路径表达式查询,为了清楚说明 SAM 方法,我们使用图 4 的形式定义路径表达式查询.

```
PathExpr ::= PathExpr '/' PathExpr
    | PathExpr '//' PathExpr
    | PathExpr '|' PathExpr
    | PathExpr '*'
    | '(' PathExpr ')'
    | NodeSchema
```

图 4 路径表达式的 BNF 定义

图 4 所示的路径表达式的定义并没有包括"?"和"+"操作符,因为这两个操作符可以很容易地通过以下公式消去:

$$E^{+} \Leftrightarrow EE^{*} \Leftrightarrow E^{*}E \tag{1}$$

$$E? \Leftrightarrow E \mid \varepsilon \tag{2}$$

这里 E 表示任意的路径表达式,而 ε 是一个特殊的路径表达式,代表空表达式.下面给出两个基于图 1 所示的 XML 文档的路径表达式查询的例子.

Q_1 找出所有章的题头:Document/Chapter/Head.

Q_2 找出文章中的所有段落:Document//Paragraph.

2.4 查询自动机

对于任意一个正则表达式,都可以找到与之等价的有限状态自动机(FSA).因此,如果一个路径表达式查询是正则的,就可以将之转换成一个 FSA,本文把这种可以表示查询的自动机称为查询自动机,查询自动机所能接收的所有句子集合就是路径表达式所代表的集合,也正是查询的最终结果.对于一个查询 Q,符号 FSA_Q 用来表示该根据该查询生成的查询自动机.一个查询自动机 A 可以表示成一个五元组.

$$A = (K_A, \Sigma_A, \delta_A, s_A, F_A) \tag{3}$$

Σ_A 表示字母表,在这里就是节点模式的集合.除此之外,Σ_A 中还包括两个特殊符号,ε 和 ξ,ε 用来表示"空",而 ξ 表示"所有".ε 在将路径表达式转化为自动机时是必须的,而 ξ 是计算"//"操作符时所需要的,这一点将在第 3 节详细介绍.假设在文档 D 中查询,那么 $\Sigma_A \subset \Sigma_D \cup \{\varepsilon, \xi\}$.$K_A$ 为自动机的状态集合,s_A 为开始状态而 F_A 为结束状态集合,有 $s_A \in K_A$,$F_A \subset K_A$.最后,$\delta_A: K_A \times \Sigma_A \rightarrow K_A$ 为状态转移函数.

3 查询自动机的生成

如前所述,在 SAM 中,路径表达式查询被转换成一个查询自动机,之后再进行匹配运算.首先,该路径表达式被转换成一棵表达式树,例如,路径表达式 $a/b/c//g$ 被转换成形如图 5(a)的形式.由于"//"并不是一个正则操作符,所以含有"//"的表达式树不能被直接转换成自动机,因此我们利用替换规则

$$E_1//E_2 \Leftrightarrow E_1/\xi/E_2 \tag{4}$$

从表达式树上消去"//"操作符.图 5 给出了这一消去过程的例子,经过这一过程,表达式树上只可能出现"/","|"和"*"操作符,在形式上已经与正则表达式相统一,可以非常容易地转换成一个自动机.

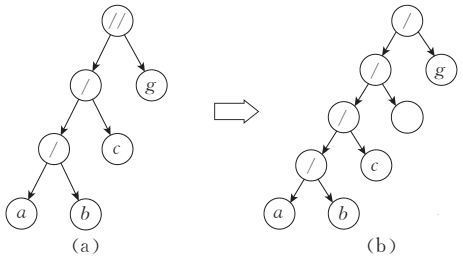

图 5 路径查询表达式树

一棵路径查询表达式树是一棵二叉树,用基本

的遍历算法就可以生成与之完全等价的查询自动机. 根据树中的每一个叶子节点都可以生成一个子自动机，而每个分枝节点根据其类型的不同而使用不同的规则，然后将其两个孩子节点生成的子自动机合成一个. 下面给出了自动机构建的算法，其中 Expression-To-Automata( ) 函数是算法的核心，它用来将以节点 *node* 为根的子表达式树转换成一个自动机. 其中的 New-State( ) 函数每调用一次即返回一个新的自动机状态. 符号 $\hat{\delta}$ 用来表示两个自动机的空连接转移函数，对于任意的两个自动机 M_1 和 M_2，$\hat{\delta}_{M_1}^{M_2}=\{(q,\varepsilon,s_{M_2})\mid q_0\in F_{M_1}\}$，包括了所有从 M_1 的结束状态到 M_2 的开始状态的空转移函数，使用它就可以方便地表示出自动机之间的连接运算. 需要注意的是，这里函数 New-State( ) 和 Expression-To-Automata( ) 都是过程型函数，重复调用会返回不同的结果，因此算法中的 I_1 和 I_2 的值并不相等，因为两个自动机拥有不同的状态. Construct-Schema-Automaton( ) 用来处理 ξ，返回与被查询文档相关的模式自动机，模式自动机的概念及构建算法将在第 4 节具体介绍. 最后，Simplify( ) 是一个标准的过程，用来将自动机化简并加以正规化.

Construct-Query-Automata(Q)
1. $M\leftarrow$Expression-To-Automata(root[Q])
2. Simplify(M)
3. return M

Expression-To-Automata(*node*)
1. $q_0=$New-State( ), $q_1=$New-State( )
2. $I_1\leftarrow$Expression-To-Automata(ε),
   $I_2\leftarrow$Expression-To-Automata(ε)
3. switch
4. case $node\in\Sigma_D$:
5. return$\langle\{q_0,q_1\},\{a\},\{q_0,a,q_1\},q_0,\{q_1\}\rangle$
6. case $node=$'ε':
7. return$\langle\{q_0\},\varnothing,\varnothing,q_0,\{q_0\}\rangle$
8. case $node=$'ξ':
9. return Construct-Schema-Automaton(D)
10. case $node=$'/':
11. $M_1\leftarrow$Cons-Auto($lchild$[$node$]),
    $M_2\leftarrow$Cons-Auto($rchild$[$node$])
12. return$\langle K_{M_1}\cup K_{M_2},\Sigma_{M_1}\cup\Sigma_{M_2},\delta_{M_1}\cup\delta_{M_2}\cup\hat{\delta}_{M_1}^{M_2},s_{M_1},F_{M_2}\rangle$
13. case $node=$'|':
14. $M_1\leftarrow$Cons-Auto($lchild$[$node$]),
    $M_2\leftarrow$Cons-Auto($rchild$[$node$])
15. return$\langle K_{M_1}\cup K_{M_2}\cup K_{I_1}\cup K_{I_2},\Sigma_{M_1}\cup\Sigma_{M_2},\delta_{M_1}\cup\delta_{M_2}\cup\hat{\delta}_{I_1}^{M_1}\cup\hat{\delta}_{I_1}^{M_2}\cup\hat{\delta}_{M_1}^{I_2}\cup\hat{\delta}_{M_2}^{I_2},s_{I_1},F_{I_2}\rangle$
16. case $node=$'*':
17. $M\leftarrow$Cons-Auto($lchild$[$node$])
18. return$\langle K_M\cup K_{I_1}\cup K_{I_2},\Sigma_M,\delta=\delta_M\cup\hat{\delta}_M^M\cup\hat{\delta}_{I_1}^M\cup\hat{\delta}_M^{I_2}\cup\hat{\delta}_{I_1}^{I_2},s_{I_1},F_{I_2}\rangle$

图 6 给出了由 Q_1 和 Q_2 转换成的查询自动机. 其中 Q_2 中的"//"操作符暂时使用 ξ 代替，并将在第 4.3 节中加以解决.

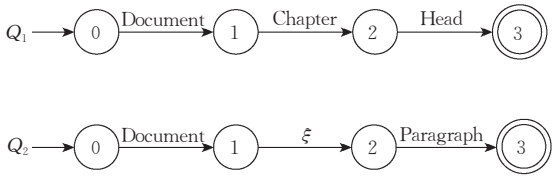

图 6　查询自动机示例

4　模式自动机

第 3 节介绍了如何根据路径表达式查询构造查询自动机，在此转换过程中，如果路径表达式中含有"//"操作符，则利用式(4)消去，替代它的是通配符号 ξ. 本节重点讲述"//"操作符的处理方法，提出了模式自动机的概念用来重写 ξ.

4.1　使用模式自动机重写操作"//"

在 XPath 中，"//"是一个非常特殊的操作符，该操作符的意义是从上下文节点出发找到其所有的后代节点. 由于在传统上我们研究的正则表达式中并不包括这个操作符，因此当前有很多文献专门针对这一操作符进行深入的研究. 早期的做法通常是根据文档的信息将"//"操作符重写为一系列的路径表达式. 例如，Document/Chapter//Paragraph 将会重写成 Document/Chapter/Paragraph，Document/Chapter/Section/Paragraph，Document/Chapter/Section/Section/Paragraph 等多条路径，其中 Section 的具体展开的次数通常要由事前对文档统计而得到的统计信息决定，而对于另外一些结构复杂的 DTD 通常需要进行复杂的转换和计算才能找出重写后的所有可能路径. 由于使用这种方法重写后的路径表达式的数量通常非常多，尤其是文档的 DTD 结构具有递归嵌套结构的时候，导致查询的效率非常低.

因此，人们转而研究具有更高效率的方法. 近期的研究对"//"的处理方法主要是利用结构包含连接的思想，对 XML 实例进行编码，从编码中找出祖先节点与后代节点的关系，将他们连接在一起.

在计算"//"操作符时，重写通常被认为是效率非常低的做法. 然而，上述的方法之所以效率低是因

为为了正确的表述查询语意，需要计算出“//”所能代表的所有路径，并且由于 XML 文档结构的随意性，使得重写生成的路径表达式的数量相当之多，而这些路径表达式又要逐一地进行查询，合并之后才能得到最终的结果. 重写生成的路径表达式的数量直接影响着查询效率. 而在 SAM 中，我们使用查询自动机而不是直接使用路径表达式进行查询，因此，如果我们同样地采用重写策略，那么重写后的结果就不再是许多路径表达式，而是查询自动机的片断. 由于自动机本身可以表示相当复杂的嵌套循环结构，因此可以期待使用简单得多的结构来重写“//”操作符.

在将“//”重写成路径表达式的过程中，由于表达式结构自身的限制，重写过程只能从“如何找到可以正确表达其含义的最少的路经表达式”这一点出发，但如果重写的结果是自动机，那情况就大不一样. 我们在研究中利用了完全相反的思想，试图找到一种“通用的”结构，可以表述所有在文档中可能出现的模式片断，并用这种结构去重写路径表达式中出现的“//”，我们把这种结构称为模式自动机. 之所以如此称呼是因为它包含了并且可以接收文档中的所有的模式片断信息.

下面，我们使用递归的方式给出模式自动机的形式定义，在该定义中说明模式自动机具有以下的特点：

(1) 一个文档的模式自动机可以接收空串 ε，即它的开始状态也是它的结束状态之一.

(2) 模式自动机可以接收文档中的所有模式路径.

(3) 如果模式自动机可以接收某一字符串，那么它也可以接收这个字符串的所有子串.

定义 4. 假设我们使用 $L(M)$ 表示自动机 M 所能接收的语言，对于任意一个 XML 文档 D，自动机 $S=(K_S,\Sigma_S,\delta_S,s_S,F_S)$ 是 D 的模式自动机当且仅当：

(1) $\Sigma_S\in\Sigma_D$；

(2) $\varepsilon\in L(S)$，即 $s_S\in F_S$；

(3) $\Psi_D\subseteq L(S)$；

(4) 对于 $\forall\alpha\in\Psi_D$ 和 $|\alpha|>1$，如果 $\alpha\in L(S)$ 则 $left(\alpha,|\alpha|-1)\in L(S)$ 且 $right(\alpha,|\alpha|-1)\in L(S)$.

上面的定义说明了只要一个自动机可以接收某一文档中的所有模式路径以及其子串，那么就可以称之为该文档的模式自动机. 显然，这样一种结构可以完全地替代“//”操作符作为查询的输入，得到正确的查询结果. 在第 2 节中提到 SAM 在处理“//”的时候，首先把它变成一个含有符号 ξ 的表达式. ξ 是查询自动机中的一个特殊字母，用来表示任意的序列. 在实际的匹配当中无法处理 ξ，需要在匹配操作之前将其重写，在重写过程中，每一个出现在查询自动机的 ξ 都用一个模式自动机所代替.

模式自动机可以根据 XML 文档、DTD 或 XML Schema 进行建立，也可以利用 5.1 节将要介绍的模式路径树进行构建，从效率上考虑，如果系统中已经为 XML 文档建立了模式路径树，那么直接利用模式路径树构建模式自动机最为适合及方便. 其基本思想是利用以上包含了 XML 文档中的足够的模式结构的数据结构，建立一个可以接收 XML 文档中的所有路径模式的自动机，我们称之为 XML 文档的结构自动机. 然后，我们在结构自动机中加入空转移，使之可以接收这些路径模式的所有子串，最后将自动机确定化和最小化. 下面的算法给出了如何根据 XML 文档建立模式自动机.

```
Construct-Schema-Automata(D)
1. T←Construct-Structure-Automata(D)
2. s_S←New-State(), f_S←New-State(), δ_S←δ_T
3. for each q in K_T
4.     do δ_S←δ_S ∪ {(s_S, ε, q), (q, ε, f_S)}
5. S=⟨K_T ∪ {s_S, f_S}, Σ_T ∪ {ε}, δ_S, s_S, {f_S}⟩
6. Simplify(S)
7. return S

Construct-Structure-Automata(D)
1. t←New-State(), map[name[root[D]]]←t,
      waitlist←children[root[D]]
2. s_T←New-State(), K_T←{s_T, t},
   δ_T←{(s_T, name[root[D]], t)}
3. while waitlist≠∅
4.   do n←pop(waitlist)
5.     if ∃map[name[n]]
6.       then t←map[name[n]]
7.       else t←New-State()
8.         map[name[n]]←t
9.         K_T←K_T ∪ {t}
10.    δ_T←δ_T ∪ {(map[name[parent[t]]],
                name[n], t)}
11. T←⟨K_T, Σ_D, δ_T, s_T, K_T⟩
12. return T
```

Construct-Structure-Automata 算法根据文档的结构构造一个与文档结构相对应的结构自动机，该自动机可以接受所有从文档根节点出发的路径模式片断. 算法遍历文档数据树，将文档中出现的所有

父子关系加入自动机的转移函数中. 与模式自动机的要求不同,结构自动机并不能接受所有不从根节点出发的路径模式片断,Construct-Schema-Automata 解决了这个问题. 该算法在结构自动机的基础上新增了两个状态作为模式自动机的开始状态和结束状态,并在开始状态和结束状态与结构自动机的所有原有状态之间加入空状态转移函数. 最后经过自动机的正规化,得到化简后的模式自动机.

4.2　优化模式自动机

在后面的测试中我们会看到,使用模式自动机重写"//"操作符是一个相当有效的方法,尽管如此,本小节将探讨对这一方法进行进一步优化的可能性和方法. 下面我们将提出两种方法,一种是缩减模式自动机的大小的方法(RSA),而另一种是将模式自动机持久化的方法(PSA).

我们使用的自动机可以接收 XML 文档中的任何路径模式片断,而在某一个具体的路径表达式中的某一特定位置出现的"//"所能代表的路径模式是比较有限的. 也就是说,考虑到"//"操作符的上下文关系,可以使用一个相对来说比模式自动机小的自动机来进行重写. 基于这样的考虑,一个模式自动机应该可以利用上下文而缩减其大小,并且不改变其意义. 这里,"//"操作符的上下文关系我们用两个集合表示,一个称之为前驱集,该集合中包含了所有可能直接出现在"//"之前的模式,另一个称为后继集,包含所有可能直接出现在它后面的模式. 例如,对于路径表达式 Document/Chapter//Paragraph 中的"//"操作符的前驱集合为{Chapter},后继集合为{Paragraph}. 而对于路径表达式 $a/(b|c)//(d/e|f)*/g$ 中的"//"操作符,其前驱集合为$\{b,c\}$,后继集合为$\{d,f,g\}$. 下面给出的算法 Calculate-Preceding-Following-Sets 用来计算一个路径表达式中所有节点的前驱集合和后继集合,并把结果放入表 PS 和 FS 中. 算法 Head 和 Tail 分别用来计算可能出现在某一子路径表达式开头和结尾的模式,而 May-Empty 算法计算某一子路径表达式是否可能为空. 可能为空的路径表达式的形式有 ε,$(E)*$等.

```
Calculate-Preceding-Following-Sets(n)
  if n∈ε,ξ∪Σ_D
    then return
  n_l←lchild[n],e_l←May-Empty(n_l)
  h_l←Head(n_l),t_l←Tail(n_l)
  n_r←lchild[n],e_r←May-Empty(n_r)
  h_r←Head(n_r),t_l←Tail(n_r)
  switch
    case n='/':
      PS[n_l]←PS[n],FS[n_r]=FS[n]
      FS[n_l]←h_r∪(e_r? FS[n]: ∅)
      PS[n_r]←t_l∪(e_1? PS[n]: ∅)
    case n='|':
      PS[n_l]←PS[n_r]←PS[n]
      FS[n_l]←FS[n_r]←FS[n]
    case n='*':
      PS[n_l]←PS[n]∪t_l
      FS[n_l]←FS[n]∪h_l

Head(n)
switch
  case n=ε:return ∅
  case n = ξ: return{ξ}
  case n∈Σ_D:return n
  case n= '/':return Head(lchild[n])∪
      (May-Empty(lchild[n])?Head(rchild[n]):∅)
  case n='|' return Head(lchild[n])∪
      Head(rchild[n])
  case n='*'return Head(lchild[n])

Tail(n)
switch
  case n=ε: return ∅
  case n=ξ return{ξ}
  case n∈Σ_D: return{n}
  case n='/':return Tail(rchild[n])∪(May-
      Empty(rchild[n])?Tail(lchild[n]):∅)
  case n='|': return←Tail(lchild[n])∪
      Tail(rchild[n])
  case n='*': return 〈Tail(lchild[n])

May-Empty(n)
switch
  case n∈{ε,ξ,'*'}:return true
  case n∈Σ_D return false
  case n='/':return May-Empty(lchild[n])∧
      May-Empty(lchild[n])
  case n='|':return May-Empty(lchild[n])∨
      May-Empty(lchild[n])
```

利用上述这两个集合所表示的操作符上下文,可以通过缩减模式自动机大小的方式对重写进行优化. 我们假设查询路径表达式具有 $E_1//E_2$ 的形式,其中 E_1 和 E_2 分别为"//"操作符两侧的子路径表达式,并且我们假设该操作符的前驱集合和后继集合分别为 PS 和 FS. 如果 $\omega_1/x_1/\omega_2/x_2/\omega_3$ 是一条符合该表达式的模式路径,其中 x_1 和 x_2 为模式而 ω_1,ω_2 和 ω_3 为模式的序列,并且 ω_2 由查询表达式中的"//"生成. 那么显然,$x_1\in PS$ 且 $x_2\in FS$. 由于

模式自动机可以接收 XML 文档中的任何模式片断，那么它也可以接收 $x_1/\omega_2/x_2$. 这样，对于所有可能的 ω_2，接受它的自动机是模式自动机的一部分，这一部分的开始状态由模式自动机的开始状态读 x_1 到达，而它的结束状态读入 x_2 可以到达模式自动机的结束状态. 在这样的一种情况下，“//”可以用这一部分进行重写而不需要是整个自动机. 我们扩展开来，当 x_1 和 x_2 分别取遍 PS 和 FS 中的所有值，将所有这些情况所生成的部分叠加在一起就得到了可以完全替代模式自动机的部分，这一部分就是经过缩减后的自动机. 下面给出了用于缩减模式自动机优化的 Reduce-Schema-Automata 算法(RSA).

```
Reduce-Schema-Automata(S,Q,n)
 1. PS[Root[Q]]←FS[Root[Q]]←{ξ}
 2. Calculate-Preceding-Following-Sets(Root[Q])
 3. if ξ∈PS[n]
 4.    then waitlist←K_S
 5.    else waitlist←{δ_S(s_S,a)|a∈PS[n]}
 6.       S_S←wl,vs←∅,rs←∅
 7. while wl≠∅
 8.    do p←pop(wl),vs←vs∪{p}
 9.       for each b that δ_S(p,b)=q exists
10.          do if b∈FS[n]
11.             then rs←rs∪{p}
12.          if q∉vs
13.             then wl←wl∪{q}
14. F_R←rs,vs←∅
15. while rs≠∅
16.    do r←pop(rs),rs←rs∪{r}
17.       for each c that δ_S(w,c)=r exists
18.          do if w∉vs
19.             then rs←rs∪{w}
20. K_R←vs,I←Expression-To-Automata(ε)
21. δ_R←δ_S∩(K_R×Σ_S×K_R)∪({(s_I,ε)}×S_S)
22. R←⟨K_R∪K_I,Σ_S,δ_R,s_I,F_R⟩
23. return R
```

该算法对 XML 文档的模式自动机进行两次遍历，第一次遍历从自动机的开始状态出发，搜索所有由开始状态读入前驱集合中任意模式后再读入任意模式串后所有可以到达的自动机状态；第二次从自动机的结束状态出发，在第一次遍历所得到的状态中搜索读入任意一个以后继集合中模式结尾的模式串后可以到达自动机结束状态的状态. 最后，只保留这些状态以及这些状态之间的转移，除去模式自动机中其他无关的状态及转移，生成缩减模式自动机. 缩减模式自动机通常比原始的模式自动机小很多，在查询时需要用模式自动机重写时使用缩减模式自动机就可以减小匹配时的查询自动机，从而提高查询效率.

对于前面所说的方法，模式自动机需要在查询时动态构建，这样一个过程要耗费一定的时间. 实际上模式自动机的结构相对来说十分稳定，对于有相同 DTD 的文档来说，它们的模式自动机结构通常十分相似，可以合并成同一个模式自动机. 因此，将模式自动机作为一种特殊的索引结构进行持久化，在查询时读入模式自动机而不是重新构建，是一种简单而又有效的优化方法. 这种优化方法我们称之为持久化模式自动机(PSA).

上述两种优化方法(RSA 和 PSA)可以结合使用，即将模式自动机作为索引存储在磁盘上，在需要进行重写操作时，将模式自动机读入内存，根据前驱集合与后继集合优化模式自动机，最后进行重写操作.

5 自动机匹配查询算法

前面第 2 节介绍了 SAM 方法中的一些基本概念，包括实例路径、模式路径、路径表达式查询和查询自动机等. 本节介绍如何结合各种索引和数据结构来使用这些概念完成 SAM 方法.

图 7 给出了 SAM 方法的查询流程，其中索引部分由持久化模式自动机、模式路径树和实例路径树组成，这些索引的建立是在 XML 文档加载时完成的. 输入的查询语句如果包含“//”操作符，则通过持久化模式自动机经化简后共同生成查询需要的查询自动机，而查询自动机进一步与模式路径树相匹配，得到符合查询的模式节点，最后在实例树上取得查询结果.

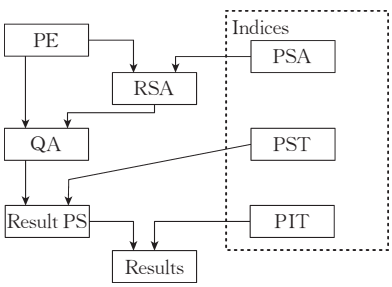

图 7 SAM 方法查询示意图

5.1 模式路径树与实例树

模式路径是由节点模式构成的句子，如果某一条路径模式 α 可以被查询自动机所接收，那么 $Inst(\alpha)$ 中的所有实例都是查询的最终结果. 逐个验证每条

路径模式是否可被自动机接收是可行的，但是十分耗时，因为这种做法会重复匹配带有相同前缀的模式路径. 因此，一个有效的改进方法是使用树型结构合并模式路径，这就形成了一棵模式路径树，如图 8 所示. 而对于 XML，所有的实例节点就是由树形所组织，这样就形成了一棵实例树.

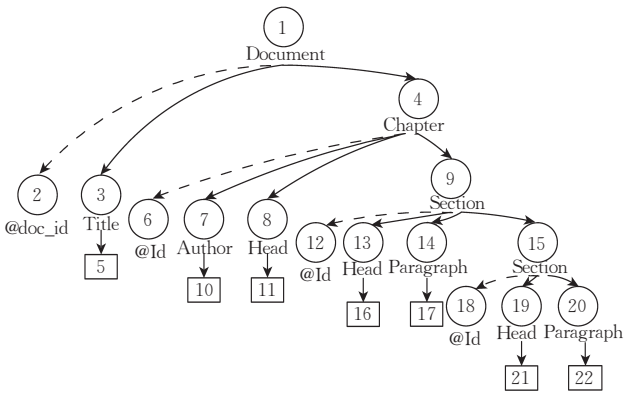

图 8　路径模式树

模式路径树在实现上采用了类 B 树的结构，在构建时尊循以下原则：

(1) 每个模式路径树的节点是查找操作的基本单位，为每个模式树节点分配一个 ID，此 ID 为 B 树查找时的关键词；

(2) 模式树节点中存储节点模式名、逻辑上的父亲节点 ID 以及该节点对应的实例节点引用；

(3) 在 B 树的每个页内必须保持完整的树结构，即必须有唯一的根节点，如果现有节点不能保持结构，则通过插入虚根节点的方法解决；

(4) 在进行插入操作需要进行页面分裂时，应将根节点提出插入到父亲页面中，保持页面层次顺序与模式树的层次顺序相一致.

为了可以高效地取出与模式树节点对应的实例，将实例按照其对应的模式树节点 ID 排序，用 B+树进行存储，而在模式节点中存储的实例节点实际上是实例 B+树中的一对用来表示范围的指针.

5.2　SAM 查询算法

传统的自动机算法只可以用来接收句子，但是在我们的方法中，查询自动机必须与一棵树进行匹配，为了可以在模式树上找到与自动机相匹配的节点，必须设计新的匹配算法. 下面给出的算法 AutomationMatch 是用来完成此功能的. 该算法的主要思想是从根出发将模式路径树与自动机的状态相匹配，在与结束状态相匹配的节点上取得查询结果. 其中要注意的细节是对于查询谓词的处理，这里采用的方法是使用 MarkExclusive 过程标注以不满足谓词的节点为根的实例子树，而在最终取得结果之前使用 CheckResult 过程检查该对应实例节点是否存有标记，如果没有被标记，说明该节点为查询结果，反之则不是查询结果，应予以排除. 在运行算法进行匹配的过程中，对于不对应任何自动机状态的节点，则不会进一步进行匹配，这样可以达到剪枝的效果，加快匹配的速度.

该匹配过程计算了查询表达式中所有的正则操作符，而唯一的非正则操作符“//”已经在 ConstructQueryAutomata 过程中通过模式自动机的重写消去，所以包含任意操作符的查询路径表达式均可通过 SAM 查询方法得到正确的结果.

```
AutomatonMatch(Q, SPT)
M ← ConstructQueryAutomata(Q)
match[Root[SPT]] ← s_M
waitlist ← {Root[SPT]}
while waitlist ≠ ∅
   do n ← pop(waitlist)
      q ← match[n]
      if q[pred] ≠ null
         then MarkExclusive(mt[n], pit, q[pred])
      if q ∈ F_M
         then for each i in mt[n]
            do if CheckResult(i)
               then rs ← rs ∪ i
      for each c in Children[n]
         do if (∃q)(δ_M(p, c[name]) = q)
            then match[c] ← q
               waitlist ← waitlist ∪ {c}
```

6　性能测试与分析

为了测试 SAM 方法的性能，我们在同样的系统平台上实现了 DOM，EJ 和 SAM 3 种查询方法，对它们在不同的查询表达式类型、不同的文档大小以及不同的查询复杂度上的查询性能进行测试和分析. 其中 DOM 是在 XML 数据树上进行遍历的方法，而 EJ 是使用外延索引进行外延连接来计算 XML 查询的方法. 同时为了测试使用模式自动机重写方法计算“//”操作符的性能以及我们提出的两个优化方法的效果，除了基本的重写方法 SAM 外，还加入了使用缩减模式自动机优化的 SAM+RSA 方法，持久化模式自动机的 SAM+PSA 方法以及同时使用这两种优化的 SAM+RSA+PSA 方法，并将它们与流行的 STJ 方法进行比较.

我们测试时使用的软硬件环境为：PIII933 CPU，128MB SDRAM 主存，20GB 硬盘以及 Mi-

crosoft Windows 2000 Professional 操作系统. 测试系统使用 Microsoft Visual C++ 6.0 编译器编译.

在测试时我们有针对性地选取了两个测试集. 我们在 XMark[14] 测试集上进行 DOM,EJ 和 SAM 方法的测试. XMark 中包含了 20 条查询,可以全面地反映 3 种方法在不同种类的查询之上的性能表现. 但是测试时发现,由于 XMark 查询标准使用了 XQuery 语法,对于大部分查询,在计算出主要的路径表达式的结果之后,往往还要进行一些后处理操作用于分组统计、结果重构等. 对于我们测试的两种方法,这些后处理操作都是相同的,在测试时的消耗时间也是相同的,但是某些后处理操作的时间消耗非常大,容易掩盖 3 种方法在路径查询中的时间差异,如图 9. 并且,XMark 中的一条查询往往包含有多条路径表达式,这样就很难细致地划分路径表达式的类型. 鉴于以上原因,我们将这些查询中包含的所有查询表达式分离出来,合并相同的表达式,最终得到 17 条查询表达式,以这些查询表达式作为查询测试集进行测试. 表 2 分类总结了这些查询表达式的特点. 因为第二类的测试主要针对"//"操作符,而目前并没有标准的专门针对该操作符的测试集. 因此我们自行设计了一组查询表达式作为测试集,如表 3 所示.

表 2 XMark 路径表达式分类

类型	路径
短路径	P2 P6 P8-1 P8-2 P9-3
中等路径	P5 P11-1 P12-1 P20-1 P20-3
长路径	P1 P7 P10 P13 P15 P18

表 3 自行设计的路径表达式

查询	路径表达式
Q_1	$a/b/c//g$
Q_2	$a//e//j$
Q_3	$a/(b\|d)/(g\|h)$

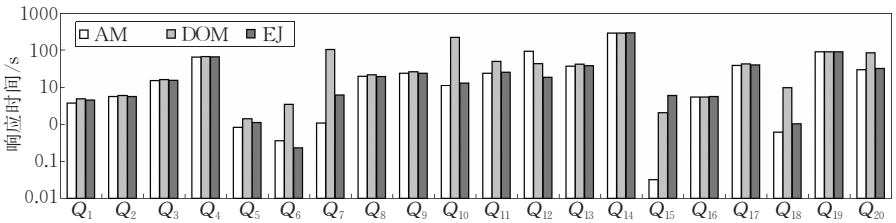

图 9 XMark 查询性能(100M)

图 10、图 11 和图 12 分别给出了使用 SAM,EJ 和 DOM 3 种方法对短路径、中等长度路径和长路径 3 类表达式在规模参数为 1.0(100M XML 文档)的 XMark 测试集上进行测试所得的查询性能. 从图中我们可以看出,对于短路径查询我们提出的 SAM 方法并没有优势,对于所有的查询使用 SAM 方法都比相同条件下使用 EJ 方法要慢. 这是因为 SAM 方法中自动机的构建和启动都需要一定的时间,而对于中等长度的路径情况就有所改观,SAM 与 EJ 相比要稍快一些. 而对于长路径查询,SAM 方法的优势就相当明显,相差最多的 P15 甚至与另外两种方法的响应时间差在 400 倍左右. 从这些图表的对比中,我们可以初步得出 SAM 方法适合长路径查询的结论.

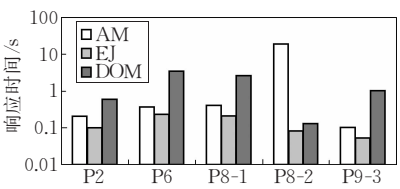

图 10 XMark 短路径查询性能

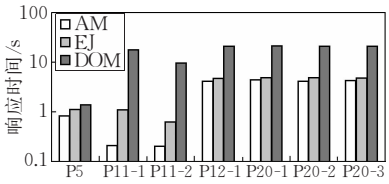

图 11 XMark 中等路径查询性能

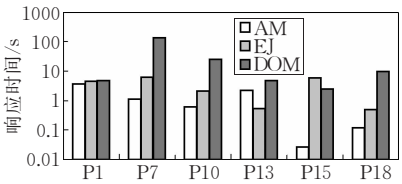

图 12 XMark 长路径查询性能

为了可以更明确地看出这一点,我们选取了最长的一条路径 P15,将它从头截取 2 个路径步,3 个路径步,直到达到所有的路径长度,分别测试它们的查询性能,以观察 3 种方法在路径逐渐增长时的性能表现,结果如图 13 所示. 从图中可以很清楚地看出,虽然 EJ 方法对于 3 步以下的短路径有一定的优势,但是它是随着路径复杂度的增大而性能下降最大的方法,而 DOM 方法是在一定的范围内查询响

应时间逐步增加. 这张图令人惊讶地表明 SAM 方法的查询响应时间几乎不受查询复杂度的影响，甚至复杂表达式查询的响应时间有可能比简单的更少. 我们经过研究发现，SAM 方法的响应时间与查询结果的数量有着密切关系，图 14 给出了对应于图 13 各表达式的查询结果数量，它与 SAM 算法响应时间曲线的走势非常一致.

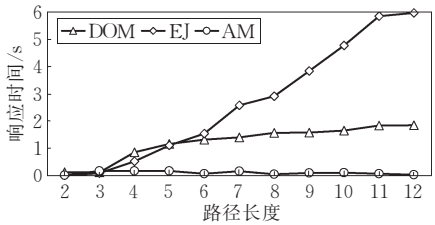

图 13　查询性能随路径长度变化曲线

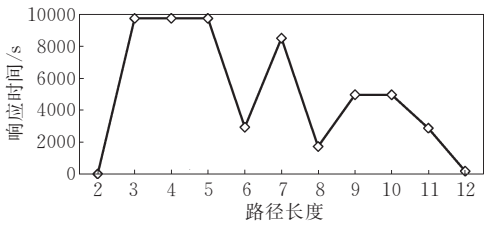

图 14　查询结果数目随路径长度变化曲线

图 15 和图 16 给出了其中两个查询随着数据库中文档大小变化时查询响应时间的变化曲线，从这两个图中我们可以看出 3 种方法在数据库变大时基本上都呈线性增长，其中 SAM 方法和 EJ 方法的增长幅度比 DOM 方法小.

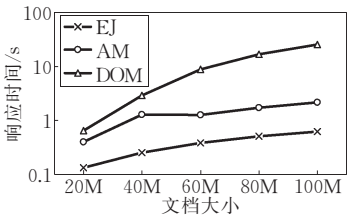

图 15　查询性能随文档大小变化曲线(P10)

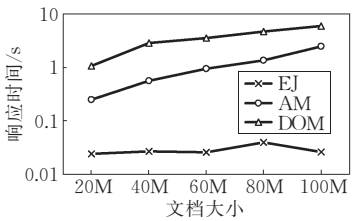

图 16　查询性能随文档大小变化曲线(P15)

图 17 给出了针对“//”操作符进行测试的 5 种方法在 100M 测试文档上的表现. 由图可见，未经优化的 SAM 方法在处理此类查询时已经有很高的效率，可以与 STJ 算法相媲美. 而经过优化的 SAM 方法可以显著地进一步减小查询的响应时间，加快查询的速度. 图中显示 RSA 和 PSA 两种优化手段的效果都很好，效率上来说不相上下，而同时使用两种优化手段的 SAM 方法在 5 种方法中总是最快的，并且相比较而言有很大的优势. 图 18 表示的是上述 5 种方法的查询响应时间随数据库中文档大小变化的曲线，从这张图上可以清楚地看到，文档越大 SAM 族方法的优势越明显.

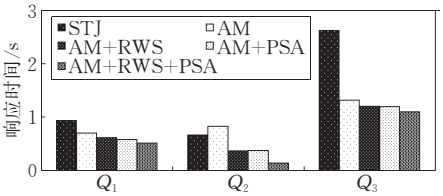

图 17　模式自动机查询性能(不同查询)

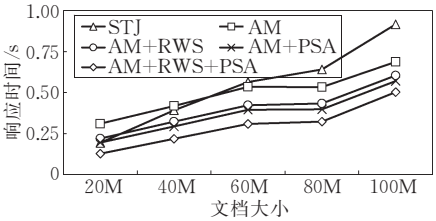

图 18　模式自动机查询性能(随文档大小变化)

7　结　论

本文提出的 SAM 方法是一种相当有效的计算 XML 路径表达式查询的方法. 本文介绍了如何使用 SAM 方法进行路径表达式查询的计算以及如何应用模式自动机重写的方法计算路径表达式中“//”操作符的方法，并给出相应的优化算法. 性能测试的实验结果表明：SAM 方法对于长路径和相对复杂的路径的性能非常好，模式自动机重写方法及其优化方法均非常有效.

参　考　文　献

[1] W3C Recommendation. Extensible Markup Language (XML) 1.0 (2nd Edition). 2000. http://www.w3.org/TR/REC-xml

[2] Fankhauser P. XQuery formal semantics: State and challenges. SIGMOD Record, 2001, 30(3): 14-19

[3] W3C Working Draft. XML Path Languages(XPath), ver 2.0. December 20 2001. Tech. Report WD-xpath20-20011220, W3C, 2001, http://www.w3.org/TR/WD-xpath20-20011220

[4] Abiteboul S, Quass S, McHugh J, Widom J, Wiener J. The Lorel query language for semistructured data. International Journal on Digital Libraries, 1997, 1(1): 68-88

[5] Zhou A, Lu H, Zheng S, Liang Y, Zhang L, Ji W, Tian Z.

VXMLR: A visual XML-relational database system//Apers P M G, Atzeni P, Ceri S, Paraboschi S, Ramamohanarao K, Snodgrass R T eds. Proceedings of the 27th International Conference on Very Large Data Bases. Roma, Italy, 2001: 719-720

[6] McHugh J, Abiteboul S, Goldman R, Quass D, Widom J. Lore: A database management system for semistructured data. SIGMOD Record, 1997, 26(3): 54-66

[7] Li Q, Moon B. Indexing and querying xml data for regular path expressions//Apers P M G, Atzeni P, Ceri S, Paraboschi S, Ramamohanarao K, Snodgrass R T eds. Proceedings of the 27th International Conference on Very Large Data Bases. Roma, Italy, 2001: 361-370

[8] Chung C, Min J, Shim K. APEX: An adaptive path index for XML data//Franklin M J, Moon B, Ailamaki A eds. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. Madison, Wisconsin, USA, 2002: 121-132

[9] Lv Jian-Hua, Wang Guo-Ren, Yu Jeffrey Xu, Yu Ge, Lu Hong-Jun, Sun Bing. A new path expression computing approach for XML data//Proceedings of 1st VLDB Workshop on Efficiency and Effectiveness of XML Tools and Techniques (LNCS 2590). Hong Kong, China, 2002: 35-46

[10] Al-Khalifa S, Jagadish H V, Koudas N, Patel J M, Srivastava D, Wu Yu-Qing. Structural joins: A primitive for efficient XML query pattern matching//Proceedings of the the 2002 International Conference on Data Engineering. San Jose, California, USA, 2002: 141-154

[11] Chien Shu-Yao, Vagena Z, Zhang Dong-Hui, Tsotras V J, Zaniolo C. Efficient structural joins on indexed XML documents//Bernstein P A, Ioannidis Y E, Ramakrishnan R, Papadias D eds. Proceedings of the 28th International Conference on Very Large Data Bases. Hong Kong, China, 2002: 263-274

[12] Jiang Hai-Feng, Lu Hong-Jun, Wang Wei, Ooi B C. XR-Tree: Indexing XML data for efficient stuctural join//Proceedings of the the 2003 International Conference on Data Engineering. 2003: 253-263

[13] Altmel M, Franklin M. Efficient filtering of XML documents for selective dissemination of information//Abbadi A E, Brodie M L, Chakravarthy S, Dayal U, Kamel N, Schlageter G, Whang Kyu-Young eds. Proceedings of the 26th International Conference on Very Large Data Bases. Cario, Egypt, 2000: 53-63

[14] Schmidt A, Waas F, Kersten M, Carey M J, Manolescu I, Busse R. XMark: A benchmark for XML data management//Bernstein P A, Ioannidis Y E, Ramakrishnan R, Papadias D eds. Proceedings of the 28th International Conference on Very Large Data Bases. Hong Kong, China, 2002: 974-985

[15] Bonifati A, Ceri S. Comparative analysis of five XML query languages. SIGMOD Record, 2000, 29(1): 68-79

[16] Cattell R, Barry D, Berler M et al. The Object Database Standard: ODMG 3.0. San Francisco, California: Morgan Kaufmann Publishers, Inc., 2000

[17] Ceri S, Comai S, Damiani E, Fraternali P, Paraboschi S, Tanca L. XML-GL: A graphical language for querying and restructuring XML documents. Computer Networks, 1999, 31(11-16): 1171-1187

[18] Chamberlin D, Robie J, Florescu D. Quilt: An XML query language for heterogeneous data sources//Proceedings of the 3rd International Workshop WebDB. Dallas, USA, 2000: 1-25

[19] Cooper B F, Sample N, Franklin M J, Hjaltason G R, Shadmon M. A fast index for semistructured data//Apers P M G, Atzeni P, Ceri S, Paraboschi S, Ramamohanarao K, Snodgrass R T eds. Proceedings of the 27th International Conference on Very Large Data Bases. Roma, Italy, 2001: 341-350

[20] Gottlob G, Koch C, Pichler R. Efficient algorithms for processing XPath queries//Bernstein P A, Ioannidis Y E, Ramakrishnan R, Papadias D eds. Proceedings of the 28th International Conference on Very Large Data Bases. Hong Kong, China, 2002: 95-106

[21] Milo T, Suciu D. Index structures for path expressions//Beeri C, Buneman P eds. Proceedings of the 7th International Conference Database Theory. Jerusalem, Israel, 1999: 277-295

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

YU Yong-Qian, born in 1966, Ph. D. candidate. His research interests include XML data management, data mining.

SUN Bing, born in 1978, Ph. D. candidate. His research interests include XML data management.

Background

Path expression is the core part of most XML query languages and a lot of methods have been proposed recently to solve the problem of computing path expressions. However, the existing methods are not efficient for computing complex path expressions, especially for computing long path expressions. To solve this problem, this paper proposes an efficient method SAM to compute XML path expression queries based on the automata technique. In order to compute the path expressions containing the "//" operation, this paper propose a schema automata method to rewrite such path expressions. The experimental results given in this paper that the proposed method is very efficient for computing long path expressions and complex path expression queries.