基于树自动机的 XPath 在 XML 数据流上的高效执行*

高 军+, 杨冬青, 唐世渭, 王腾蛟

(北京大学 信息科学技术学院,北京 100871)

Tree Automata Based Efficient XPath Evaluation over XML Data Stream

GAO Jun+, YANG Dong-Qing, TANG Shi-Wei, WANG Teng-Jiao

(School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

+ Corresponding author: Phn: +86-10-62765825, Fax: +86-10-62765822, E-mail: gaojun@db.pku.edu.cn, http://www.pku.edu.cn

Received 2003-08-27; Accepted 2004-05-08

Gao J, Yang DQ, Tang SW, Wang TJ. Tree automata based efficient XPath evaluation over XML data stream. *Journal of Software*, 2005,16(2):223−232. http://www.jos.org.cn/1000-9825/16/223.htm

Abstract: How to efficiently evaluate massive XPaths set over an XML stream is a fundamental problem in applications of the data stream. The current methods can not fully support the commonly used features of XPath, or can not meet the space and time requirement of the data stream applications. In this paper, a new tree automata based machine, XEBT, is proposed to solve the problem. Different from traditional ones, XEBT has the following features: First, it is based on tree automata with a powerful expressiveness, which can support Xpath {[]} without extra states or intermediate results; Second, XEBT supports many optimization strategies, including DTD based XPath tree automata construction, partial determination to reduce the concurrent states at running time with limited extra space costs, and the combination of bottom-up and top-down evaluation. Experimental results show that XEBT supports the complex Xpath and outperforms the former work in both efficiency and space cost.

Key words: XPath; tree automata; XML; data stream

摘 要: 如何在 XML 数据流上高效地执行大量的 XPath 查询成为数据流应用中一个迫切需要解决的关键问题.目前提出的算法或者不能完全支持 XPath 的常规特性,或者在算法的执行效率和空间代价上不能满足数据流应用的要求.提出了基于树自动机的 XEBT 机来解决这个问题.与传统方法相比,XEBT 机具备如下特征:首先,XEBT 机基于表达能力丰富的树自动机,无须附加中间状态,或保存中间结果,就能处理支持{[]}操作符的 XPath;其次,XEBT 机支持多种优化策略,包括基于 DTD 的 XPath 查询自动机的构造;在空间代价有限增加的情况下采用局部确定化减少并发执行的状态;采用自上而下和自下而上相结合的查询处理策略.实验结果表明,提出的方法能够支持复杂的 XPath 查询,在执行效率和空间代价方面优于传统算法.

* Supported by the National High-Tech Research and Development Plan of China under Grant No.2002AA4Z3440 (国家高技术研究发展计划(863)); the National Grand Fundamental Research 973 Program of China under Grant No.G1999032705 (国家重点基础研究发展规划(973))

作者简介: 高军(1975－),男,山东临沂人,博士,讲师,主要研究领域为数据库与信息系统;杨冬青(1945－),女,教授,博士生导师,主要研究领域为数据库与信息系统;唐世渭(1939－),男,教授,博士生导师,主要研究领域为数据库与信息系统;王腾蛟(1973－),男,博士,讲师,主要研究领域为数据库与信息系统.

关键词: XPath;树自动机;XML;数据流
中图法分类号: TP311 文献标识码: A

随着 XML 成为 Internet 环境中数据表示和交换的标准,出现了大量的 XML 数据流[1−6]相关的应用,如主动服务中的订购和发布系统、基于内容的 XML 路由、XML 文档分发等.与传统应用环境相比,数据流环境有很多特点:数据流环境中的查询是连续的;查询处理所使用的内存远远小于数据流本身;查询处理过程中数据仅仅能够被扫描一遍.

我们以主动服务为例来说明 XML 数据流之上海量查询处理的应用背景.Internet 环境中海量的用户利用 XPath 描述其需求,保存到系统中,这一阶段称为订购;当 XML 数据以网络速度流入时,系统判断 XML 数据流是否满足某个用户的需求,如果满足,则触发应用程序采取某种动作,如向用户返回相关的文档,这个阶段称为发布.如何在 XML 数据流上高效地执行大量 XPath 的查询成为主动服务实现的关键问题之一.衡量数据流之上不同查询处理方法的因素包括处理机所支持的 XPath 特性、处理机的执行效率和处理机所需要的空间代价.

1.相关工作

YFilter 根据 XPath 构造了 NFA 自动机[1],所有自动机合并为一个自动机.如果自动机在数据的驱动下到达某个终止状态,则数据流满足该终止状态所对应的查询.YFilter 能够减少不同查询处理中的重复计算,高效地处理不包含{[]}的 XPath,但是需要保存中间状态上的计算结果,来处理支持{[]}的 XPath 查询.

Xtrie[2]是对基于 NFA 自动机的查询处理的一种扩展,NFA 自动机每接受一个元素事件,都查找相关的转换,但是 XTrie 在接受元素事件序列之后,才选择相关的处理器进行响应.一般而言,响应元素序列的 XPath 处理机比响应单一元素的 XPath 处理机要少得多.通过这种方式,Xtrie 减少了接受元素输入序列的可能响应的查询处理器的数量,从而提高了运行时刻的处理效率.

上述方法本质上是非确定处理机,即在运行过程中,处理机内部存在若干活动状态,执行效率随着所支持 XPath 的增加而减少.而现有的另外一类 XML 数据流的查询处理,是基于确定自动机来构造的.

Dan Suciu[3]首先根据 XPath 构造 NFA 自动机,然后在 NFA 自动机之上执行确定化.这样,系统在运行的任意时刻,始终只有一个运行状态,系统的处理效率与所处理的 XPath 数目无关.然而,这种方法可能导致相对于查询数目指数级别的空间复杂性.另外,整个查询处理机表达能力有限,不支持 XPath 中的{[]}操作符.

XPush[4]解决了 NFA 自动机[3]的表达能力的问题.这一扩展主要利用了支持表达路径之间 AND/OR 关系的 AFA 自动机,AFA 自动机利用扩展的状态来保存不同路径的执行情况.XPush 执行器也是基于确定化自动机,提高了系统的查询效率,但是同样也面临着指数级别的确定化空间代价的问题.

2.本文的工作

尽管目前存在大量的相关研究,但是,XML 数据流之上海量 XPath 的查询处理在查询的表达能力、处理器的时间复杂性和空间复杂性方面依然很难满足应用需求.本文提出基于树自动机的 XEBT 机(XPath evaluation based on tree automata)高效地执行 XML 数据流之上大量复杂 XPath.本文的贡献在于:

1) 采用了树自动机作为 XEBT 的基础,树自动机不但能够自然表达 XPath{//,*,/,[]}查询,而且有利于利用 DTD 完成 XEBT 的进一步优化;

2) 提出了局部确定化的 XEBT 的优化策略,以有限的空间代价减少 XEBT 中同时运行的状态,从而提高 XEBT 的运行效率;

3) 提出了自上而下和自下而上相结合的 XEBT 的优化策略,在查询满足判定的代价不增加的情况下,减少查询集合中的重复操作;

4) 通过实验结果表明,XEBT 在空间复杂性和时间复杂性方面要优于现有算法.

本文第 1 节介绍相关的背景知识,包括 XPath 的描述、XML 的 SAX 解析事件和树自动机的概述.第 2 节提出基本的 XEBT 机.第 3 节提出基本 XEBT 机的各种优化策略,包括基于 DTD 的 XPath 查询执行器的构造、局部确定化、自上而下和自下而上相结合的执行策略等.第 4 节通过实验数据证明 XEBT 机的有效性.第 5 节总结全文并展望将来的工作.

1 背 景

1.1 XPath

XML 可以看作是点标记的有向树.XPath 是 XML 树中导航查询的基本机制[7].XPath 支持丰富的路径查询特性.非形式化地,我们给出 XPath 中常见的操作符号语义:‘/’表示数据节点之间的父子关系,‘//’表示节点之间的祖孙关系,‘[]’表示路径之间的条件关系,‘@’表示 XML 元素的属性,‘*’表示任意的数据元素,另外支持在路径表达式中定义逻辑表达式,包括‘<|≤|>|≥|=|≠’.

1.2 SAX

XML 数据流中的 XML 文档解析利用 SAX 解析器完成.SAX 解析器的输出不是一个文档树,而是前序扫描 XML 文档树时所触发的事件,包括 startDocument(),startElement(),text(),endElement(),endDocument()等.

例 1:一个 XML 数据流片断:

〈book〉〈title〉Database Principle〈/title〉〈author〉Ullman〈/author〉〈/book〉

SAX 解析器处理例 1 中文档的结果如下:StarteElement(book),StartElement(title),Text(“Database Principle”),endElement(title),StartElement(author),Text(“Ullman”),endElement(author),endElement(book),… 本文研究 XML 数据流之上的 XPath 集合的高效处理,其输入实际上是 XML 的 SAX 解析事件流.

1.3 树自动机

树自动机[8]是传统字符自动机的扩展.在字符自动机中,状态转移前后最多只能存在一个状态,而树自动机中的状态转换前或转换后支持状态集合,这种特性使得树自动机能够更加自然地处理树状数据结构.人们希望利用树自动机来解释 XML 查询的本质问题[9].

树自动机支持两种执行策略,自上而下和自下而上.以自下而上树自动机为例,我们给出树自动机的定义.树自动机通过 4 元组 $T=(Q,\Sigma,\delta,F)$来定义,其中,Σ是字母集合,Q 是有限状态的集合,$F\subseteq Q$ 是终止状态集合,状态转移函数δ用 $2^{Q^*}\times\Sigma\to Q$ 来表示,也可以通过 $f(q_1,\ldots,q_n)-a\to q$ 来表示,其中 f 表示状态的正则表达式,$q\in Q,q_1\in Q,\ldots,q_n\in Q,a\in\Sigma$.

树自动机表达能力强,能够表达 XML 的结构约束 DTD 和 XPath 查询.以一个简单的例子说明 XPath 对应的树自动机,给定 XPath 表达式 $//a[b=1][c>2]$,我们构造树自动机 (Q,Σ,δ,F),其中 $Q=\{s_1,s_2,s_3\},\Sigma=\{a,b,c\}$,$\delta=\{\{\}-b\to s_5,\{\}-c\to s_3,s_3s_5-a\to s_2,s_2-all\to s_1,s_1-all\to s_1\},F=\{s_1\}$.这个自动机接受所有包含 XPath 所描述路径的 XML 文档.

2 基本 XEBT 机

首先,我们描述本文所要解决的问题:给定 XPath 查询集合 $P=\{p_1,\ldots,p_n\}$,在 XML 数据流 S 之上处理 P 集合,返回矢量$\{a_1,\ldots,a_n\}$,其中 $a_i(1\le i\le n)$的取值范围是 0 或 1,表明数据流之上执行 XPath 查询 p_i 是否成功.XPath 支持的特性已在第 1.1 节中有所描述.

本文提出 XEBT 机完成 XML 数据流之上的 XPath 查询集合的查询处理,本节介绍基本 XEBT 机,在第 3 节中介绍优化后的 XEBT 机.

定义 1(基本 XEBT 机). 基本 XEBT 机 $E=(Q,\Sigma_{endElement},\delta,F,S)$,其中,$Q$ 表示 E 的状态集合,$\Sigma_{endElement}$ 表示输入字母集合,δ表示状态转换函数集合,$F\subseteq Q$ 表示状态的终止集合,S 表示状态和字母堆栈.

上述定义仅仅给出了 XEBT 机的静态描述,我们通过数据输入事件规则来描述 XEBT 机的动态特性.

根据前面讨论的结果,XEBT 机的数据输入是 XML 的 SAX 解析事件流,包括 startDocument,endDocument,startElement,endElement 和 Text 事件,我们给出 XEBT 在每一类事件触发时的响应动作:

```
startDocument:add an empty layer to stack T;     //在堆栈 T 中增加一个空层
StartElement(element)                            //元素 element 是当前元素
    pushElement(T, element);                     //将元素 element 压入堆栈 T
```

```
Text (value)                                  //value 是当前元素的值
    Add value into the top level of stack T;  //在堆栈的当前层增加元素的值
EndElement(element)
    Pop states set S from Stack T;            //从堆栈 T 中弹出当前层,将当前层的状态集合置于 S
    s=f((s1,…,sn), element),其中(s1,…,sn)∈S;  //检查堆栈最上层状态中是否存在转换规则,在当前元素
                                                  element 的输入下,发生状态转换
    Add s into top level of stack T;          //将转换后的状态合并到堆栈的当前层
EndDocument
    Pop up the states set from top layer      //弹出堆栈的当前层中的状态集合
    Decide which XPath is satisfied           //决定哪些 XPath 是满足的
```

2.1 基本XEBT的构造

给定 XPath,我们获取 XPath 的树模式[10].通过树模式构造树自动机是简单、直接的.下文中,我们给出构造的实例.现有方法中,与本文方法最相近的是基于 AFA 的 XPush[4].针对给定的 XPath,我们构造了 AFA 自动机和树自动机,以用来说明两者构造方法的不同.

例 2:给定 XPath 查询为,/person[(name or sex) and age]

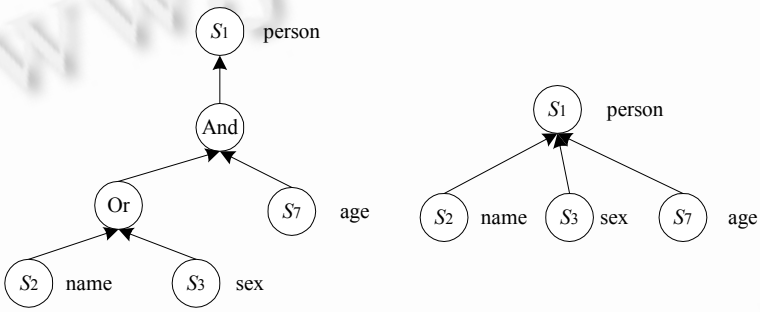

Fig.1 AFA (left) and tree automata (right)

图 1 AFA(左)自动机和树自动机(右)的状态比较

如果查询执行器基于 AFA 自动机来构造,则在 AFA 自动机中通过附加 AND 状态节点和 OR 状态节点支持状态的 AND/OR 关系.在图 1 中,OR 节点描述 name 和 sex 之间的关系,AND 节点描述 age 和 OR 节点之间的关系.这样,利用 AFA 表达给定的 XPath 查询需要 6 个状态.

如果查询执行器基于树自动机来构造,则我们可以利用树自动机中的状态表达式来描述 person 节点的子节点之间的关系.在图 1 中,涉及到 person 的状态转换表达式可描述为(s_2 or s_3) and s_7–person→s_1.状态转换表达式的含义是:如果树自动机的活动状态满足(s_2 or s_3) and s_7,并且数据的输入是 person,则树自动机的状态转换为 s_1.利用树自动机表达给定查询需要 4 个状态.

从上述的例子中可以看出,利用树自动机来表达 XPath 查询,所占用的空间比 AFA 自动机要少.由于 Internet 环境中查询数目是海量的,减少处理所需要的空间代价对于查询处理器非常重要,这是 XEBT 基于树自动机的一个重要原因.另外,从下文中可以看到,如果采用树自动机作为基础,则能够完备地利用 DTD 中所蕴含的结构约束来优化查询执行器.

2.2 基本XEBT的运行示例

上文中主要给出了基本 XEBT 机的构造方法,主要的思想是构造 XPath 对应的树自动机,自动机的输入就是 XML 文档 SAX 解析事件,按照 XEBT 的动态响应规则,处理 XML 数据流中的元素解析事件.如果自动机运行到达终止状态,则 XML 数据流满足特定的 XPath.下文将以例 2 中的例子说明 XEBT 运行的过程.

例 3:假定输入 XML 文档〈person〉〈age〉24〈/age〉〈name〉smith〈/name〉〈/person〉,则根据 XEBT 的动态响应规

则,图 1(b)中树自动机的运行过程如图 2 所示.

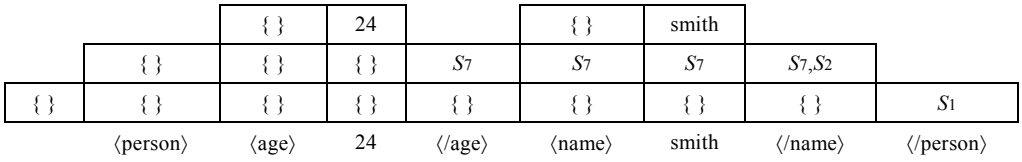

Fig.2 The running stack in XEBT on XML data stream

图 2 XML 数据流之上 XEBT 机堆栈变化

根据前面所定义的基本 XEBT 机的动态规则,堆栈在 StartDocument 事件触发时,压入一个空状态集合;当 startElement 事件触发时,系统将当前元素压入堆栈;当 endElement 事件触发时,系统弹出压入的元素,检查是否存在转换规则,如果存在,则调用转换规则,将新生成的状态加入堆栈的当前层中.举例来讲,我们讨论在 person 的 endElement 事件的时候,由于堆栈的最顶层保存了状态 s_7 和 s_2,则根据状态转换规则,(s_2 or s_3) and s_7,转换规则成立,所以,状态转换为 s_1,压入堆栈的当前层.在 endDocument 事件触发时,自动机的当前堆栈保存了状态 s_1,而 s_1 是树自动机的终止状态,即自动机接受 XML 数据的输入.

3 优化 XEBT 机

基本的 XEBT 机能够支持 XML 数据流之上 XPath 查询集合执行,并且支持 XPath 常见的特性.但是,在基本 XEBT 中,每个查询对应一个树自动机,这样就不能避免多 XPath 查询处理之间的重复计算.为提高基本 XEBT 的查询执行效率,同时不会导致指数的空间代价,本文引入了基本 XEBT 的 3 种优化策略:基于 DTD 的 XPath 树自动机的构造;XEBT 的局部确定化;自上而下和自下而上结合的执行策略.

3.1 基于DTD的XEBT的优化

基于 DTD 的 XEBT 的优化基本思想是:基于 DTD 重写给定的 XPath,从而减少状态转换过程中的状态转换圈和接受所有元素的状态转换规则.这种自动机中存在的非确定的因素,是自动机确定化操作产生大量新的状态和新的状态转移规则的一个重要原因.例如,如果我们能够根据 DTD 的约束信息,将 XPath 查询//p 等价转换为/p,则转换后 XPath 所对应的树自动机中包含更少的状态,并且不包含状态转换圈.

由于树自动机可以同时表达 DTD 和 XPath,则我们可以分别构造 DTD 树自动机和 XPath 树自动机,完成 DTD 树自动机和 XPath 树自动机的乘积操作,则得到的自动机乘积的结果就是 XPath 在 DTD 下的优化形式[11].

3.2 基本XEBT的局部确定化

基本的 XEBT 机相当于并行执行所有的 XPath 形成的树自动机,没有考虑 XPath 集合内部的重复计算.如果 XEBT 支持 n 条 XPath 查询语句,则基本 XEBT 机中存在 n 个非确定的树自动机,在基本 XEBT 的运行阶段,至少存在 n 个活动状态,即 XEBT 执行过程中至少需要判定 n 个状态转换,这导致 XEBT 的效率随着所支持的 XPath 数量的增加而急剧降低.

树自动机的确定化,解决同一个状态在数据输入的作用下不同状态输出的问题.确定化操作能够减少树自动机中并发运行的状态个数,提高 XEBT 机的执行效率.然而,完全的树自动机确定化操作将导致相对于 XPath 查询数目指数级别的状态空间,在数据流环境中是不可接受的.

为解决 XEBT 的优化阶段的空间复杂性和执行阶段的时间复杂性之间的矛盾,我们提出了树自动机的局部确定化操作.所谓局部确定化,就是按照一定的规则,选择部分的状态加以合并,在有限的空间增长的情况下,尽可能减少运行过程中并发的活动状态.

执行局部确定化操作,首先要确保执行局部确定化操作的树自动机和确定化前的自动机等价.给定两个树自动机 T_1 和 T_2,如果对于 T_1 接受的任意树 K_1,都能被 T_2 接受,反之亦然,则两个自动机是等价的.其次,需要获取哪些状态是可以合并的.在自动机保持等价的情况下,合并状态可能导致新的状态和新的转换规则的产生.根据状态合并后自动机进行调整的代价,我们选择部分状态加以合并.

在合并过程中,我们首先处理不参与任何状态转换圈的状态.不失一般性,假定存在两条状态转换规则,其初始的状态函数 f 一致,在数据输入 d 的作用下,转换到状态 s_1 和 s_2,并且,状态 s_1 和状态 s_2 没有参与任何的状态转换圈,如果我们将 s_1 和 s_2 的状态合并,则不会产生指数的状态和状态转换规则.

另外,假定存在两条状态转换规则,其初始的状态函数 f 一致,在同样数据输入 d 的作用下,转换到状态 s_1 和 s_2.其中,状态 s_1 在字母 k 的输入下,只能转换到状态 s_1,状态 s_2 在字母 k 的输入下,只能转换到状态 s_2,如果我们合并状态 s_1 和状态 s_1,则引入的新的状态和状态转换是可控的,我们认为这一类状态也是可以合并的.

在局部确定化的过程中,我们避免了参与状态转换圈的状态的合并操作.我们分析合并上述状态可能带来的代价.在树自动机 T 中,假定存在状态转换序列 S,使得状态 a 经过一次以上的转换之后回到状态 a,并且存在状态函数 f,接受数据 d 输入,得到状态 a 和状态 c,如果我们将状态 a 和状态 c 合并为一个新的状态 s_1,则由于状态 a 参与状态转换圈,新的状态 s_1 必然能够转换到状态 a.这种包含参与状态转换圈的状态的合并方式,将产生大量新的状态和新的转换,同时新的自动机中依然保留原有的状态和相关的状态转移规则.针对不同特性的树自动机的状态,我们给出了可一次合并状态的概念.

定义 2(可一次合并状态). 给定自动机 T 中的状态 a,如果在 T 中不存在一个转换序列 $S=a,a_1,\ldots,a$,使得状态 a 经过一次以上的状态转换,转换到状态 a,则状态 a 称为可一次合并状态.

在数据流环境中,我们利用树自动机来表示 XPath.路径操作符号{/,*,[]}在对应的树自动机中不会引入状态转换圈,而{//}操作符号可能引入了一个状态 a,状态 a 经过数据输入,转换到状态 a 自身.上述两种情况都满足可一次合并的状态的条件.另外,我们在第 3.1 节中讨论的 DTD 优化,主要减少 XPath 中的{*,//}操作符号,在非递归的 DTD 的作用下,消除 XPath 树自动机中的状态转换圈[11].这种优化,实际上能够提高局部确定化过程中状态合并的可行性.

我们给出可一次合并状态的合并规则.

合并规则 1. 假定一个状态函数 $f(e_1,\ldots,e_j)$,在数据 d 的作用下,转换到状态集合 $T=\{s_1,\ldots,s_n\}$,则状态集合 T 的合并规则如下:(1) 构造状态集合 $A=\{a_1,\ldots,a_m\}$,$A\subseteq T$,使得 $a\in A$ 不参与任何状态转换圈;构造状态集合 $B=\{b_1,\ldots,b_k\}$,$B\subseteq T$,使得 $b\in B$ 参与转换到自身的状态转换圈.构造新的状态,使得 $t_1=(a_1,\ldots,a_m)$,$t_2=(b_1,\ldots,b_k)$;(2) 假定树自动机中存在状态转换规则,$f(e_1,\ldots,e_h,\ldots,e_j)-d_1\to g$,如果 $e_h=a_i(1\le h\le j,1\le i\le m,a_i\in A)$,则构造新的状态转换规则 $f(e_1,\ldots,t_1,\ldots,e_j)-d_1\to g$;如果 $e_h=b_i(1\le h\le j,1\le j\le k,b_i\in B)$,则构造新的状态转换规则 $f(e_1,\ldots,t_2,\ldots,e_j)-d_1\to g$;(3) 删除原有的状态转换规则:删除原有 $f(e_1,\ldots,e_j)-d\to a$,其中,$a\in A$,增加状态转移规则 $f(e_1,\ldots,e_j)-d\to t_1$;删除原有 $f(e_1,\ldots,e_j)-d\to b$,其中,$b\in B$,增加状态转移规则 $f(e_1,\ldots,e_j)-d\to t_2$;(4) 删除原有状态:如果不存在状态转换规则,转换后的状态是 $a\in A$,则删除状态 a;如果不存在状态转换规则,转换后的状态是 $b\in B$,则删除状态 b.

我们实现局部确定化的过程,实际上就是不断获取新的等价树自动机的过程,其中每一个新产生的等价自动机都会减少原有树自动机的非确定的状态转换规则.状态合并的代价是产生了新的状态转移规则和状态,增加了查询处理器的空间代价.下面,我们给出局部确定化正确性的证明.

定理 1. 树自动机 T_1 按照转换规则转换 1,转换 T_1 中的状态转换规则集合 $f(e_1,\ldots,e_j)-a\to s_1,\ldots,f(e_1,\ldots,e_j)-a\to s_n$,得到新的树自动机 T_2,则 T_1 和 T_2 是等价的.

对于 T_1 所接受的语言 L,如果在自动机 T_1 接受 L 的状态转换序列 S_1 中不涉及到 $f(e_1,\ldots,e_j)-a\to s_1,\ldots,f(e_1,\ldots,e_j)-a\to s_n$ 的状态转换,则 T_2 自然接受语言 L.如果涉及到 $f(e_1,\ldots,e_j)-a\to s_1,\ldots f(e_1,\ldots,e_j)-a\to s_n$ 之间的状态转换,则假定状态转换过程中,包含 $f(e_1,\ldots,e_j)-a\to s_1,f_1(s_1)-b\to p$,则根据合并规则 1,则在自动机 T_2 的状态转换规则中,必然存在状态转换序列 $f(e_1,\ldots,e_j)-a\to t_1,f_1(t_1)-b\to p$.我们将 S_1 中的 $f(e_1,\ldots,e_j)-a\to s_1,f_1(s_1)-b\to p$,替换为 $f(e_1,\ldots,e_j)-a\to t_1,f_1(t_1)-b\to p$,得到状态转换序列 S_2,则容易证明,T_2 执行状态序列 S_2 接受语言 L.

对于 T_2 所接受的任何语言 L,如果在自动机接受的状态转换序列 S_2 中不涉及到 $f(e_1,\ldots,e_j)-a\to t_1$ 的状态转换,则 T_1 自然接受语言 L.如果涉及到 $f(e_1,\ldots,e_j)-a\to t_1$ 之间的状态转换,则假定状态转换过程中,包含 $f(e_1,\ldots,e_j)-a\to t_1,f_1(t_1)-b\to p$,则根据合并规则 1,在自动机 T_1 中,必然存在状态转换序列 $f(e_1,\ldots,e_j)-a\to s_1,f_1(s_1)-b\to p$.如果将 S_2 中的 $f(e_1,\ldots,e_j)-a\to t_1,f_1(t_1)-b\to p$,替换为 $f(e_1,\ldots,e_j)-a\to s_1,f_1(s_1)-b\to p$,得到

状态转换序列 S_1,则容易证明,T_1 执行状态序列 S_1 接受语言 L.

所以,T_1 和 T_2 自动机是等价的. □

例 4:给出 3 个 XPath p_1=/a/b[c][d],p_2=/a/b[c][e],p_3=/k/b[c][//e],构造 XPath 查询所对应的查询自动机如图 3 所示.

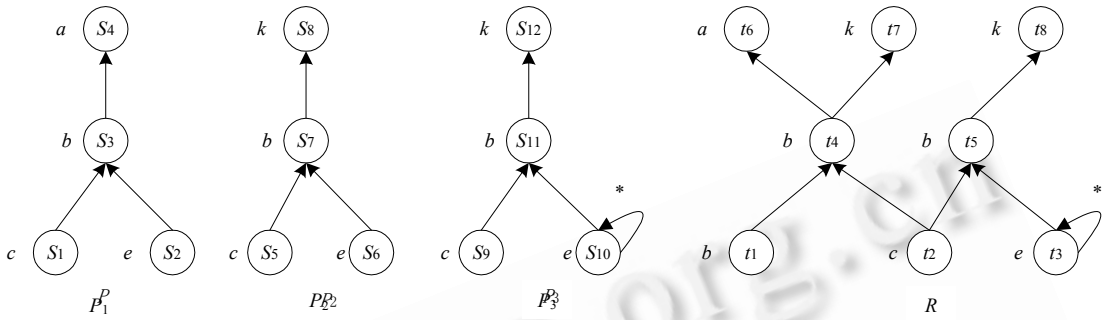

Fig.3 The sample of the partial determination

图 3 局部确定化的示例

按照算法 1,在 p_1,p_2,p_3 这 3 个查询所对应的自动机之上执行局部确定化的操作,得到 R 子图对应的局部确定化树自动机.举例来讲,树自动机中存在转换规则,{}−c→s_1,{}−c→s_5,{}−c→s_9,根据状态合并规则,我们产生了新的状态 t_4=(s_1,s_5)和 t_2=(s_9).易证合并后的树自动机和原有的树自动机等价.

根据上文中所描述的局部确定化的思想,我们给出树自动机局部确定化的算法.

算法 1. 局部确定化自动机.

输入:当前的树自动机 T_1

输出:局部确定化之后的树自动机 T_2

1) 扫描 T_1 中的状态转换序列,获取状态转换序列集合 P,P 集合中状态转换的初始状态和输入数据相同,但是转换后的状态不同.

2) 按照合并规则 1,处理状态转换序列集合 P.

3) 继续算法第 1 步,直到找不到可以合并的状态为止.

3.3 自上而下和自下而上结合

树自动机的确定化,本质是减少了不同查询之间的重复路径.XEBT 采用自下而上的树自动机作为基础,如果在自动机的起始状态,输入数据的条件稍有不同,则导致我们很难合并其他状态,即使这些查询中间存在大量的重复计算.举例来讲,给定 XPath 查询//b/c=1 和//b/c=3,这两个查询自上而下的路径表达式是一致的.但是如果按照自下而上的方法构造树自动机,由于算术表达式不同,两个树自动机的初始状态不能合并,所以,我们无法合并自根节点开始的重复路径.

根据 XPath,我们可以获取 XPath 查询的树模式.如果自根节点开始到出度>1 的节点之间,按照自上而下构造自动机,则自动机中有且仅有一个终止状态.我们在这一段中,按照自上而下的策略来执行,不会带来终止状态判定的复杂性的提高.下文中,我们形式化地给出分界点的概念,标记自根节点出发,所遇到的第 1 个度>1 的节点.

定义 3(分界点). 给定 XPath P,构造 XPath P 的树模式,自根节点 r 开始,如果存在节点 k,r 到 k 路径中的每个节点 t,t 的出度=1,而 k 的出度>1,则 k 为分界点.

如果我们把 XPath 看作树模式,则扩展 XEBT 的主要思想是,在根节点到分界点的过程中,XEBT 响应 startElement 事件,采用自上而下的执行策略;从叶结点到分界点的过程中,XEBT 响应 endElement 事件,执行自下而上的执行策略.如果在状态堆栈的某个层次中同时存在同一 XPath 所对应的自上而下树自动机的终止状态和自下而上的树自动机的终止状态,则数据流满足 XPath 查询输入.我们结合自上而下执行的 XEBT 和自下

而上执行的 XEBT,给出扩展的 XEBT 的定义.

定义 4(扩展 XEBT 机). 扩展的 XEBT 机可以定义为 9 元组 $T=(Q_{start},Q_{end},\Sigma_{startElement},\Sigma_{endElement},\delta_{start},\delta_{end},F_{start},F_{end},S)$, 其中 ,$\delta_{start}$ 表示 $f(q)-a\rightarrow(q_1,\ldots,q_n),\{q,q_1,\ldots,q_n\}\subseteq Q_{start},\delta_{end}$ 表示 $f(p_1,\ldots,p_n)-a\rightarrow p,\{p,p_1,\ldots,p_n\}\subseteq Q_{end}$, $F_{start}\subseteq Q_{start},F_{end}\subseteq Q_{end}$.

扩展 XEBT 是由自上而下的树自动机和自下而上的树自动机合并而成.我们通过描述事件响应动作,给出扩展 XEBT 的动态特性:

startDocument:add an empty layer to stack; //在堆栈中增加一个空层

StartElement(*element*) //元素 *element* 是当前元素

S=current States set in the top layer; //获取当前状态集合

Check δ_{start} to find $f(q)-element\rightarrow(q_1,\ldots,q_n),q\in S$; //检查自上而下的转换规则

If exist, $S_1=S_1\cup\{(q_1,\ldots,q_n)\}$; //如果存在,则获取转换后的状态

pushStack(S_1); //将转换后的状态压入堆栈

Text:

Add the current Text to current layer in stack; //将当前数据增加到堆栈的当前层

EndElement

S= pop states in current layer; //获取堆栈的当前状态集合

Check δ_{end} to find $f(q_1,\ldots,q_n)-element\rightarrow q,\{q_1,\ldots,q_n\}\subseteq S$; //检查自下而上的转换规则

If exist, $S_1=S_1\cup\{q\}$; //如果存在,则获取转换后的状态

add S_1 to current layer; //将转换后的状态追加到当前层

If exists $q_1\in F_{end}$, and $q_2\in F_{start}$, and q_1, q_2 belong to the same XPath, then output query into queue *Q*;
//检查当前状态中,是否表明某个 XPath 满足,如果是,则输出到队列中.

EndDocument

Based on the queue, decide which XPath is satisfied //根据队列,输出结果

与基本 XEBT 机不同,扩展 XEBT 机的终止状态,不一定在堆栈的第 1 层.我们在自下而上树自动机满足终止状态时,检查堆栈的当前层中是否存在同一个 XPath 所对应的自上而下的树自动机的终止状态.如果存在,则易证 XML 数据流满足 XPath 查询.

4 算法的性能分析

我们参照 XPush[4]构建了 XEBT 的实验环境,实验数据是 NASA 的 XML 数据集合[12],数据量是 25M.在数据流环境中,XML 数据是通过单遍扫描加以处理,在特定时刻只能同时处理一个元素事件,所以 XML 数据流所处理的数据量是随着时间的推移而增长,XML 文件的大小不是我们所关心的主要因素,我们主要测试 XML 数据流系统中并发查询集合的处理效率和空间代价.我们指定 XPath 操作符{*}的概率 *w*,{//}的概率 *d* 和{[]}的概率 *b*,利用 XPath 生成器中获得不同特性组合的 XPath 查询集合 5k,10k,15k,20k.

我们利用 Java 1.31 实现了 XEBT 机,同时参照文献[4]中的算法,实现了 XPush.本实验的硬件环境是:硬件系统为 Optiplex 商用计算机,操作系统为 Windows 2000,CPU 为 P4,主频为 1.4G,主存为 512M.下文中,B 表示自下而上执行,*T* 表示自上而下执行,PD 表示部分确定化,BTPD 表示自上而下和自下而上同时进行确定化的操作,BPD 表示自下而上进行确定化,TPD 表示自上而下进行确定化的操作,DTD 表示采用 DTD 进行优化.

图 4 完成了 XEBT 机的空间复杂性分析,如果是基本的 XEBT 机,则产生的状态与输入的 XPath 线性相关.在基本 XEBT 机上完成的局部化确定操作,导致了自动机状态有限增加.由于树自动机表达 XPath 更加自然,与 XPush 所基于的 AFA 自动机相比,树自动机不需要中间状态,所以利用 XEBT 生成的空间状态要少于 XPush 的状态.同样,我们分析了局部确定化在包含重复 XPath 的集合中的作用,如果 XPath 集合中存在重复,则局部确定化可以减少不同 XPath 之间的重复计算,减少所需要的状态数目.

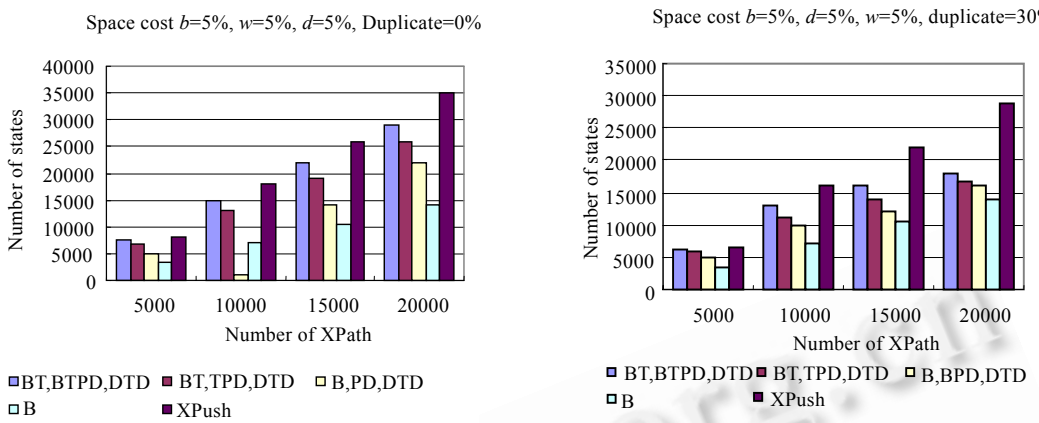

Fig.4 The space cost of the XEBT (Without duplicate XPath and with duplicate XPath)

图 4 空间复杂性分析(包含不重复 XPath 和包含重复 Xpath)

本文完成了不同特性的 XPath 在 XEBT 机上运行的时间复杂性.在图 5 中,我们测试了不同比率的{[]}的 XPath 在 XEBT 机上的执行效率.由于{[]}操作符增加了状态转换规则的判定的复杂性,则 XPath 存在更多{[]}将导致复杂性的增加.我们测试了 XPath 查询中'//'和'*'发生变化的情况下,XEBT 的查询执行效率.由于我们测试的两种 XEBT 机,都包含 DTD 乘积的策略.实验证明,在某种程度上,DTD 乘积能够减少 XPath 查询中的不确定的因素,从而使得支持 DTD 乘积的 XEBT 机对于'//'和'*'不是很敏感.

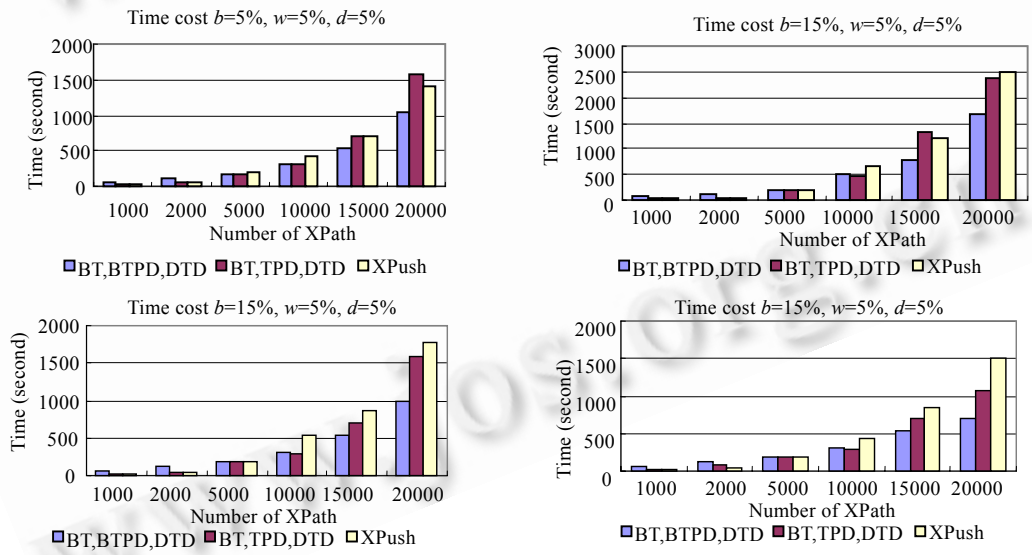

Fig.5 The time cost with various features of XPath

图 5 时间复杂性分析 XPath 中特性的变化

图 5 的结果表明,在并发查询数据较小的情况下,由于执行多种优化策略的 XEBT 机处理机制相对复杂,执行效率低于 XPush,但是在并发查询数据较大的情况下,完成优化策略的 XEBT 机的执行效率要高于 XPush.由于基本 XEBT 的动态执行效率随着所处理的并发查询数据的增加而急剧降低,所以,时间复杂性分析中没有对比基本 XEBT 机的效率.

5 总结和下一步工作

本文提出了基于树自动机的 XEBT 来解决支持{//,*,/,[]}的复杂 XPath 在 XML 数据流上高效处理的问题,

并且在基础的 XEBT 机之上探讨了多种优化策略.实验表明,我们在执行效率和空间复杂性方面要优于现有算法.今后的工作将讨论如何在 XML 数据流中支持 XQuery 查询.

References:

[1] Diao Y, Fischer P. YFilter: Efficient and scalable filtering of XML documents. In: Proc. of the 18th Int'l Conf. on Data Engineering. 2002. 341−345.

[2] Chan C, Felber P, Garofalakis M, Rastogi R. Efficient filtering of XML document with XPath expressions. In: Proc. of the Int'l Conf. on Data Engineering. San Jose: IEEE Computer Society, 2002. 235−244.

[3] Green TJ, Miklau G, Onizuka M, Suciu D. Processing XML streaming with deterministic automata. In: Calvanese D, Lenzerini M, Motwani R, eds. Proc. of the Int'l Conf. on Data Theory. LNCS 2572, Springer-Verlag, 2003. 173−189.

[4] Gupta AK, Suciu D. Stream processing of XPath queries with predicates. In: Halevy AY, Ives ZG, Doan AH, eds. Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data. ACM, 2003. 419−430.

[5] Nguyen B, Abiteboul S, Cobena G, Preda M. Monitoring XML data on the Web. In: Aref WG, ed. Proc. of the ACM/SIGMOD Conf. on Management of Data. 2001. 437−448.

[6] Chen J, Dewitt D, Tian F, Wang Y. NiagaraCQ: A scalable continuous query system for internet databases. In: Chen WD, Naughton JF, Bernstein PA, eds. Proc. of the ACM/SIGMOD Conf. Management of Data. ACM, 2000. 379−390.

[7] Clark J. XML Path language (XPath). 1999. Available from the W3C, http://www.w3.org/TR/XPath

[8] Neven F. Automata, logic, and XML. In: Proc. of the 16th Int'l Workshop Computer Science Logic. CSL, 2002. 2−26.

[9] Milo T, Suciu D, Vianu V. Typechecking for XML Transformers. In: Proc. of the PODS 2000. ACM, 2000. 11−22.

[10] Miklau G, Suciu D. Containment and equivalence for an XPath fragment. In: Popa L, ed. Proc. of the 21 Symp. on Principle of Database Systems. ACM, 2002. 65−76.

[11] Gao J, Yang DQ, Tang SW, Wang TJ. DTD based deterministic XPath rewriting and logical optimization. Journal of Software, 2004,15(12):1860−1868 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/15/1860.htm

[12] NASA's Astronomical Data Center. ADC XML Resource Page. http://xml.gsfc.nasa.gov

附中文参考文献:

[11] 高军,杨冬青,唐世渭,王腾蛟.一种基于 DTD 的 XPath 逻辑优化方法.软件学报,2004,15(12):1860−1868. http://www.jos.org.cn/1000-9825/15/1860.htm