面向 XPath 执行的 XML 数据流压缩方法*

王腾蛟+, 高 军, 杨冬青, 唐世渭, 刘云峰

(北京大学 信息科学技术学院,北京 100871)

XPath Evaluation Oriented XML Data Stream Compression

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

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

+ Corresponding author: Phn: +86-10-62765823, E-mail: tjwang@pku.edu.cn, http://www.pku.edu.cn

Received 2003-10-13; Accepted 2004-03-02

Wang TJ, Gao J, Yang DQ, Tang SW, Liu YF. XPath evaluation oriented XML data stream compression. *Journal of Software*, 2005,16(5):869−877. DOI: 10.1360/jos160869

Abstract: Because XML (extensible markup language) is self-described, there is much redundant structural information in XML data stream. How to compress XML data so as to reduce the network transfer cost and support XPath evaluation on the compressed data is a new area of research. The existing methods on XML compression require the multi-pass scan on data or can not support real time query processing on compressed data. In this paper, a novel compression method XSC (XML stream compression) is proposed to compress and decompress XML stream in real time. XSC constructs XML element event sequence dictionary and outputs the related index dynamically. When DTD is available, XSC can generate the XML element event sequence graph for producing more reasonable encoding before XML data stream is processed. The compressed XML data stream can be decomposed directly for XPath evaluation. Experimental results show that XSC outperforms other methods in compression ratio and compression efficiency, and the cost of XPath evaluation on compressed data stream is acceptable.

Key words: XML (extensible markup language); data stream; compression; DTD; XPath

摘 要: 由于 XML(extensible markup language)本身是自描述的,所以 XML 数据流中存在大量冗余的结构信息.如何压缩 XML 数据流,使得在减少网络传输代价的同时有效支持压缩数据流上的查询处理,成为一个新的研究领域.目前已有的 XML 数据压缩技术,都需要扫描数据多遍,或者不支持数据流之上的实时查询处理.提出了一种 XML 数据流的压缩技术 XSC(XML stream compression),实时完成 XML 数据流的压缩和解压缩,XSC 动态构建 XML 元素事件序列字典并输出相关索引,能够根据 XML 数据流所遵从的 DTD,产生 XML 元素事件序列图,在压缩扫描之前,产生更加合理的结构序列编码.压缩的 XML 数据流能够直接解压缩用于 XPath 的执行.实验

* 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: **王腾蛟**(1973−),男,山东济南人,博士,副教授,主要研究领域为数据库与信息系统;**高军**(1975−),男,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;**刘云峰**(1973−),女,博士生,主要研究领域为数据库与信息系统.

表明,在 XML 数据流环境中,XSC 在数据压缩率和压缩时间上要优于传统算法.同时,在压缩数据之上查询的执行代价是可以接受的.

关键词: XML;数据流;压缩;DTD;XPath
中图法分类号: TP393 文献标识码: A

XML(extensible markup language)已成为 Internet 上数据表示和交换的事实标准,越来越多的应用系统采用 XML 作为标准来交换数据.在许多涉及海量数据交换的重要应用系统中,XML 数据在网络中以流的形式存在.由于 XML 文档除了数据本身,还包括大量结构描述信息,因此 XML 数据流的传输代价非常高,如何有效地完成 XML 数据流的压缩,同时支持对 XML 数据流的查询处理,成为一个迫切需要解决的问题.

(1) 相关工作

针对 XML 中的大量冗余的结构信息,目前出现了多种 XML 数据压缩的算法,这些算法的主要目的是减少静态 XML 文档的存储空间代价和传输代价.其中,典型的系统包括:

XMill 系统[1].Dan Suciu 在 XMill 系统提出了 XML 数据的压缩算法.XMill 系统的主要思想是将 XML 文档中的每一种元素分别存放并压缩.但是,Xmill 系统以整个 XML 文档作为压缩粒度,如果希望查询数据,则只能解压缩全部数据;同样,我们在压缩数据之上,不能执行类似创建视图等有意义的操作.

XGrind 系统[2].XGrind 系统的最大特点是在压缩数据之上支持查询.XGrind 系统的压缩粒度是 XML 数据中的元素/属性,压缩数据中保持原有 XML 数据的结构;在查询过程中,支持部分解压缩数据回答查询.但是,利用 XGrind 系统压缩数据需要扫描 XML 数据两遍,第 1 遍获取 XML 元素的频率统计并构造字典表,第 2 遍扫描完成数据压缩.

XPress 系统[3].XPress 系统采用一种数字编码方式压缩 XML 数据,这种数字编码方式能够方便地捕捉 XML 的文档结构.与 XGrind 系统一样,XPress 系统支持压缩数据之上的查询,系统支持局部解压缩回答用户查询.但是,XPress 系统同样需要扫描数据两遍,不能满足数据流环境中的要求.

(2) 本文的工作

针对数据流环境中数据处理的要求,本文提出了一种 XML 数据流的压缩算法 XSC(XML stream compression)的 XML 数据流的压缩算法.本文的贡献如下:

XSC 算法中压缩和解压缩 XML 数据均是一遍扫描完成,同时,XSC 算法不需要保存和传递压缩过程中产生的压缩编码字典,减少了数据流中的传输代价.

XSC 算法支持压缩数据上的 XPath 查询处理.XSC 算法是一种同构压缩算法,XML 数据流在数据源端经过 XSC 算法压缩,在数据目标端经过 XSC 算法还原为原有序列,XML 数据流之上的数据压缩不影响原有 XML 数据流之上的查询处理.

XSC 算法可以有效利用 DTD(document type definition)来提高 XML 数据流的压缩比率和压缩效率.XSC 首先分析 DTD,构造元素事件序列图,找到所有可能的事件序列,并且估算出事件序列的出现频率,根据事件序列的出现频率,产生 Huffman 编码,获取更高的压缩比率.

本文第 1 节给出 XSC 的基本算法,来压缩和解压缩 XML 的 SAX 事件流.第 2 节给出 XSC 的优化算法,基于 Huffman 编码的思想,利用 DTD 获取频繁出现的元素事件序列,提高了基本的 XSC 压缩算法的压缩比率和运行效率.第 3 节给出实验数据,证明本文提出方法的有效性.第 4 节总结全文.

1 基本 XSC:面向 XPath 处理的 XML 数据流压缩算法

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 数据流的压缩,实际上是 SAX 解析事件流的压缩.

1.1 问题定义

给定 XML 数据流,在数据源,利用 SAX 解析器将数据流解析成为 XML 事件输出流 L,XSC 完成 XML 事件流的压缩,获得 $XSC(L)$;网络传输 $XSC(L)$;在 XML 数据流的处理阶段,XSC 解压缩 $XSC(L)$,解压缩后的结果输入到原有 XML 数据流的查询处理器中.

不同于静态 XML 文档的压缩,XML 数据流压缩带有自身的特点:首先在 XML 数据流的压缩只能扫描数据流一遍;其次,静态 XML 文档压缩需要保持节点之间的父子关系,而 XML 数据流的压缩需要保持元素的前序扫描顺序.

1.2 基于Lempel-Ziv编码的XML数据流压缩算法

XML 数据流中记录了 XML 元素的数据和结构.在 SAX 解析器的输出流中,结构主要是通过 StartElement 和 EndElement 事件序列来体现,数据主要以 Text 的事件来体现.XPress[3]系统的研究结果表明,XML 数据流中数据部分采用传统的基于数据类型的压缩算法效果良好.在 XML 数据流中,结构信息大量冗余,所以本文中方法的创新之处侧重于 XML 数据流中结构部分的压缩.

本文提出的基本 XSC 的压缩算法借鉴了 Lempel-Ziv[4]编码的思想,其核心是在压缩阶段,针对 XML 数据流的结构部分,利用字典表记录出现的事件序列,输出事件序列所对应的编号;在解压缩阶段,动态恢复压缩过程中产生的元素事件序列,构造字典表.对于 XML 数据流中数据部分,采用了数据类型相关的压缩算法[1−3].如果数据是数值类型,则数据压缩采用增量压缩算法;如果是字符类型,则采用字典压缩算法.

算法.基本 XSC 压缩算法.

输入:XML 元素的 SAX 解析器的事件序列.

输出:XML 元素的 SAX 解析器的压缩序列.

```
currentSymbol='';                                  //初始化 currentSymbol 为空
Init DICT with element in XML;                     //根据 XML 中的元素,初始化字典 DICT
OnElement                                          //SAX 解析的元素事件响应处理
currentSymbol=currentSymbol+currentElement;        //根据当前的元素,构建新的 currentSymbol
  Select the currentSymbol in DICT;
  IF not selected then
     add the currentSymbol to DICT;                //是字典中新出现的,追加到字典中
     Select the index of CurrentSymbol-CurrentElement;  //输出上一个序列在字典中的序号
     Output the index;
     currentSymbol=currentElement;
  EndIF
OnText                                             //SAX 解析的值域事件响应处理
  TypeReference(Text);                             //获取当前元素的类型
  If character, using the dictionary to compress data;  //利用字典压缩算法,压缩字符数据
  If numeric, using the delta to compress data;    //利用 delta 增量,压缩数值数据
```

基本 XSC 的解压缩算法,需要根据压缩的数据,动态构造压缩过程中产生的编码字典表.下面,我们给出 XSC 解压缩算法的框架.

算法.基本 XSC 解压缩算法.

输入:压缩后的 XML 数据流的 SAX 序列.

输出:解压缩之后的 XML 数据流的 SAX 序列.

```
Init DICT with element in XML;                          //根据 XML 中的元素,初始化字典 DICT
Do while not end of compressed stream
    CurrentSymbol=readSymbol;
    If currentSymbol is Element
        NextSymbol=readSymbol.value;                    //value 是记录在字典中的编码所对应的序列
        NewSymbol=currentSymbol.value+firstElement of NextSymbol.value;
                                                        //根据压缩数据,动态构造字典
        Add the NewSymbol into DICT;
        Output the Value of currentSymbol.value;        //输出当前符号对应的事件序列
        CurrentSymbol=NextSymbol;
    If currentSymbol is data
        Decompress and output the currentSymbol based on Data Type;
End do
```

值得注意的是,与 Lempel-Ziv 编码一致,在解压缩过程中,算法的输入不包括结构序列所对应的编码表.但是,基本 XSC 和单纯的 Lempel-Ziv 编码也存在不同.基本 XSC 算法基于元素/属性粒度;单纯 Lempel-ziv 算法基于字符粒度,需要扫描 XML 数据元素多次才能将 XML 数据元素看做是一个字符序列.由于加入了 XML 数据的语义信息,基于 XSC 算法的压缩效果比传统的 Lempel-Ziv 编码效果要好.

我们给出一个 XML 示例,说明基本 XSC 的压缩过程.

例 2:给定 XML 文档,⟨class⟩⟨name⟩zhang⟨/name⟩⟨name⟩li⟨/name⟩⟨name⟩zhao⟨/name⟩⟨/class⟩,利用基本 XSC 压缩算法后的事件序列编码.见表 1.

Table 1 The sample dictionary in XML compression

表 1 压缩过程中产生的字典表

⟨class⟩	1
⟨name⟩	2
⟨/name⟩	3
⟨/class⟩	4
⟨class⟩⟨name⟩	5
⟨/name⟩⟨name⟩	6
⟨/name⟩⟨/class⟩	7

字典表初始为 XML 文档中的元素,序号从 1~4,剩余序列是动态增加的.结构序列压缩的输出序号是:1,2,3,2,6,3,4.

我们在图 1 中,给出了数据流环境中的压缩和解压缩的框架.

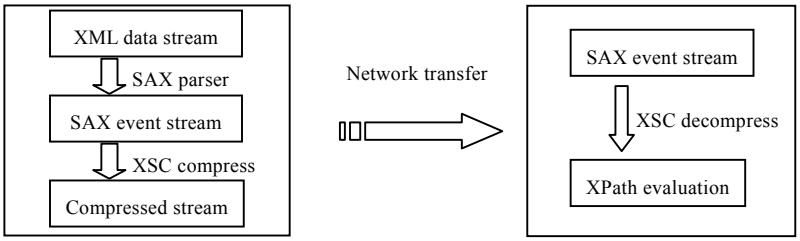

Fig.1 The architecture of XML data stream compression

图 1 XML 数据流的压缩体系结构

2 优化算法

基本 XSC 算法能够满足数据流环境的要求,完成 XML 数据流的压缩.但是,在基本 XSC 算法中,事件序列

是在 XSC 算法运行过程中动态产生的,这种方式增加了压缩过程中的处理代价;而且,在基本 XSC 算法的 XML 事件序列中,没有考虑元素事件序列出现的频率,而仅仅依赖于元素事件序列在 XML 文档的位置.这样,最后产生的事件序列编码可能是不合理的.

目前的 XML 一般遵循 DTD 结构约束.XSC 的优化算法的基本思想是:在数据流处理之前,分析 DTD,获取可能的事件序列,估算事件序列的频率,产生相应的编码,从而减少动态压缩过程中的代价,提高压缩比率.

2.1 基于DTD来构造元素事件序列的Huffman编码

我们约定,如果 DTD 定义中的一个元素 e 可以通过其他元素来定义,则元素 e 称为复杂元素,否则,元素 e 称为原子元素.我们首先根据 DTD 定义构造 DTD 元素事件图.

定义 1(DTD 元素事件图). DTD 元素事件图 $G=(N,E,W,Y)$,其中,N 表示节点集合,Y 表示 DTD 中所有出现的元素名称,边集合 $E=Estart\cup Eend\cup Eempty$,$Estart$ 表示元素 $e\in Y$ 的 startElement 事件,$Eend$ 表示元素 $e\in Y$ 的 endElement 事件,$Eempty$ 表示空事件;W 是 E 集合到自然数的映射,表明边的权重.给定任何元素 e,我们限定元素 e 的 $Estart$ 边的权重和 $Eend$ 边的权重是相等的.

定义 DTD 元素事件图的目的,一方面是希望利用 DTD 元素事件序列图来找到可能的元素事件序列,另一方面是希望利用 DTD 元素事件图中边的权重,来获取元素事件序列发生的频率.

在根据 DTD 来构造 DTD 元素事件图的过程中,我们假定在 DTD 定义中,一个元素定义存在并且仅存在一次.如果同一元素存在多个定义,则可以通过操作符"|"(或者)连接这些 DTD 元素定义.这样,元素实际上可以通过其子元素的正则表达式来完成.

算法.根据 DTD 构造 DTD 元素事件图.

输入:XML 结构约束 DTD 的定义.

输出:DTD 的元素事件图.

1. 初始化构造图 $G=(N,E,W,Y)$,其中,图 G 中包含两条边,e_1 和 e_2,e_1 标记为 startElement(root),权重为 1,e_2 标记为 endElement(root),权重为 1,e_1 的终点连接到 e_2 的初始点.

2. 扫描 DTD 中 e 元素的定义,定位当前 DTD 元素事件序列图中 startElement(e)边和 endElement(e)边,当前 startElement(e)的权重为 K.

3. 根据元素 e 的定义,我们假定 $e=f(k)$,构造元素 e 的事件序列子图,假定定义中包括 n 个不同元素,则增加构造子图的起始点 $startNode$ 和终止点 $endNode$,构造 $2n$ 条边,分别标记为元素的 startElement 和 endElement 事件.

如果元素 e 的定义中存在 $a.b$,则将 endElement(a)边的终止结点和 startElement(b)边起始节点合并起来;

如果存在 $a|b$,则将 startElement(a)边的起始结点和 startElement(b)边的起始结点合并,endElement(a)边的终止结点和 endElement(b)的终止节点合并;

如果存在 a^*的情况,则连接 endElement(a)边的终止节点和元素 startElement(a)事件边的起始节点,连接 startElement(a)边的终止节点和元素 endElement(a)事件边的起始节点,并且,StartElement(a)的权重是 k+1,endElement(a)的权重是 k+1.

上述操作递归完成.假定元素 $first$ 是元素 e 的可能的开始子元素,则连接 startNode 到 startElement($first$)的起始点;假定元素 end 是元素 e 的可能的最后子元素,则连接 endElement(end)的终止节点到 endNode;

4. 合并元素 e 所构造的子图到父图.

父图中断开 startElement(e)和 endElement(e)的连接,父图中 startElement(e)终点连接到子图定义的起始点 startNode,子图定义的终点 endNode 连接到父图中 endElement(e)边的起始点.

如果在子图中出现了父图中标记相同的边 startElement(d)和 endElement(d),则删除子图中 startElement(d)和 endElement(d),增加子图 startElement(d)的起点和父图的 startElement(d)的起点的空连接;增加父图 startElement(d)的终点和子图的 startElement(d)的终点的空连接;增加子图 endElement(d)的起点和父图的 endElement(d)的起点的空连接;增加父图 endElement(d)的终点和子图的 endtElement(d)的终点的空连接.

5. 重复第 2~4 步,直到所有的元素定义处理完毕为止.

根据 DTD 元素事件图的构造过程,我们分析 DTD 元素事件图的性质:

性质 1. DTD 元素事件图中不存在两条标记相同的边.在每个 DTD 子元素的子图构造中,不包含重复标记的边;而在子元素定义的元素事件序列子图和父元素的元素事件序列父图的合并过程中,去掉了重复的边.

性质 2. 在最终的 DTD 元素事件图中,如果出现 startElement(e)和 endElement(e)连接的情况,则元素 e 一定是简单元素.

性质 3. 对于满足 DTD 的 XML 文档中出现的任意事件序列,都可以在 DTD 元素事件图中找到对应边的连接序列.

分析事件序列的性质,事件序列都是从某个原子元素的 endElement 事件开始,到另一个原子元素的 startElement 终止.建立事件序列图的目的,就是压缩之前获取 XML 数据流中可能出现的元素事件序列.但是,如果 DTD 定义是递归的,则这个序列长度可能是无限的.下文中,我们给出非递归 DTD 所对应的 XML 中所包含的事件序列的最大长度,同时将它作为递归 DTD 所对应的 XML 中所包含的事件序列的长度的上限.

定义 2(DTD 的高度). 给定 DTD,我们标记 DTD 的根元素 r 的高度为 $h(r)=1$;如果元素 s 是元素 p 的子元素,则 $h(s)=h(p)+1$;每个元素仅标记一次,则 DTD 所有元素 $h(k)$的最大值称为 DTD 的高度.

根据上述 DTD 高度的定义,我们知道例 3 中 DTD 的高度为 4,标记序列如下:$h(R)=1$,$h(A)=2$,$h(K)=3$,$h(D)=4$.

在压缩过程中,我们希望能够保持 XML 数据流数据部分和结构部分之间的次序,所以,描述结构的事件序列结束于原子元素的 startElment 事件,开始于原子元素的 endElement 事件.在结构序列之间,实现 XML 数据部分的压缩.本文主要侧重于结构序列,即原子元素的 endElement 事件到原子元素的 startElement 事件的压缩,而这种序列的长度,对满足非递归 DTD 的 XML 文档来讲,与 DTD 的高度是相关的.

定理 1. 在满足非递归DTD的XML文档中,我们能够找到的XML数据流中最长元素事件序列长度是DTD高度的 2 倍.

证明:假定在高度为 H 的 DTD 的 XML 查询中,存在长度超过 $2H$ 的事件序列,则由于 DTD 的高度是 H,XML 数据流的 SAX 事件相当于按照前序扫描的事件,则超过 $2H$ 的元素事件序列一定会包含元素 k 的 startELement 边,其中元素 k 是原子元素,并且 k 不是序列的最后元素.而原子元素 startElement 事件之后,就是原子元素所对应的数据,所以,这个序列应该终止于元素 k 的 startElement 边,所以,这个事件序列是不合理的. □

对于递归 DTD 来讲,可能存在长度无限的元素事件序列.在本文中,我们将结构序列的长度限制在 DTD 高度的两倍.这种频繁序列长度,能够满足非递归 DTD 的要求.如果在遵从递归 DTD 的 XML 文件中,出现了大于 DTD 高度两倍的结构序列,则需要在压缩过程中动态产生事件序列所对应的编码.

算法.DTD 元素事件图中的元素事件序列的发现.

输入:DTD 的元素事件图.

输出:DTD 的元素事件图的发现.

增加 DTD 元素事件图中标记所有原子元素 endElement 边和 root 元素的 startElement 边到队列 Q 中;

元素事件序列集合 F 设置为空;

for (i=1 TO $2H$) //H 是 DTD 的高度

取出队列 Q 中的事件序列 $l=e_1,e_2,\ldots,e_i$,如果队列为空,则返回;

在 DTD 元素事件图中找到 e_i 的下一个非空元素事件边的集合 E;

形成新的元素事件序列 $L=\{l+e|e\in E\}$,增加到队列 Q 中;

对于任何 $l'\in L$,如果 l'元素事件序列的最后边标记的是元素 a 的 startElement 事件,并且元素 a 不是复杂元素,或者是 root 元素的 endElement 边,则 $F=F+\{l'\}$,同时在队列 Q 中删除 l';

EndFor

输出元素事件序列集合 F

元素事件序列的发现,采用类似于基于队列的图扫描的方法:首先给出长度为 1 的事件序列,其次逐渐增加事件序列的长度.根据 DTD 元素事件图的构造过程我们知道,对于满足 DTD 的 XML 文档中出现的事件序列,

都可以在 DTD 元素事件图中获取(见性质 3).

下一步,我们需要估算获取的元素事件序列的频率.给定元素事件序列 $l=e_1e_2...e_n$,则整个事件序列的频率等价于整个序列中频率最小的元素的频率,即 $weight(l)=\min(weight(e_1),\ldots,weight(e_n))$.每个元素事件所对应的权重是通过 DTD 事件图来获取的.在上文的 DTD 事件图的构造过程中,DTD 中的"*"的操作符号能够提高元素事件序列的权重.元素事件序列的权重实际描述了元素事件序列出现的频率,根据频率,我们可以构造元素事件所对应的 Huffman 编码[5].

2.2 基于DTD的压缩和解压缩数据

与基本的 XSC 压缩算法不同,我们首先根据 DTD 获取所有可能的元素事件序列,根据估算的频率,生成对应的 Huffman 编码,而不是在数据流处理过程中动态生成.采取这种处理方式,一方面能够获取更加合理的序列编码,另一方面能够提高实时压缩速率.如果 DTD 支持递归,则 XML 数据流中有可能出现长度超过两倍 DTD 高度的元素事件序列,在这种情况下,我们需要扩充元素事件序列字典编码表,扩充的部分需要加入到压缩数据中,并传递给数据目标.

图 2 给出了 XML 数据流的整个压缩处理框架.XML 数据流压缩之前,利用 DTD 构建 XML 元素事件序列图,获取事件序列,估算事件序列的频率,产生事件序列的 Huffman 编码.在 XML 数据流的解析过程中,如果发现 SAX 事件是 Text 事件,则表明与数据相关,调用数据压缩算法完成数据压缩,输出压缩结果;如果发现 SAX 事件不是 Text 事件,则获取最长的事件序列(从一个 Text 事件之后到另一个 Text 事件之前,中间不包含其他的 Text 事件),检查产生的 Huffman 编码,如果存在,则直接输出 Huffman 编码;如果不存在,表明是递归 DTD 的 XML 事件,则输出事件序列.整个 XML 数据流压缩处理形式是边扫描边输出.在压缩输出流中,采用前缀编码实现不同类型压缩数据的区别.

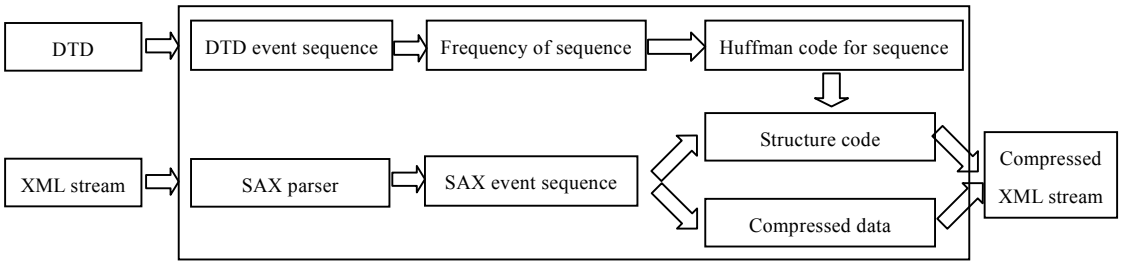

Fig.2 The whole compression architecture of optimized XSC

图 2 优化 XSC 整体压缩框架

3 实验数据

为了验证 XSC 压缩算法的有效性,我们建立了测试的实验环境,与传统压缩方法比较了数据的压缩率、压缩时间,比较了数据流压缩之后的执行效率和压缩前的执行效率.我们共实现了两种 XSC 算法,基本 XSC 算法在下文中简称为 BXSC 算法,优化 XSC 算法在下文中简称为 OXSC 算法.

我们采用 C=1–(压缩后的数据体积/压缩前的数据体积)作为压缩数据的压缩率[3].XPress[3]的研究结果表明,通用的压缩算法 GZip[6]对 XML 数据的压缩率,要高于支持压缩数据查询的 XPress 和 XGrind(由于它们支持压缩数据的直接查询),但要低于不支持压缩数据之上查询的 XMill 算法.本文主要比较 BXSC 算法和 OXSC 算法的压缩率.

3.1 环境构建

数据准备:本文的数据采用 XMill 的测试数据,目前测试了 4 种数据,包括 SwissProt 数据[7](112.1M),NASA 数据集合[8](24.4M),TreeBank_E[9]数据集合(84M)3 种真实数据,以及利用 IBM 的 XML 数据产生器基于 NASA 的 DTD 产生的模拟数据(137M)[10].测试运行在 Dell Optiplex 商用计算机,操作系统为 Windows 2000,内存 512M,处理器 1.5G,算法实现的语言为 Java 1.4.

3.2 实验结果

XPress[3]中比较了 Xgrind,XPress 以及 Gzip 的压缩比率.XGrind 和 XPress 保持了原有的数据结构,支持压缩数据的查询,所以压缩率远低于 Gzip 的压缩比率.XSC 压缩是对 XML 数据流的压缩,而数据流可以看做是一维数据的压缩,压缩需要一遍完成,但是在压缩数据中无须保存 XML 数据原有的结构,只需要保持元素事件先后顺序,所以,XSC 的压缩比率可能高于 XPress 和 XGrind.从图 3 的实验结果来看,XSC 与 Gzip 的压缩比率类似.在 4 组测试中,XSC 在两组测试集合之上压缩率要高于 Gzip,有一组持平,有一组略低,而 Gzip 的压缩比率普遍高于 XPress 和 XGrind.在 Generated data 中,由于存在大量的重复数据,Gzip 和 XSC 的压缩比率超过 95%.

由于优化的 XSC(OXSC)考虑了 XML 数据模式的信息,构造的元素事件序列的编码更加合理,这一特点使得 OXSC 的压缩率要高于没有考虑 XML 数据模式的压缩算法(BXSC).图 3 的实验结果证明了这一结论.

在 XML 数据流的压缩时间效率上,我们比较利用 SAX 解析器处理 XML 数据流的时间代价和利用 OXSC 处理 XML 数据流的时间代价.图 4 的实验结果表明,尽管 OXSC 中需要构造 DTD 的元素序列图,计算不同元素事件序列出现的频率,生成元素事件序列的编码,但是 OXSC 并没有导致数据流处理代价的急剧增长.

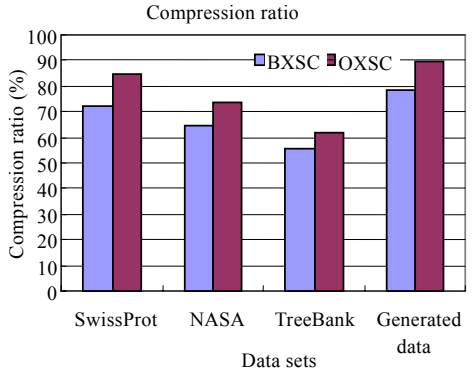

Fig.3 Compression ratio of BXSC and OXSC

图 3 BXSC 和 OXSC 的压缩比率

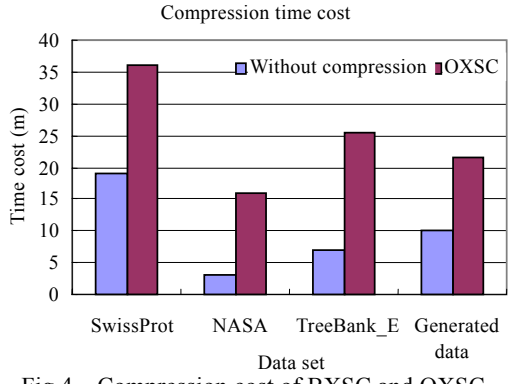

Fig.4 Compression cost of BXSC and OXSC

图 4 BXSC 和 OXSC 的压缩代价

XSC 算法的创新主要体现在 XML 数据流结构序列的压缩上.图 5 比较了基本 BXSC 和优化 OXSC 的结构序列压缩效果.由于优化 OXSC 算法能够分析 DTD 获取元素事件序列,并且按照估算的权重,获取近似优化的编码,而且,基本 BXSC 中包含了大量的中间结构编码,所以,优化 OXSC 算法对于结构序列的压缩比率要高于基本 XSC 算法的压缩比率.

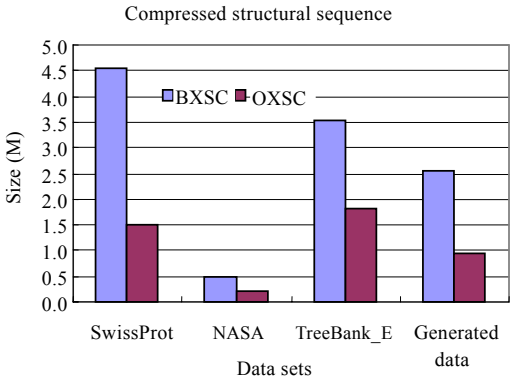

Fig.5 Size of compressed file with OXSC and BXSC

图 5 OXSC 和 BXSC 压缩文件大小

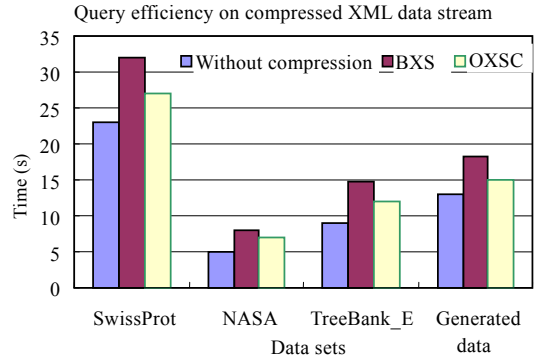

Fig.6 Query cost on compressed data

图 6 压缩数据上的查询代价

图 6 主要比较了查询在压缩数据之上和在非压缩数据之上的查询处理代价.我们在压缩数据流上执行一

组单个 XPath 查询,比较压缩前和压缩后的平均执行效率.在压缩数据流之上执行 XPath 查询的代价,与压缩前相比,并没有明显的增加.优化的 XSC 算法与基本 XSC 算法相比,保存了较少的元素事件序列,从而减少了序列的解码次数.OXSC 算法所产生的压缩数据之上的查询效率要高于 BXSC 算法所产生的压缩数据之上的查询效率.

4 结束语

本文考虑了 XML 数据流的环境要求,提出了适应数据流环境中的 XML 数据流压缩算法,扫描数据流一遍完成数据压缩.而且,本文将 DTD 引入了压缩过程,利用 DTD 产生了更加合理的编码,提高了数据压缩率和压缩效率.

References:

[1] Hartmut L, Dan S. XMill: An efficient compressor for XML data. In: Weidong C, Jeffrey F, eds. Proc. of the SIGMOD 2000. Texas: ACM Press, 2000. 153−164.

[2] Pankaj MT, Jayant RH. XGRIND: A query friendly XML compressor. In: Proc. of the ICDE 2002. San Jose: IEEE Computer Society, 2002. 225−234.

[3] Jun KM, Myung JP, Chin WC. XPRESS: A queriable compression for XML data. In: Alon Y, Zachary G, eds. Proc. of the SIGMOD 2003. San Diego: ACM Press, 2003. 122−133.

[4] Jacob Z, Abraham L. A universal algorithm for sequence data compression. IEEE Trans. on Information Theory, 1977,23 (3):337−343.

[5] Jeffery SV. Design and analysis of dynamic Huffman codes. Journal of the ACM, 1987,34(4):825−845.

[6] Jean LG. GZIP. 2003. HTTP://www.gzip.com

[7] SwissProt Data Set. 1998. http://www.cs.washington.edu/research/xmldatasets/data/SwissProt/SwissProt.xml

[8] NASA Data Set. 2001. http://www.cs.washington.edu/research/xmldatasets/data/nasa/nasa.xml

[9] Tree Bank Data Set. 2002. http://www.cs.washington.edu/research/xmldatasets/data/treebank/treebank_e.xml

[10] Angel LD, Douglas L. XML generator. 1999. http://www.alphaworks.ibm.com/tech/xmlgenerator