基于节点位置信息的降低更新代价前缀编码方案研究

徐 娟[1] 李战怀[1] 娄 颖[1,2]

(西北工业大学计算机学院 西安 710072)[1] (河南科技大学电子信息工程学院 洛阳 471003)[2]

摘 要 分析了现有的几种XML文档前缀编码[1-4]方法,研究了在XML文档树不同位置插入节点时的更新代价,提出了一种基于位置信息的前缀编码方案,对更新代价较大的节点预留较大的空间。设计了更新算法,在产生新插入节点的编码的同时,为今后插入节点也预留空间,且采用"借"空间算法,减小插入操作造成重新编码的数量。充分的试验结果证明,采用提出的编码方法,具有相对较小的平均编码长度和编码时间,查询速度很快,更重要的是能够有效降低更新操作引起的编码长度增加、重新编码节点数以及更新时间。

关键词 XML,前缀编码,更新代价,预留空间

Novel Position-based Prefix Encoding Approach to Reduce Update Cost for Dynamic XML Data

XU Juan[1] LI Zhan-huai[1] LOU Ying[1,2]

(School of Computer Science and Technology, Northwestern Polytechnical University, Xi'an 710072, China)[1]

(Electronic Information Engineering College, University of Technology of Henan, Luoyang 471003, China)[2]

Abstract Prefix scheme[1-4] is popular to label XML tree. Inserting nodes or sub-tree in XML tree will cause abundances of nodes renumbering. Nodes inserted in different positions cause different update cost. We proposed a novel position-based prefix encoding approach to reduce update cost for dynamic XML data, which preserves bigger coding space for the higher update cost nodes. In our update algorithm, coding space was preserved when the nodes were inserted. And we proposed "borrow" space algorithm, which can decrease the number of nodes renumbering by inserted operation. Extensive experimental results show that the new position-based prefix encoding approach has relatively smaller mean coding length and encoding time, higher query response time. And the method can effectively decrease the increasing code length, renumbering nodes number and update time causing by update.

Keywords XML, Prefix encoding, Updating cost, Preserving coding space

1 前言

XML已经成为互联网上数据表示和交换的标准,大量的XML文档出现在网络中,有效地存储XML数据并提供高效的XML数据查询,成为当今急需解决的问题。目前,大部分有关XML数据的索引和查询技术都是基于某种对XML文档树的编码方法。XML编码就是按照一定规则给XML树中每一个节点分配唯一的编码,通过编码,可以在不遍历XML文档树的前提下,直接判断两个节点之间的关系。目前,区间编码和前缀编码是主流的两种编码方式。区间编码方法采用深度优先遍历XML树的方法给XML文档中的每个节点赋予一对整数值,使它们形成的区间涵盖其后裔节点的编码区间范围,这样对节点间的结构关系的判断等价于区间包含关系的判断。前缀编码则保存了元素的路径信息,任何元素的编码是其后裔元素编码的前缀,这样元素间祖先后裔关系的确定就等价于基于前缀字符串包含关系的判断。现在已经提出了很多处理基于路径索引,编码等方式的查询处理技术,但是,大多数方法只能提高对XML数据的查询性能,当XML数据需要频繁地更新时,现有的编码方案均需要将XML树中的大量节点重新编码,以维持XML文档的顺序,这样便产生了很高更新代价,形成了系统性能的瓶颈。当XML文档树中插入节点或子树时,采用前缀编码只需要将插入节点的右兄弟节点及其子孙节点重新编码,但是区间编码则需要将插入节点文档序之后的所有节点重新编码,所以在对更新操作的支持上,重新编码的节点数量远远小于采用区间编码,前缀编码优势要远远高于区间编码[1]。XML全文检索是当前XML领域的一个新的研究热点[7],大多数XML全文检索算法均使用前缀编码方案对XML文档进行编码。因此,对前缀编码的研究,特别是如何降低频繁更新XML数据的更新代价,对于解决XML研究领域的许多问题很有意义。本文对现有的前缀编码方案进行分析比较,研究了XML文档树中节点的位置特性,提出一种基于更新代价的前缀编码方案,该方法具有以下特点:

(1) 可以在常数复杂度时间内判断树中任意两个节点间结构关系;

(2) 充分考虑了节点在XML文档树中的位置特性,对于

到稿日期:2008-03-18 本文受国家自然科学基金(60573096)资助。

徐 娟 博士生,主要研究方向为XML数据管理,E-mail: xuj@mail.nwpu.edu.cn;李战怀 博士,教授,博士生导师,主要研究方向为数据库理论与技术。

不同位置上的节点，预留不同大小的空间，使得在对 XML 文档树进行节点或子树的插入、删除等更新操作时，尽量减少重新编码节点的数量。

本文的内容安排如下：第 2 节详细介绍了 XML 文档前缀编码的研究现状，分析了各种编码的优缺点；第 3 节综合考虑更新代价，提出了一种新的前缀编码方案；第 4 节设计了使用该前缀编码时的更新算法；第 5 节研究了该方案的性能；最后总结了全文。

2 相关研究

Tatarinov 等人[1]使用 Dewey ID 对 XML 文档树进行编码，XML 文档中的每一个元素表示为一个向量，其中：(1)根节点用空串标记；(2)对于非根节点 u，当 u 是 s 的第 n 个孩子时，label(u)=label(s).n。label(s)表示 u 的父节点 s 的编码，整数 n 表示节点 u 的自编码(self_label)，“.”是连接父节点编码和自编码的分隔符。Dewey ID 支持元素间有效的结构连接，即当且仅当 label(u)是 label(s)的前缀时，元素 u 是 s 的祖先。Dewey 编码的结构关系判断条件简单，查询效率较高。

Cohen 等人[3]提出了简单前缀(Simple Prefix，SP)编码方案，该编码方案使用二进制字符串标记 XML 树中的节点，将树根编码为空串，根的第一个孩子 label 是 0，第二个是 10，第三个是 110，第四个是 1110，以此类推。显然 SP 编码的码长很大，而且它不能避免更新操作时的重新编码问题。

张剑妹等人[4]提出了一种适用于顺序 XML 树的前缀编码方法。在这种编码方法中，节点编码由自编码、前缀码和自编码的长度码 3 部分组成。每个节点的第一个孩子节点的自编码为“1”，第二个孩子节点的自编码为“2”，当节点的顺序码全为“9”时在自编码后补加“0”，这样第九个孩子的自编码为“90”，第十个孩子的自编码为“91”，依此类推。每个节点编码都继承其父节点的自编码和前缀码作为其自编码的前缀。为了便于从节点编码中提取节点自编码，在每个节点编码的前面用固定长度的编码存储节点顺序码的长度。这种编码对顺序 XML 文档查询能够较好的支持，也具有较小的更新代价，但是，其最大的不足是编码长度较大。

Dewey 编码、SP 编码和文献[4]提出的编码共同的缺陷是不能解决频繁更新时，大量节点需要重新编码的问题，需要对插入节点之后的兄弟节点和他们的子孙进行重新编码以保持文档顺序，更新代价非常大。

ORDPATH 编码[5]是一种扩展的 Dewey 编码，它使用奇数进行初始编码，当 XML 树进行更新时，它使用两个奇数之间的偶数连接一个奇数作为新的编码。ORDPATH 编码能够部分解决更新操作造成的重新编码问题。但是 ORDPATH 编码浪费了一半编码空间，使得编码的规模很大。由于 ORDPATH 需要花费很多的时间基于奇数或者偶数来判断前缀的层次，它的查询效率低下，而且会出现编码溢出，不能完全解决 XML 文档的动态更新。对于 XML 数据库而言，为了解决动态更新问题，使用 ORDPATH 便大大降低了查询性能，使用这种编码方式的代价太大。

图 1 列出了几种前缀编码方案示例。现有的前缀编码方案都不能完全解决动态更新问题，或者代价太大。针对这个问题，本文定义了不同位置上节点更新代价的概念，对于文档树中不同位置节点，基于其更新代价，预留相应的编码空间，这种编码方案获得较好的查询性能和较小的更新代价。

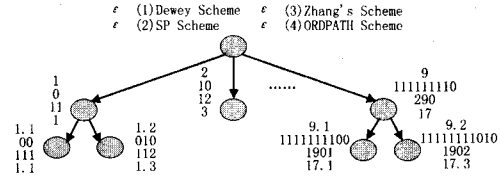

图 1 XML 文档树的各种前缀编码方案

3 新的前缀编码方案

当在文档树中插入节点或子树时，会造成树中部分节点的重新编码，进而引起更新操作的代价增高，为了度量重新编码节点的个数，我们首先引入更新代价的概念。

定义 1 给定一棵已编码的 XML 文档树，在指定位置插入一个节点或子树引起原文档树中节点重新编码节点的个数称为更新代价。

使用前缀编码的 XML 文档树，在不同位置插入新节点(或子树)所引起的更新代价不同。插入节点的层次越低(位置越靠近根节点)，且节点在兄弟间越靠左，插入节点所引起的更新代价越大(即插入操作引起重新编码节点的数目越多)；相反，插入节点引起的更新代价越小。为了克服文献[1-4]中提出的前缀编码的不足，对 XML 文档树中不同位置的节点，根据更新代价大小，相应的预留编码空间，对于更新代价大的节点，预留较大的空间，而对于更新代价小的节点，预留较小的空间。

我们用符号 label(u)=precode(u).self(u)表示节点 u 的编码。其中，precode(u)是节点 u 的前缀码，即 u 父节点的编码；self(u)表示节点 u 的自编码，表征兄弟节点之间的先后(左右)关系；“.”是分隔不同编码分量的连接符。

为了方便今后对 XML 文档树进行更新操作，减小更新代价，在不同位置需要预留不同大小的编码空间，因此，节点 u 的自编码分量 self(u)与其对应的位置参数 n(树中层次)和 k(兄弟节点间的位置)相关。其中 k 表示节点 u 是其父节点的第 k 个孩子。假设节点 w 是节点 u 与 v 的双亲节点，当 u 不是 w 的第一个孩子，v 是 u 最近的左兄弟节点。这样，节点 u 的自编码分量 self(u)定义为

$$self(u)=\begin{cases} self(v)+\Delta(n,k), & u\text{ 不是 }w\text{ 的第一个孩子} \\ \Delta(n,k), & u\text{ 是 }w\text{ 的第一个孩子} \end{cases} \quad (1)$$

$$\Delta(n,k)=\begin{cases} \alpha(M-n)+\beta(L-k), & \alpha(M-m)+\beta(K-k)>0 \\ 1, & \alpha(M-m)+\beta(K-k)\leqslant 0 \end{cases} \quad (2)$$

其中，M 为大于等于 Tr 深度 N 的整数，L 为大于等于 Tr 中节点最大扇出 K 的整数，而 α,β 为调节预留空间的参数，可以由用户根据实际情况(如 XML 文档更新的频率等)设定。

这样，本文提出的前缀编码方案可以描述如下：(1)我们将根节点 r 编码为空串 ε；(2)对于非根节点 u，首先根据其位置参数 n 和 k，以及相邻左兄弟 v 的自编码 self(v)，代入式(1)，计算节点 u 的自编码 self(u)，将节点 u 的双亲节点 w 的编码 label(w)作为前缀码(precode(u))，与自编码 self(u)连接，即可获得 u 的编码为 label(u)=precode(u).self(u)=label(w).self(u)。

对应的子树编码算法如图 2 所示。其中 SubEncoding 为

递归算法，用以计算以 lr 为根的子树 sTr 的扩展前缀编码；XML 文档树编码算法如图 3 所示。

```
Algorithm 1:SubEncoding(sTr,lr,nlevel)
Input:sTr 是 XML 文档树中的一棵子树,lr 是子树 sTr 的根节点编
      码,nlevel 表示子树 sTr 的根节点在整个文档树中的层次
Output:子树 sTr 所有节点的前缀编码
Description:
1: nsib=0  // 记录孩子节点的位置
2: r=root(sTr)  // 记录子树 sTr 的根节点
3: pre_self=0  // 记录前一个孩子节点的自编码
4: 令 nd 指向节点 r 的第一个孩子
5: while nd 不为空,即存在 r 的孩子节点没有编码
6:   nsib++
7:   delta=α(M-nlevel)+ β(L-nsib)
8:   if delta<=0
9:     delta = 1
10:  end
11:  self(nd)=pre_self+delta
12:  label(nd)=lr. self(nd)
13:  SubEncoding(ST(nd),label(nd),nlevel)
     //ST(nd)表示以节点 nd 为根的子树
14:  pre_self = self(nd)
15:  nd 指向节点 r 的下一个孩子
16:end
17:return
```

图 2 子树编码算法

```
Algorithm 2:Prefix_Encoding(Tr)
Input:Tr 是 XML 文档树
Output:文档树 Tr 中所有节点的前缀编码
Description:
1: lr=NULL // 令 Tr 的根节点编码为空
2: nlevel=1 // 令 Tr 的根节点层次为 1
3: SubEncoding(Tr,lr,nlevel)
4: return
```

图 3 XML 文档树编码算法

对于图 1 所示的文档树，根据本文提出的编码算法，我们可以得到各节点的编码(其中，$L=3, M=3, \alpha=2, \beta=2$)，如图 4 所示。

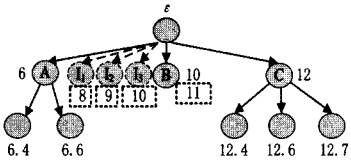

图 4 基于更新代价前缀编码方案示例

为 XML 文档树中的节点预留编码空间是更新操作影响更新代价的重要因素。对于 XML 文档树上不同位置的节点，通过本文提出的编码方案，需要对更新代价较大的节点预留较大的空间，因为在这些节点右侧插入新的兄弟节点会引起较多的后继兄弟节点及后继兄弟节点的后裔节点重新编码。节点 v 的预留空间可以表示为

$$es(v)=self(next\_sibling(v))-self(v)-1 \quad (3)$$

其中，$next_sibling(v)$是节点 v 的相邻右兄弟节点，若该节点不存在，$self(next_sibling(v))$为∞，即节点 v 的预留空间是∞。

下面研究文献[1,4,5]以及本文提出的编码方案中不同节点预留空间特性，结果如表 1 所列。

表 1 XML 文档前缀编码方案预留空间

Preorder Number	2	3	4	5	6	7	8	9
Dewey Scheme	0	0	∞	0	∞	0	0	∞
Zhang Scheme	0	0	∞	0	∞	0	0	∞
ORDPATH Scheme	∞	∞	∞	∞	∞	∞	∞	∞
Proposed Scheme	4	1	∞	1	∞	1	0	∞

由表 1 可以看出，文献[1,4]提出的前缀编码方案为连续编码，没有预留空间；ORDPATH 编码没有考虑节点的位置特性，给所有节点预留了同样的空间，浪费了编码空间；而本文的前缀编码方案充分考虑了节点的位置特性，可以更好地降低更新代价。对于考虑左侧插入时对应的预留空间具有同样的结论。

4 更新算法

删除操作不会造成节点的重新编码，因此，本文考虑的更新指的是插入操作。由第 3 节可知，根据节点位置不同，预留了不同的空间，对于距根节点越近、在兄弟间越靠左的节点，预留较大的空间，反之，留有较小的空间。

下面，我们具体分析一下不同位置插入时的更新操作，主要包括生成插入节点和重新编码后继节点 2 种操作。

4.1 插入节点编码

在不同位置插入新节点，其编码的生成并不相同。

• 在节点 w 的第一个孩子节点 child_first 的左侧插入一个节点

假设节点 child_first 的前缀编码为 $precode_0$、自编码为 $self_0$，若 $self_0$ 大于 1，则表示该节点预留了空间，新插入的节点 child_insert 的编码可以表示为 $precode_0.ceil(self_0/2)$，其中 ceil(•)表示向上取整函数；若 $self_0$ 等于 1，则表示该节点没有预留空间了，节点 child_insert 的编码即为 child_first 的编码，而节点 child_first、后继兄弟节点以及它们的后裔节点需要重新编码。

• 在节点 w 的最后一个孩子节点 child_last 的右侧插入一个节点

由于 child_last 是兄弟节点最后一个节点，在其右侧插入一个新节点 child_insert，不会引起其它节点的重新编码。child_insert 的编码可以表示为 $precode_N.self_N+1$，其中 $precode_N$，$self_N$ 为节点 child_last 的前缀编码和自编码。

• 在节点 w 的相邻两个孩子 child_1 和 child_2 之间插入一个节点

假设节点 child_1 和 child_2 的前缀编码为 precode，自编码分别为 $self_n$ 和 $self_{n+1}$。若 $self_n+1<self_{n+1}$，则表示该位置有预留空间，则新插入的节点 child_insert 的编码可以表示为 $precode.ceil((self_n+self_{n+1})/2)$；若 $self_n+1=self_{n+1}$，则表示该位置没有预留空间了，节点 child_insert 的编码即为 child_2 的编码，而节点 child_2、后继兄弟节点以及它们的后裔节点需要重新编码。

这样的插入节点编码生成方式可以为今后插入操作留有空间。

4.2 后继节点重新编码

对于后继节点的重新编码，基于本文提出的编码方案，可以采用一种“借”空间的方法来减小重新编码节点的数量。这种“借”空间的方法可以描述为：

(1)设当前需要更新的节点为 *node*1，其前缀编码为 precode，自编码为 $self_1$，深度为 S；

(2)若节点 *node*1 是兄弟节点中最后一个兄弟，*node*1 的编码为 precode.($self_1+1$)，更新以 *node*1 为根节点的子树中所有节点(除根节点)的第 S 个分量编码为($self_1+1$)，结束重新编码；

(3)若节点 *node*1 不是兄弟节点中最后一个兄弟节点，即存在相邻右兄弟节点 *node*2，假设 *node*2 的自编码为 $self_2$；

(4)如果 $self_1+1<self_2$，表示存在预留空间，*node*1 的编码为 precode.ceil(($self_1+self_2$)/2)，更新以 node1 为根节点的子树中所有节点(除根节点)的第 S 个分量编码为 ceil(($self_1+self_2$)/2)，结束重新编码；

(5)如果 $self_1+1=self_2$，表示不存在预留空间，*node*1 的编码为 *node*2 的编码，即为 precode.$self_2$，更新以 *node*1 为根节点的子树中所有节点(除根节点)的第 S 个分量编码为 $self_2$。设置 *node*2 为当前更新节点，重复第(1)步。

更新算法的示例如图 4 所示。在节点 A 和 B 之间插入节点 I1，编码为 8，同时为 I1 和 B 之间插入节点 I2 预留空间，但是，I2 和 B 之间插入节点 I3 时，就需要借用节点 B 和 C 之间的预留空间，其中 I3 的编码为 10，B 的编码为 11。

5 编码性能分析

这一节我们研究了本文提出的前缀编码方案的编码长度、查询效率和更新代价等性能，并将它们与文献[1,4,5]提出的编码方案的性能进行比较。本文的测试程序是基于 Visual C++ 6.0 实现的，运行硬件参数为：Pentium IV CPU，主频为 2.4GHz，2G DDR 内存，120 G 的 IDE 硬盘，操作系统为 Windows XP Professional。仿真的数据集都是来自现实世界的一些 XML 数据，相关参数如表 2 所列。对于本文提出的编码方案，根据参数集的特点，参数选择如下：$L=50$，$M=7$，$\alpha=5$，$\beta=1$。另外，为了充分说明 ORDPATH 编码方案的特点，原始编码时，考虑有 1/3 的节点是被插入的。

表 2 测试数据集

Data-sets	Topics	# of files	Max / average fan-out	Max / average depth	Max/ Total of nodes
D1	Movie	490	38/2.74	5/4.34	125/26044
D2	Depart-ment	19	257/3.76	4/3.34	2840/48542
D3	Actors	482	368/2.83	5/4.57	1110/56995
D4	Play	4	434/5.60	7/5.77	179690/392996
D5	Shakespeare's play	37	434/5.57	6/4.77	6636/179689
D6	NASA	2436	1190/2.94	7/4.56	6022/533193

5.1 编码长度和时间

图 5－图 6 显示了采用不同编码方案对数据集 D1－D6 进行编码时的平均编码长度和编码时间。如图 5 所示，采用本文提出的编码方案，随着平均扇出和平均深度的增加，平均编码长度也增加，但是，总体来说，比 Zhang 编码和 ORDPATH 编码的平均编码长度小，而大于传统的 Dewey 编码。由图 6 可以看出，Dewey 编码、Zhang 编码和本文提出的编码这 3 种方案，由于涉及到的操作都是简单的基本运算，编码时间基本相同；而 ORDPATH 编码由于编码时，节点之间的层次关系判断比较复杂，因此相应的编码时间也较长，编码时间性能较差。

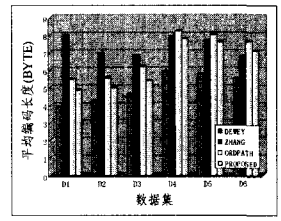

图 5 不同编码方案对应的平均编码长度

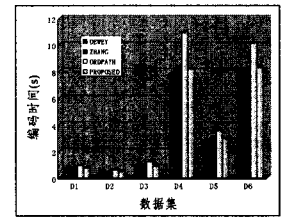

图 6 不同编码方案对应的编码时间

5.2 查询效率

为了测试各种编码方案的查询性能，设计仿真试验如下：首先将数据集 D5 重复 10 次，这样数据集中的文件数变成 370 个；然后对这个扩展的数据集进行编码；基于这些编码查询如表 3 所列的正则路径表达式[6]的结果(其中，Q1，Q3，Q4 是包含顺序的查询)。图 7 显示了不同编码方案对查询 Q1－Q5 的响应时间。如图 7 所示，由于 Dewey 编码、Zhang 编码和本文提出的编码这 3 种方案，在比较两个编码的大小时，层次判断简单，并且分量编码比较只是简单的数值比较，因此查询响应时间较快，基本相同。而 ORDPATH 编码方案，编码比较时，同样由于层次关系判断比较复杂，且分量编码比较时，需要进行多次数值比较，因此相应的查询响应时间较长。

表 3 数据集 D5(重复 10 次)上的查询

	Queries	# of nodes retrieved
Q1	/play/act[4]	370
Q2	/play//personae[. title]/pgroup[. //grpdescr]/persona	2690
Q3	/play/personae/persona[12]/proceding-silling:: *	4240
Q4	//act[2]/following: speaker	184060
Q5	//act/scene/speech	309330
Q6	/play/ * //line	1078330

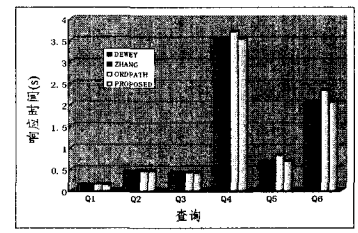

图 7 不同编码方案对应的查询响应时间

5.3 更新代价

下面，我们讨论一下各种编码方案的更新代价。这里，考虑两种更新方式：均匀更新和倾斜式更新。其中，均匀更新表示在文档树中任意位置进行频繁插入；而倾斜式更新指的是在一个固定的位置进行频繁插入，这种更新方式的更新代价通常是最大的。这一部分的试验数据选择数据集 D5 中 XML 文件 Hamlet.xml。

5.3.1 均匀更新

Hamlet.xml 文件对应的文档树中包含 6636 个节点，在任意两个连续节点之间插入一个新的节点，总共需要插入

6635 个节点，这样新的文档树中包含 13271 个节点。定义这样的一系列操作为一个插入系列。同理，基于这个新文档树，第二个插入系列需要插入 13270 个节点，依次类推。

图 8—图 10 显示了均匀更新时，不同编码方案对应的编码长度的增加、重新编码节点数以及更新时间。如图 8 所示，随着编码规模的增加，Zhang 编码和 ORDPATH 编码的平均编码长度急剧增加，这样就导致它们的增加编码长度要大于 Dewey 编码和本文提出的编码；由于存在预留空间，更新时，采用本文提出的编码时增加的编码长度要大于 Dewey 编码。如图 9 所示，由于 Dewey 编码和 Zhang 编码是连续编码，每次插入操作都会引起后继兄弟节点及后继兄弟节点的后裔节点的重新编码，重新编码节点数量较大，效率很低；ORDPATH 编码采用了负数和偶数作为预留空间，可以不进行重新编码；而本文提出的编码在一些可能更新代价较大的位置上预留了空间，因此，插入操作引起的重新编码数目较小。如图 10 所示，由于增加的编码长度、编码时间和重新编码节点数的共同影响，本文提出的编码对应的均匀更新时间远小于 Dewey 编码和 Zhang 编码，略大于 ORDPATH 编码。

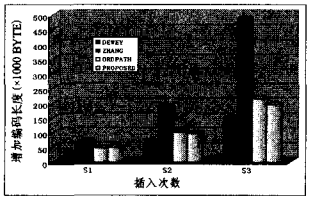

图 8 均匀更新时不同编码方案对应的增加编码长度

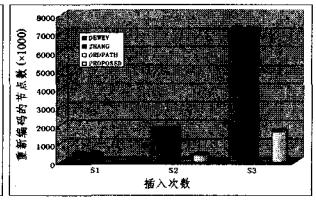

图 9 均匀更新时不同编码方案对应的重新编码节点数

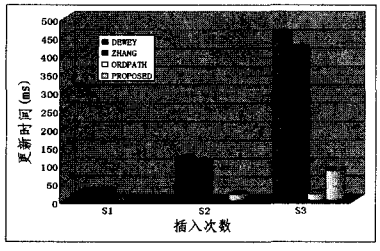

图 10 均匀更新时不同编码方案对应的更新时间

5.3.2 倾斜式更新

为了测试倾斜式更新时的更新代价，我们在 Hamlet. xml 对应的文档树中，任意选择一个位置，连续插入 200 个节点。图 11—图 13 显示了倾斜式更新时，不同编码方案对应的编码长度的增加、重新编码节点数以及更新时间。对于更新引起的编码长度的增加，采用本文提出的编码小于 Zhang 编码和 ORDPATH 编码，但是，大于 Dewey 编码；对于更新引起的重新编码节点数量，采用本文提出的编码要小于 Dewey 编码和 Zhang 编码，但是大于 ORDPATH 编码；而对于更新时间，本文提出的编码小于 Dewey 编码和 Zhang 编码，大于 ORDPATH 编码。

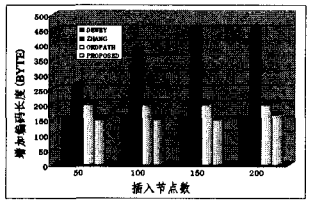

图 11 倾斜式更新时不同编码方案对应的增加编码长度

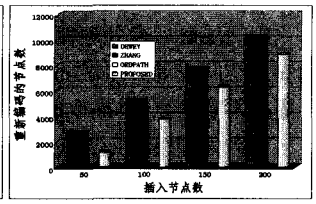

图 12 倾斜式更新时不同编码方案对应的重新编码节点数

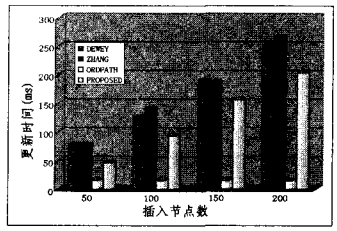

图 13 倾斜式更新时不同编码方案对应的更新时间

总体来说，本文提出的编码方案在处理更新时，综合考虑了编码长度、重新编码节点数以及更新时间，是一种较好的前缀编码，尤其是在处理均匀更新操作，综合性能更好。

结束语 为了有效地支持以路径表达式为核心的 XML 查询，需要能够快速判断 XML 文档中元素间的相对关系，前缀编码是一种主流的 XML 文档编码方法。本文分析了现有的几种 XML 文档前缀编码方法，研究了在 XML 文档树不同位置插入节点时的更新代价，提出了一种基于位置信息的前缀编码方式，给出了明确的节点编码的计算表达式。该方法对更新代价较大的节点预留较大的空间，而对于更新代价较小的节点预留较小的空间。同时给出了更新算法，产生新插入节点的编码的同时，为今后插入节点也预留空间，而采用“借”预留空间算法，减小插入操作造成重新编码的数量。通过试验分析证明，采用本文提出的编码方法，具有相对较小的平均编码长度和编码时间，查询速度很快，更重要的是能够有效降低更新操作引起的编码长度增加、重新编码节点数以及更新时间，尤其是在处理均匀更新操作，性能更优。因此，本编码方法在不提高编码效率和不损失查询速度的前提下，可以较好地解决更新操作所造成的节点重新编码的问题，是一种较好的前缀编码。

参 考 文 献

[1] Tatarinov I, Viglas S, Beyer K S, et al. Storing and querying ordered XML using a relational database system // Proceedings of SIGMOD. 2002: 204-215

[2] Kaplan, Milo T, Shabo R. A Comparison of Labeling Schemes for Ancestor Queries // Proceedings of ACM-SIAM Symposium on Discrete Algorithms. 2002

[3] Cohen E, Kaplan H, Milo T. Labeling Dynamic XML Tree // Proceedings of the 21st ACM Symposium on Principles of Database Systems. Madison, Wisconsin, USA, 2002: 271-281

[4] 张剑妹，陶世群. 一种适用于顺序 XML 树的前缀编码方法. 计算机应用，2005，25(12)：2879-2881

[5] O'Neil P, O'Neil E, Pal S, et al. ORDPATHs: Insert-friendly XML node labels // Proceedings of SIGMOD. 2004: 903-908

[6] Li C, Ling T W, Hu M. Efficient Updates in Dynamic XML Data // Proceedings of ICDE. 2006: 13-22

[7] Amer-Yahia S, Shanmugasundaram J. XML Full-Text Search: Challenges and Opportunities // Proceedings of VLDB. 2005: 1386-1387