

XML 数据扩展前序编码的更新方法

Updating of Extended Preorder Numbering Scheme on XML

罗道峰 孟小峰 蒋喻

中国人民大学信息学院 北京 100872

Abstract

Most of XML query processing strategies are based on some numbering scheme. Nodes on the XML tree will be assigned a unique code by the numbering scheme, and ancestor-descendant relationship could be directly told through the codes. The most famous numbering scheme is Region Based Numbering Scheme. However, XML data will be updated. Once the data is updated, the region code should be adjusted to keep the indexing and query processing techniques working. Unfortunately, few studies have been reported on the issue of the numbering scheme. This paper focuses on this issue, proposing a series of space preserving and updating algorithm. Extensive experiments are conducted to test the effectiveness of the algorithms.

关键字: XML, Region 编码, 预留, 更新

摘要

大部分 XML 查询技术都是基于某种对 XML 树的编码方法。对 XML 树的编码,是指按照某种规则对 XML 树的每一个结点分配唯一的编码,目的是通过任意两个结点的编码,能够直接判断两个结点之间是否具有祖先后代关系。最常用的编码方法是区域编码方法 (Region Based Numbering Scheme)。然而,XML 数据也会面临插入删除等更新问题。数据一旦更新,区域编码也要作相应的调整,才能保证基于这个编码的各种索引和查询算法的正确性。在编码的更新方面,目前研究得还不多。本文主要研究区域编码的更新问题,采用预留编码空间的方法,针对不同特征的 XML 数据和应用环境提出了一整套预留算法和编码更新算法,并做了大量的实验,检验这些算法的有效性。

1. 引言

随着 XML 的迅速发展,对 XML 数据的高效索引和查询的需求越来越迫切。近年来,人们提出了各种各样的关于 XML 数据的索引和查询技术,如 EE-Join, EA-Join 和 KC-Join[6], MPMCJN[8], tree-merge, stack-merge[1], XPath Accelerator, Containment Join size Estimation[9], 等等。这些技术大部分基于某种对 XML 树的编码方法[2,4,5,6,8,10],其中最为流行的一种是基于区域的编码方法[2,4,6,8] (Region Based Numbering Scheme)。

在基于区域的编码方法中,树中的每个结点都被赋予一对数字,这对数字表达这个结点所覆盖的区域。如果一个结点的区域包含另一个结点的区域,则在树

中前者是后者的祖先。在[2,8]中,每个结点以它在树的前序遍历顺序中的开始 (start) 和结束 (end) 编号作为编码。这种编码方法最大的缺点是缺乏灵活性,一旦插入新的子树,整个树不得不重新编码。为了解决这个问题,[6]提出了一种扩展的前序编码方法 (extended preorder numbering scheme),每个结点都用一个 $\langle \text{order}, \text{size} \rangle$ 对来标识,其中,order 是该结点在树中的前序遍历顺序, size 是以该结点为根的子树的结点个数。对于给定结点 T1 和 T2, T1 是 T2 的祖先,当且仅当 $T1.\text{order} < T2.\text{order}$ 且 $T1.\text{order} + T1.\text{size} > T2.\text{order} + T2.\text{size}$ 。该编码方法在编码时给结点的 size 值比它实际的大小要大,这样,就可以容纳将来新插入的结点,而不影响其它结点的编码。

当 XML 文档发生更新时(如插入一棵子树),需要相应地调整结点的编码以维持其反映祖先-后代关系的特性。这种重新编码的代价可能是很大的,有时甚至需要更新整棵树的编码。减少编码更新代价的关键在于怎样预留编码空间和更新发生时如何重新编码。不幸的是,就我们所知,在这个问题上研究得还比较少。尽管[6]曾提到通过给结点一个比实际大小更大的一个 size 来容纳将来的插入,但是它并没有涉及如何预留以及将来如何使用预留的空间的问题。本文将就区域编码的预留和更新问题进行讨论。这篇文章的主要贡献如下:

- 提出了一整套编码预留算法。不同的数据和应用,对编码预留有不同的影响。本文提出了在模式独立、基于数据模式和基于更新模式三种情况下的编码预留算法。

- 提出了编码的单次更新算法，并在此基础上进一步提出了批量更新算法。

- 我们设计并实现了在各种不同的情况下的编码更新，并且通过实验展示了以上几种方法的有效性。

以下的内容是正文的展开部分，安排如下：第二部分详细分析影响编码预留和更新的因素。第三部分针对影响编码预留的因素讨论不同的编码空间预留算法；第四部分阐述编码的单次更新和批量更新算法；第五部分是实验结果；第六部分总结全文和展望下一步工作。

2. 影响编码预留的因素

在一棵新的子树 (inserted subtree) 插入到一棵 XML 目标树 (target tree) 中之前，我们需要确定子树的插入位置 (insert position)。在这里，我们用元组 $\langle \text{parent}, \text{childIndex} \rangle$ 表示插入位置，其中 parent 表示 inserted subtree 的父亲， childIndex 表示它在它父亲的孩子中的编号 (第一个孩子编号是 0)。

在编码的预留和更新中，有下面几个考虑因素：

1) 数据模式。

在 XML 的更新的处理中，数据模式扮演了很重要的角色。根据数据有没有模式约束，数据可以分为两种：基于模式的数据和模式独立的数据。对基于模式的数据，模式信息 (如 DTD 或者 XML Schema) 对插入的子树有所限制，因为插入子树之后的数据，仍需要符合模式定义；而对模式独立的数据来说，XML 数据的插入没有任何限制，它可以发生在树中的任何可能的插入点。

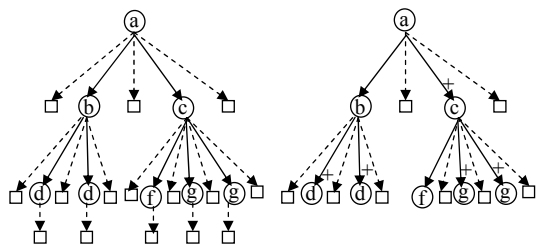


图 2.1: 数据模式

2) 更新模式

在实际中，各种结点插入的频率是不相同的。根据结点插入的频率，可以为每一个结点赋一个插入权重。权重越高，表示插入的频率越高，插入的可能性越大。我们把结点的插入权重叫做更新模式 (Update Pattern)。就编码更新这个问题而言，更新模式比数据模式更能有效地预留编码空间，因为数据模式只是告诉我们可以或者不可以，而更新模式更具体地告诉我们大概插入多少。因此，如果有更新模式支持，预留算法将会更行之有效，更新代价会更小。

3) 批量更新

在数据仓库的应用中，数据是在某个时间段批量插入的。类似地，在 XML 数据中，这种情况同样存在。这时，对编码的更新是批量的。关键问题是给新插入的子树分配编码的顺序。一个可能的方法是按照数据插入的顺序，先插入的子树先分配编码；或者按照插入的数据的前序遍历顺序，前序遍历靠前的新插入子树先分配编码。但是，我们将会看到，这些方法最大的问题是它会带来重复编码，即一棵子树已经给新分配了编码，后来由于别的子树，再一次更新它的编码。在批量更新的情况下，如何避免重复编码，减少更新代价，也是一个重要的问题。

论文将就这三个问题逐步深入地展开讨论。为了简单起见，本文使用 DTD 作为模式信息；不失一般性，本文简单地认为 XML 数据是一棵由结点组成的树；考虑到删除并不影响编码的更新，本文只集中讨论数据插入的情况，而且不考虑插入可选结点的情况；

3. 编码空间的预留

在扩展前序编码方法中，通过将 $\langle \text{order}, \text{size} \rangle$ 对中的 size 设得比实际的 size 大来预留编码空间。为了表示区别，编码中的 size 值，叫做 regionSize ，结点的实际大小，叫做 actualSize 。在这一节里，从简单到复杂，我们分别讨论模式独立的数据，有数据模式的数据和有更新模式的数据的编码预留方法。

3.1 模式独立数据的预留算法

模式独立的数据，对新插入子树的类型和插入的位置没有任何限制。在这种情况下，我们采取“平均预留”策略：把编码空间平均预留到所有可能的插入位置。为了最大限度地利用编码空间，显然，目标树的根结点的 regionSize 应该被赋值为整个编码空间的大小。比如，如果 regionSize 的长度是 4 个字节，那么根结点的 regionSize 应该是 2^{32} 。

平均预留策略是把编码空间平均预留到所有可能的插入位置。

定理 3.1 一棵模式独立的 XML 树，假设它的 size 是 s ，则插入位置的个数是 $2s-1$ 。

根据平均预留策略，在图 2.1 (a) 中，存在 15 个插入位置，假设最大编码空间大小 $\text{maxSize}=100$ ，我们将为每个插入位置预留 $(100-8)/15=6.13$ 的空间。模式独立数据的预留算法很简单，限于篇幅，本文略去这个具体算法。在实际计算 order 和 size 的时候，我们采用浮点数，只是在最后赋值的时候取整，这样就能有效的避免计算中的累计误差。

3.2 基于数据模式的预留算法

基于数据模式的 XML 数据，对新插入子树的类型和插入位置是有限制的。我们可以预测新插入子树与它的兄弟子树在结构上是相似的。所以，我们的策略是为一个插入位置预留的空间大小，等于该插入位置两旁的兄弟结点大小的一个倍数。为了描述我们的算法，首先引入一个概念——预留因子（reserving factor）。

定义 3.1: 一个 size 为 s 的结点，如果它的预留因子是 α ，则它对父亲的 regionSize 的贡献是 $s * \alpha$ 。

假设一个结点的预留因子是 α ，它对父结点的贡献是 $\alpha * regionSize$ ，其中， $(\alpha - 1) * regionSize / 2$ 作为预留空间，分别预留在了该结点的左右两边的位置。以图 2.1 (b) 中的树为例，我们假设所有的可重复结点 c, d_1, d_2, g_1 和 g_2 都具有相同的预留因子 α ，则 d_1 对 b 的 regionSize 的贡献是 α ， b 的 regionSize 是 $1 + 2 * \alpha$ 。类似地， c 的 regionSize 是 $2 + 2 * \alpha$ ； a 的 regionSize 是 $1 + (1 + 2 * \alpha) + (2 + 2 * \alpha) * \alpha = 2 + 4\alpha + 2\alpha^2$ 。现在我们给定最大编码空间 $maxSize = 100$ ，则可以得到方程：

$$2 + 4\alpha + 2\alpha^2 = 100 \text{-----} \langle \text{方程 1} \rangle$$

解得 $\alpha = 6.07$ 。 $s_0 + s_1\alpha + s_2\alpha^2 + \dots + s_n\alpha^n = maxSize$ 。

求解预留因子的关键是得到上述方程的系数 s_i 。

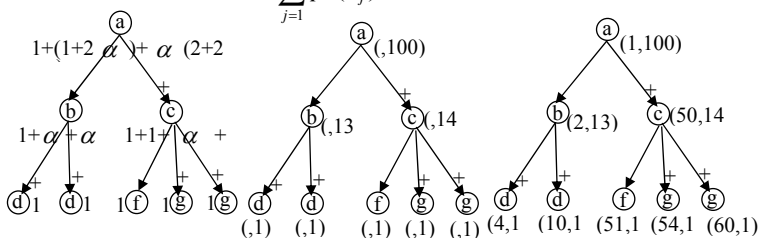
定义 3.2 (可重复结点的层次): 一个可重复结点的层次是从树根到这个结点的路径上的所有可重复结点的个数。

如在图 2.1(b)中 c 的可重复层次是 1，而 g_1 的是 2， a 的是 0。

定义 3.3 (可重复结点的本层次大小 (pure size)): 假设 e 是一个可重复结点，size 为 s ，而 d_1, d_2, \dots, d_n 是 e 的子孙可重复结点，且它们的都处在 e 的下一个可重复层次，它们的 size 分别是 s_1, s_2, \dots, s_n 。则 e 的本层次大小是 $ps(e) = s - (s_1 + s_2 + \dots + s_n)$ 。

c 的 size 是 4， c 的后继中 g_1 和 g_2 都是它下一级的可重复结点，且 size 都是 1，则 $ps(c) = 4 - 1 - 1 = 2$ 。在这棵树中，可重复级别是 1 的结点的本层次大小之和等于 2，恰好等于方程 $\langle 1 \rangle$ 中一次项的系数。不难看出：

定理 3.2: 假设 e_1, e_2, \dots, e_n 是目标树中所有可重复层次为 i 的结点。在求解预留因子的方程中，第 i 次项的系数 S_i 等于 $\sum_{j=1}^n ps(e_j)$ 。



(a) 计算预留因子 (b) 设置 size 的值 (c) 设置 order 的值

图 3.2 基于模式数据的空间预留算法

这个定理的证明此处从略。在求出 \langle 方程 1 \rangle 的各项系数后，我们采用二分法求解 α 。计算出预留因子后，给结点进行编码过程就比较简单了。请参照图 3.2。

3.3 基于更新模式的预留算法

数据模式只是告诉我们某一个插入位置可以或者不可以插入一棵子树，却不能告诉我们插入多少。因此，每一个可重复结点的预留因子都是 α 。更新模式却能准确地反映每一个插入位置可能插入新的子树的概率。根据更新模式，每一个可重复结点的预留因子就有了一个权重。

更新模式可以由统计数字获得。假设在一段时期内，结点类型 $E_1, E_2, E_3 \dots E_n$ 的结点分别被插入了 $p_1, p_2, p_3 \dots p_n$ 次 ($p_1 \leq p_2 \leq p_3 \leq \dots \leq p_n$)。结点类型 $E_1, E_2, E_3 \dots E_n$ 将被赋一个插入权重 w ， $w_1 = 1, w_2 = p_2/p_1, w_3 = p_3/p_1, \dots, w_n = p_n/p_1$ 。

假设预留因子是 α ，对于一个插入权重为 w 的结点来说，它对父结点的贡献是 $w * \alpha * regionSize$ ，其中， $(w * \alpha - 1) * regionSize / 2$ 作为预留空间，分别预留在了该结点的左右两边的位置。

同样地，为了求解预留因子 α ，需要构造方程：

$$s_0 + s_1\alpha + s_2\alpha^2 + \dots + s_n\alpha^n = maxSize$$

与 3.2 节的分析类似，不难看出：

定理 3.3: 假设 e_1, e_2, \dots, e_n 是目标树中所有可重复层次为 i 的结点。对于 $e_j (1 \leq j \leq n)$ ，从根结点到 e_j 的路径上的可重复结点的插入权重分别为 w_1, w_2, \dots, w_i 。令 $W_j = \prod_{k=1}^i w_k$ 。在求解预留因子的方程中，第 i 次项的系数 S_i 等于 $\sum_{j=1}^n ps(e_j) W_j$ 。

在有更新模式下的求解预留因子算法，设置 regionSize 算法和设置 regionOrder 算法均与 3.2 节的算法类似，限于篇幅，这里不再详述。

在更新模式下的求解预留因子算法，设置 regionSize 算法和设置 regionOrder 算法均与 3.2 节的算法类似，限于篇幅，这里不再详述。

4. 编码更新策略

在第三节中，我们针对模式独立、基于数据模式和基于更新模式的三种不同类型的数据讨论了不同的编码空间的预留方法。当预留空间不足以容纳新插入的子树时，就需要调整已有结点的编码，以腾出空间来容纳新的子树。重新编码的目标是能容纳新插入的子树，而且更新代价最小。所谓更新代价，是指在新子树插入后为它编码的过程中需要重新编码的结点的个数。在这一节里，我们从简单到复杂，分别讨论单次插入和批量插入情况下的编码更新策略。

4.1 单次插入的编码更新策略

回忆前面提到的插入位置的定义 $\langle parent$ ，

childIndex>。我们假设现在在插入位置 <p, t> 刚插入了一棵子树；p 是新子树根的父亲结点，它的所有孩子结点依次为 c(0),c(1),...,c(t),...,c(n)，c(t)就是刚插入子树，为了描述方便，我们还假设 p 有两个虚拟的孩子结点 c(-1)和 c(n+1)，令

c(-1).regionOrder=p.regionOrder;
c(n+1).regionOrder=p.regionOrder+p.reginSize;
c(-1).regionSize= c(n+1).regionSize=0;
c(-1).actualSize= c(n+1).actualSize = 0。

在 c(t)子树插入完成后，我们需要为 c(t)子树进行编码，这时最简单的情况就是 c(t)处的预留空间大小超过 c(t).actualSize，我们不必对其他结点重新编码就可以完成新子树的编码工作（判断条件是 c(t+1).regionOrder-c(t-1).regionOrder-c(t-1).regionSize ≥c(t).actualSize）。但是如果预留空间不够用，我们就需要对部分结点重新编码。这件事情可以分成两步：

步骤 1：找出需要重新编码的结点集合。

这时又可以分为两种情况：

情况一： p.regionSize ≥ p.actualSize;

请注意，这时 p.actualSize 包含 c(t).actualSize。显然，在这种情况下，p 结点不需要重新编码就可以完成这次插入的编码更新。这时存在着这样的 p 的序列的孩子列表 c(i),...c(t),...c(j)，其中 -1<i≤t 且 t≤j<n+1。这个列表满足：

$$c(j+1).regionOrder - c(i-1).regionOrder - c(i-1).regionSize \geq \sum_{k=i}^j c(k).actualSize$$

因此，我们只要对整个列表中的结点的子树进行编码即可，它的更新代价为

$$Renumbering\ Cost(i, j) = \sum_{k=i}^j c(k).actualSize - c(t).actualSize$$

对于 c(t)，可能存在许多个这样的列表，为了最小化更新代价，我们选取其中代价最小的一个列表作为重新编码的列表。

情况二： p.regionSize < p.actualSize;

对于这种情况，我们必须在 p 结点子树以外的地方寻找更多的预留空间。我们采用了一种巧妙而且有效的方法：将 p 作为新插入的子树，然后试图给它进行编码，这就回到了最开始的情况下在重新判断。

经过步骤 1 的处理，我们得到一个顺序的孩子队列，接下来是为这个队列重新编码。

步骤 2：为更新队列进行编码

确定编码空间之后，只剩下如何为更新队列编码的问题。考虑到可能的插入，在对队列完成编码的同时需要在它的内部和两旁作编码空间预留。最好的方法是能够调用第 3 节中的预留算法，给更新队列重新预留空间。为此，我们引入一个特殊的虚结点

virtualParent 作为更新队列中的结点的父结点，然后对这棵树编码。最后，删除虚结点 virtualParent，将更新队列还原到 p 下面的正确位置并且保持它们的编码不变；这就完成了对更新队列的重新编码（c(t)包含在队列中，所以它的子树的编码也随之完成）。

4.2 批量插入的编码更新策略

批量插入的编码更新策略，可以看作是单次插入的累加。一个简单的方法是按照插入的顺序，给新插入的子树编码；或者按照前序遍历的顺序给新插入子树编码。但是，不恰当的顺序将会导致重新编码，因而增加了编码更新代价。导致重新编码的原因有三个：

- 1) 某棵子树被分配了编码，后来由于祖先结点的原因，被重新编码。
- 2) 某棵子树被分配了编码，后来由于兄弟结点的原因，被重新编码。
- 3) 当前父结点已经不足以容纳新的子树，根据 4.1 节的策略，父结点被当作新插入的子树处理，导致父结点的兄弟结点被重新编码。

在处理批量插入的时候，为了避免重复编码，我们需要三个步骤：预处理，恰当的顺序和改进的搜索更新队列的算法。

步骤 1：检查新插入子树的父结点能否容纳新插入的子树，如果不能，把父结点当作新插入结点。这样做的目的是先把情况 3 去除，保证每个新插入子树的父结点都能容纳新插入的子树。

步骤 2：按照广度优先的顺序对批量插入的子树进行排序。这样可以避免情况 1 的发生，因为按照广度优先的顺序，祖先结点总是先于子孙结点被处理。

步骤 3：改进的搜索更新队列算法用来避免情况 2。在单个新插入子树的处理中，我们只为该新插入的子树寻找更新队列。为了避免情况 2，我们需要为同一个父结点下的所有新插入子树一起寻找更新队列。寻找的结果是若干条不交叉的更新队列，它们包含了该父结点下所有新插入子树。

经过这三个步骤，可以完全避免重复编码。

5. 实验

考虑到不同的 XML 数据有特定的 tree 结构，它们在实验中的表现也会不一样，我们在实验时选取了两个特征突出的数据集 Xmark[7]和 Shakespeare[11].

5.1 实验的基本过程和系统参数

首先我们列出可能影响更新代价的系统参数：

表 5.1 实验的系统参数

参数名字	说明	默认值
ST	初始目标树大小	33140
UR	最大更新比	220%
LEN	编码长度	64

ST 是初始的目标树的大小 (initsize of the target tree), LEN 是编码的长度, 它是 $\langle \text{order}, \text{size} \rangle$ 中 order 和 size 的长度之和, 它也直接确定了编码空间的大小。如 $\text{LEN} = 64$, 则 order 和 size 的取值范围都是 $(0 \sim 2^{32} - 1)$, 即编码空间大小为 2^{32} 。UR, 最大更新比, 是指在所有插入完成后目标树的大小与初始大小 ST 的比。在以后的各个图表中, 系统参数如果没有特别说明, 都取默认值。

下面是在实验过程中一次数据插入过程的描述, 它由以下的三个步骤组成:

首先是新插入子树 (inserted subtree) 的生成。我们用一个称为源树的 XML 文档, 用来生成新插入的子树。它与目标树是由同一个 XML 数据集生成的, 所以它们具有相同的模式定义。接着是插入位置的选取。插入位置由对 $\langle \text{parent}, \text{childIndex} \rangle$ 表示。最后一步, 把新插入子树插入到插入位置。这在第 4 节已有详细讨论, 这里不再多加说明。实验时, 我们不断的重复以上的三个步骤, 直到某一次插入完成后目标树的大小达到预定的值 $\text{ST} \times \text{UR}$, 然后计算更新代价。对于每次的实验, 我们都重复十次取平均作为最终的实验结果。

5.1 模式独立的数据的编码更新代价

由于没有模式信息的支持, 我们在插入时只能随机选择插入子树和插入位置。考虑在树根的附近插入一棵庞大的子树, 这时可能需要对整棵树重新编码, 在多次插入后总的更新代价必然是难以想象的。由于新插入子树和插入位置的完全随机性, 更新代价也呈现很大的随机性和不稳定性。图 5.1 反映了同一次实验反复进行了 10 次的更新代价的情况, 结果表现出极端的不稳定性。由于这种不稳定性, 我们下面的实验集中在基于数据模式和更新模式的数据。

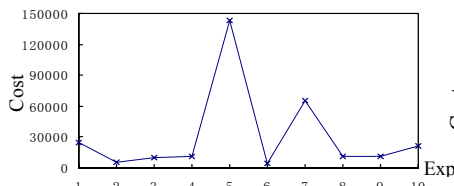


图 5.1: 模式独立数据的更新代价 (ST=17132)

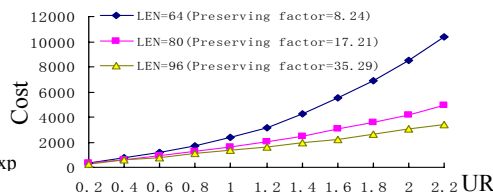


图 5.2.1 不同 LEN 下的更新代价

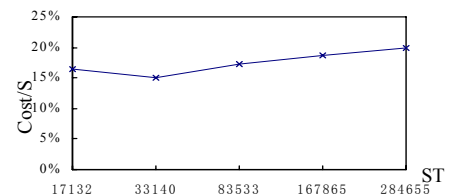


图 5.2.2 不同 ST 下的更新代价比 (LEN=80)

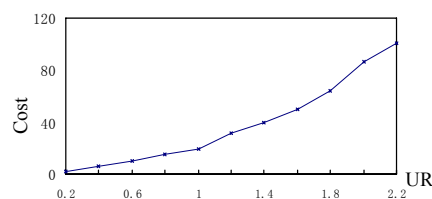


图 5.3: 文档特征的影响 (ST=28300, LEN=64)

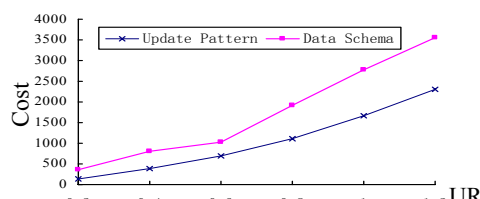


图 5.4 基于更新模式和基于数据模式的代价比较

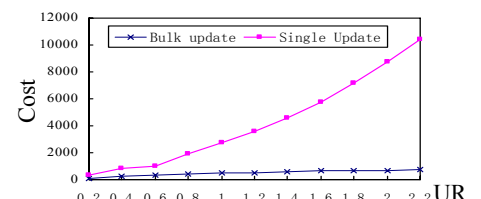


图 5.5 BulkUpdate 和多次 single update 的代价比较

5.2 基于数据模式的数据的编码更新代价

在基于数据模式的数据的编码中, LEN 和 UR 对更新代价的影响是很明显的。更大的 LEN 意味着更大的编码空间, 从而会降低重新编码的结点数; 而 UR 的增大必然导致更新代价的增大。图 5.2.1 说明了 LEN 和 UR 更新代价的影响。

再看图 5.2.2, 图中的横坐标是 ST, 纵坐标是更新比 = 更新代价 (Cost) / ST; 在实验时我们用不同的文档和不同的 LEN 下进行了多次实验, 图 5.2.2 中的数据只是其中的一组。可以看出, 图中更新比的值基本上固定在 18% 左右, 可以认为 ST 的大小对更新比的影响很小。

为了说明我们的预留和更新算法的效果, 我们做了表 5.2 的统计, 表中的粗体字指对应情况发生的次数和占总次数的百分比。在实验中, 当 UR 达到 220% 时, 大概完成了 5280 次插入操作, 其中 80% 以上的插入都没有进行重新编码, 这从一定程度上说明我们的算法时高效的。

表 5.2 更新代价分析 (ST=33140, UR=220%)

		LEN=64	LEN=80	LEN=96
情况 1	Cost=0	4332(82%)	4629(87%)	4706(90%)
	Cost>0	417(8%)	195(4)	128(2)
	Cost	6753	2632	1514
情况 2	次数	531(10%)	473(9%)	431(8)
	cost	3953	2303	1891
总更新代价		10346	4935	3405

5.3 文档特征对更新代价的影响

在基于模式的数据前提下, 扁平的树结构比相对较深的树结构有好的多的表现。图 5.3 的实验的数据集是 Shakespeare, 树结构比较扁平, $\text{ST} = 28300$; 而图 5.2.1

的数据集是 Xmark, 树结构较深且包含递归定义的结点, ST=33140。它们初始的 ST 相差不大, 但是两个图中的蓝色曲线 (均对应 LEN=64 的情况) 指的更新代价相差有 100 倍! 这是因为在图 5.2.1 中, LEN=64 时的预留因子是 8.24, 而在图 5.6 中, 同样 LEN=64, 但是预留因子达到 20.59。

5.4 基于更新模式的数据的编码更新代价

在具有更新模式的情况下, 预留空间将只分配给具有更新权重的节点的子树, 而且根据各自的更新权重的不同, 不同的可重复节点得到的预留空间也会不同。在 Xmark 数据集中, 考虑到实际应用中绝大部分的插入都几种在 item 等几种节点的子树的需求, 我们设计了有更新模式与没有更新模式的更新代价对比的实验, 图 5.8 是实验的结果, 在实验中我们的更新模式设定是: open_auction : 30, closed_auction : 30, person : 40, item : 50, bidder : 50。而其他的可重复节点我们设定更新权重均为 0。可以看出, 相对于没有更新模式的情况, 更新代价有了大幅度的削减。

5.5 批量更新的代价

大量的单个子树更新带来了大量的重复编码, 使编码更新代价很大。采用批量更新, 避免了重复编码, 相比之下, 编码代价小了很多。图 5.5 表示各种更新率下单次批量更新和多次单棵子树更新的代价比较, 其中单棵子树的更新采用的是基于数据模式的预留方法。比如, 在 UR=1.0 的情况下, 代价是 504, 表示一次批量插入 1.0×33140 个结点, 然后批量更新, 编码更新的结点数是 504。可以看出, 批量更新的代价相比同更新率的多次单棵子树更新的代价, 要小得多。

6. 结论和展望

这篇文章集中讨论了关于 XML 文档的扩展前序编码的更新问题, 根据不同的数据和应用, 提出了一整套预留和更新算法。实验的结果表明: 在各种不同的情况下, 这几个算法都表现出较好的性能。这是在 XML 文档编码更新问题上进行的有效尝试。

在这篇文章的基础之上, 在 XML 数据的编码更新领域还有很多进一步的工作可以展开。比如, 由于 XML 数据的编码是为了查询处理的需要, 在编码要更新时, 应该怎样更新才能够不影响或者尽可能小地影响查询处理的操作? 这些都是值得思考的问题。

7. 参考文献

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In Proceedings of the 18th ICDE. San Jose,

California, February 2002.

- [2] P. F. Dietz. Maintaining order in a linked list. In Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, pages 122-127. San Francisco, California, USA, May 1982.
- [3] T. Grust. Accelerating XPath location steps. In Proceedings of the 28th ACM SIGMOD, pages 109-120. Madison, Wisconsin, USA, June 2002.
- [4] D. D. Kha, M. Yoshikawa, S. Uemura. An XML Indexing Structure with Relative Region Coordinate. In Proceedings of the 17th ICDE, pages 313-320. Heidelberg, Germany, April, 2001.
- [5] Y. K. Lee, S. J. Yoo, K. Yoon. Index structures for structured documents. In ACM First International Conference on Digital Libraries, pages 91-99, Bethesda, Maryland, March 1996.
- [6] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In Proceedings of the 27th VLDB, pages 361-370. Roma, Italy, September 2001.
- [7] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, R. Busse. XMark: A Benchmark for XML Data Management. In Proceedings of 28th VLDB, pages 974-985. Hong Kong, China, August 2002.
- [8] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In Proceedings of the 27th ACM SIGMOD, pages 425-436. Santa Barbara, California, USA, May 2001.
- [9] W. Wang, H. Jiang, H. Lu, J. X. Yu. Containment Join Size Estimation: Models and Methods. To be published in Proceedings of the 29th ACM SIGMOD, pages 145-156. California, USA, June 2003.
- [10] W. Wang, H. Jiang, H. Lu and J. X. Yu. PBiTree Coding and Efficient Processing of Containment Join. In Proceedings of 19th ICDE, pages 391-402. Bangalore, India, March 2003.
- [11] Shakespeare dataset. Available at <http://sunsite.unc.edu/pub/sun-info/standards/xml/eg/>
- [12] Query 1.0 and XPath 2.0 Data Model. W3C Working Draft 15 November 2002. Available at <http://www.w3.org/TR/query-datamodel/>