Dynamically Updating XML Data: Numbering Scheme Revisited

Jeffrey Xu Yu ♮ *Daofeng Luo* † *Xiaofeng Meng* † *Hongjun Lu* ‡
♮ Chinese University of Hong Kong, Hong Kong, China, yu@se.cuhk.edu.hk
† Renmin University of China, Beijing, China, xfmeng@mail.ruc.edu.cn
‡ Hong Kong University of Science & Technology, Hong Kong, China, luhj@cs.ust.hk

Abstract

Almost all existing approaches use certain numbering scheme to encode XML elements to facilitate query processing when XML data is stored in databases. For example, under the most popular region-based numbering scheme, the starting and ending positions of an element in a document are used as the code to identify the element so that the ancestor/descendant relationship between two elements can be determined by merely examining their codes. While such numbering scheme can greatly improve query performance, renumbering large amount of elements caused by updates becomes a performance bottleneck if XML documents are frequently updated. Unfortunately, no satisfactory work has been reported for efficient update of XML data. In this paper, we first formalize the XML data update problem by defining the basic operators to support most XML update queries. We then present a new numbering scheme that not only requires minimal code-length in comparison with existing numbering schema but also improves update performance when XML data is frequently updated at arbitrary positions. The fundamental difference between our new scheme and existing ones is that, instead of maintaining the explicit codes for elements, we only store the necessary information and generate the codes when they are needed in query processing. In addition to present the basic scheme, we also discuss some optimization techniques to further reduce the update cost. Results of a comprehensive performance study are provided to show the advantages of the new scheme.

1 Introduction

XML databases are subject to change. In comparison to a large number of reported studies on XML query processing, there are few reported studies on XML updates. The XML database updates mainly involve two things: (i) physically updating the XML data and the indexes associated with, and (ii) renumbering the codes assigned to nodes, in order to maintain ancestor-descendant relationships for efficient query processing. We observe that the cost of the former is similar under the same storage model, but the cost of the latter can vary significantly, when a different numbering scheme is used. In this paper, we study XML updates and focus ourselves on the latter issue – XML numbering scheme update.

As a popular numbering scheme, the region-based numbering scheme [18, 17] encodes the starting and ending positions of an element in a document to identify the element so that the ancestor/descendant relationship between two elements can be determined by merely examining their codes. While such a numbering scheme can greatly improve query performance, renumbering large amount of elements caused by updates becomes a performance bottleneck if XML documents are frequently updated. The problem we are facing is challenging because it requires us to find a numbering scheme that can a) minimize XML query processing cost, b) minimize

renumbering costs when XML updates occurs at any positions, and c) require minimal space for encoding.

In this paper, we show such a numbering scheme does exist that satisfies the above three requirements. The main contributions of this paper are summarized below. First, we formalize the XML update problem by defining three primitive operations. Second, we propose a new prefix-based numbering scheme, called P-PBiTree. The unique features of P-PBiTree are addressed below.

- The P-PBiTree encodes elements with less bits in comparison with other prefix-based numbering schema [18, 17, 12, 7] as well as region-based schema [4, 11, 6, 8, 15], in terms of both maximum code-length and the total of code-lengths assigned.
- The dual property of P-PBiTree allows it to support both sort-based and hash-based containment joins.
- The P-PBiTree also minimizes the renumbering cost when XML updates occur, which is achieved using the following techniques: i) we support code-preserving in P-PBiTree code that reduces the probability of renumbering all siblings and their descendants when updates occur at arbitrary positions, ii), in addition, we propose a novel two-level P-PBiTree, denoted 2P-PBiTree, with which, when an update occurs, we only need to update nodes in a small cluster rather than the whole tree. Two clustering strategies will be discussed. The 2P-PBiTree code can be easily converted to P-PBiTree code on the fly when extracting elements from the XML data tree.

We conduct extensive performance studies among different numbering schema in terms of updates in various situations. The result verifies that P-PBiTree outperforms existing numbering schema significantly in terms of storage space consumption and updating performance.

The remainder of the paper is organized as follows. Section 2 gives the basic operators to support XML update queries. Section 3 discuss top-down/bottom-up propagation of existing schema. Section 4 introduces our P-PBiTree and 2P-PBiTree scheme and the techniques to minimize XML updating costs. Section 5 reports our experimental results. Section 6 concludes this paper.

2 XML Update Handling

XML documents can be modeled as XML data trees [3, 2]. In addition to the operators addressed in [3, 2], recently, a tree algebra called TAX was proposed in [5], as a natural extension of relational algebra, with a small set of operators. TAX is complete for relational algebra extended with aggregation, and can express most queries expressible in popular XML query languages. In this paper, instead of XML data retrieval, we focus ourselves on XML updates, and in particular, the maintenance of a numbering scheme when XML updates occur. We emphasize on the structural changes of XML data trees, e.g., inserting/deleting a subtree into/from an existing XML data tree. For simplicity, we define XML data tree as an ordered tree, T, and specify two primitive insertion and deletion operations for structural changes and one primitive update operation for content changes, which can all possibly cause renumbering the codes assigned. Let T and T' be two subtrees and p be a path, the three operations are given below.

- **Insert(T, p, T')**: insert an order tree T' into an order tree T, as specified by a path p ($\in T$).

- **Delete(T, p)**: delete a subtree from an order tree T, as specified by a path p ($\in T$). The deletion can return the deleted tree as a return value.
- **Update(T, p, C)**: update the content of a node to be C as specified by a path p ($\in T$).

Here, a path p can be considered as a sequence of nodes (*ref*) with an indicator to indicate the i-th child of the ref. For simplicity of showing the updating costs, without loss of generality, here, a path is a sequence of nodes and can be specified as $p = \vec{u}_0.\vec{u}_1.\vec{u}_2.\cdots.\vec{u}_{n-1}.\vec{u}_n$ where $\vec{u}_0$ represents the root of the tree, and $\vec{u}_i$, for $i > 0$, represents its position among its siblings of its parent ($\vec{u}_{i-1}$) from left to right starting from 1. For insertion, $\vec{u}_n$ is the position to insert the new subtree. For deletion and update, $\vec{u}_n$ specifies an existing node to be updated. The three operations can be used jointly to support general tree updates such as moving a subtree from a position to another by combining deletion and insertion. The general update operators such as rename are not the focus of this paper [14]. Also, unlike [14] which addresses XML updates when data are mapped onto RDBMSs, we focus on the renumbering costs associated with XML data trees.

3 Numbering Schema: Top-Down/Bottom-Up Propagations

We use the similar notion of numbering scheme defined in [4]. A numbering scheme is a pair $\langle p, L\rangle$, where L is a numbering function that assigns a unique code to a node v in a tree T, and p is a predicate such that, for every two nodes, v and u, in a tree T, $p(L(v), L(u))$ is true iff v is an ancestor of u.

Several numbering schema were proposed: region-based [18, 17, 12, 7], prefix-based [4, 11, 6, 8, 15], and k-ary complete-tree-based [10, 16]. We summarize them in brief below, and give our observations on their top-down/bottom-up propagation patterns regarding renumbering when XML updates occur.

3.1 Region-based Numbering Schema

[18, 17] proposed a region based numbering scheme for a node u using $L(u) = \langle u_s, u_e\rangle$ where u_s and u_e are the offset of the start and end of the node u in a XML data tree such that $u_s \leq u_e$. Given two nodes $L(v) = (s_v, d_v)$ and $L(u) = (s_u, d_u)$ in a tree, the predicate $p(L(v), L(u))$ is true, i.e., v is an ancestor of u, iff $v_s \leq u_s \leq u_e \leq v_e$. We call it *absolute region-based numbering scheme*, and denote the absolute region-based code for a node u as $L_{AR}(u)$.

[12] proposed a variant in the form of $L(u) = \langle u_o, u_s\rangle$, where u_o and u_s are a given traversal order and size of node u. Here, for two given nodes v and u of a tree T, v is an ancestor of u iff v occurs before u in the preorder traversal of T and after u in the postorder traversal. In other words, let $L(u) = \langle u_o, u_s\rangle$ and $v = \langle v_o, v_s\rangle$, then a node v is u's ancestor iff $v_o < u_o$ and $u_o + u_s \leq v_o + v_s$. The size of v, v_s, must be greater than or equal to the sum of all sizes of its direct child u such as $v_s \geq \Sigma_u u_s$. We call it a *durable region-based numbering scheme*, because it reserves additional number space for elements. We denote the durable region-based code for a node u as $L_{DR}(u)$. The durable region-based scheme achieves high XML query processing performance, and is supposed to reduce the high update cost due to overflow compared to the absolute region-based numbering scheme.

In [7], a *relative region-based numbering scheme* was proposed to deal with the XML update problem, where each node v is coded using the relative offset to its parent element (o_1, o_2) where o_1 (o_2) is the number of bytes from the starting byte of its parent to the starting (the end

byte) of itself. The relative region-based code can be converted into the absolute region-based code. But, the main drawback of the relative region-based numbering scheme is that the query processing cost significantly increases because all the ancestors along the path to the node in question have to be read to determine the absolute region-based code of the node. We do not discuss the relative region-based numbering scheme in this paper, because it supports XML update at the high expense of supporting query processing.

Remarks on Renumbering Costs

We discuss XML update costs in terms of renumbering for the two region-based numbering schema, L_{AR} and L_{DR}, in worst case.

For the absolute region-based numbering scheme $L_{AR}(u) = \langle u_s, u_e \rangle$, first we assume the path p specifies a leaf node $\vec{u}_n$. In such a case, the ordered tree T to be updated is divided into two parts by the path. Therefore, the codes assigned to all nodes in the right part tree and on the path p need to be updated. If the given path $p = \vec{u}_0.\vec{u}_1.\vec{u}_2.\cdots.\vec{u}_{n-1}.\vec{u}_n$ specifies a non-leaf node, we can divide the tree using the leftmost path of the subtree rooted at the immediate right sibling of $\vec{u}_n$ if such a sibling exists. Otherwise, we can do it using the leftmost path of the subtree rooted at the immediate right sibling of $\vec{u}_{n-1}$. Repeatedly if needed, such a path to a proper leaf can be found in order to divide a tree into two parts in general. The worst case is to update the leftmost leaf node in a tree. All nodes need to be renumbered. The renumbering can be caused by either inserting a node (structural change) or updating a node (content change).

For the durable region-based numbering scheme $L_{DR}(u) = \langle u_o, u_s \rangle$, where u_o and u_s are traversal order and size of node u, we consider two cases: when the updates are caused by content changes (u_s) and when the updates are caused by reorder (u_o). For the former, for a $p = \vec{u}_0.\vec{u}_1.\vec{u}_2.\cdots.\vec{u}_{n-1}.\vec{u}_n$, it needs to update the codes for the nodes on the path and all their right siblings of any $\vec{u}_i$ on the specified path. For the later, it needs to reorder all the nodes in the traversal order after the node to be inserted. L_{DR} significantly improves the update costs over L_{AR}, because it does not need to update the whole right part tree. But, like L_{DR}, the worst case for L_{AR} is the leftmost leaf node in a tree.

Remark 1 *(**Bottom-Up Propagation**) The region-based approaches mainly use the space constraint either a pair of start and end or a pair of a traversal order and size. It propagates numbering updates in a bottom-up fashion, because the 'space' needs to cover all its subspaces.*

3.2 Prefix-based numbering

Prefix-based numbering schema were studied in different contexts [4, 11, 6, 8]. The main idea of a prefix-based numbering scheme is as follows. Suppose a node v, coded $L(v)$, has k children: $u_1, u_2, \cdots, u_k$. Then, the code of a child u_i is encoded as $L(u_i) = L(v).L'(u_i)$ where $L'(u_i)$ is a unique *child code* of u_i among its sibling and "." is a concatenation operator. Unless otherwise stated, in the following, the concatenation operator "." does not appear as part of code. A predicate $p(L(v), L(u))$ is true iff $L(v)$ is a prefix of $L(u)$.

The widely-known prefix-based numbering is the Dewey decimal notation [9] which was studied in [15]. Here, given k children of a node v coded $L(v)$, $u_1, u_2, \cdots, u_k$. The code of u_i is simply $L(u_i) = L(v).i$. Note: "." will appear as a part of code, e.g., 1.2.3.

In [4], two prefix-based numbering schema were proposed to assign a code to the i-th child, u_i, of a node, v. The first is *one-bit growth* approach. Here, the first child's code is $L(u_1) = L(v).0$,

the second child's code is $L(u_2) = L(v).10$ and the third child's code is $L(u_3) = L(v).110$. Therefore, the i-th child's code is to repeat '1' for $i-1$ times with a '0' attached at the end. The second is *double-bit growth* approach. Suppose the u_i's code is $L(v).L'(u_i)$ where $L(v)$ is its direct parent code. The double-bit growth approach assigns the children as follow. $L'(u_1) = 0$, $L'(u_2) = 10$, $L'(u_3) = 1100$, $L'(u_4) = 1101$, $L'(u_5) = 1110$, $L'(u_6) = 11110000$, $\cdots$. That is, to obtain $L'(u_{i+1})$, it increment the binary code represented by $L'(u_i)$ and if the representation of $L'(u_i) + 1$ consists of all ones it double its length by adding a sequence of zeros. For a given node u, we denote its one-bit growth code and double-bit growth code as $L_{OB}(u)$ and $L_{DB}(u)$, respectively.

Remarks on Renumbering Costs

For the prefix-based numbering schema, due to the nature of prefix, the whole subtree needs to be updated if the code of the root of the subtree is changed, which occurs when overflow occurs for the bits assigned to the prefix. We explain it below. Suppose a node v has k child nodes, $u_1, u_2, \cdots, u_k$ in order, and suppose a new node is about to be inserted at the i-th child of v. Due to the ways of assigning bits as prefix to a node, when a new node is inserted as the i-th child, the child nodes originally at $u_i, u_{i+1}, \cdots, u_k$ need to be shifted by one position. Therefore, all nodes in the subtrees rooted at $u_i, u_{i+1}, \cdots, u_k$ need to be updated. Content updates do not cause any renumbering for prefix-based numbering schema.

Remark 2 *(**Top-Down Propagation**) The prefix-based approaches propagate in a top-down fashion. When a prefix assigned to the root of a subtree is changed, then all its descendants need to be updated.*

Based on the two remarks, a region-based approach performs well when it updates nodes at the top portion of the tree, whereas a prefix-based approach performs well when it updates nodes at the bottom portion of the tree.

3.3 K-ary complete-tree-based numbering

A *k-ary complete tree scheme* was introduced in [10] which uses tree traversal order to determine the ancestor-descendant relationship between any pair of nodes in the tree. As reported in [12], one problem of this scheme is that it requires a large numbering space when the ary and height of the complete tree are getting large.

[16] proposed a numbering scheme, called PBiTree (Perfect Binary Tree). A PBiTree is a perfect binary tree, where each non-leaf node has two children and all leaf nodes are at the same level. Each node is encoded with a number (of the in-order of traversal of the tree). For a given node u of a perfect binary tree, its ancestor v at a given height h_v can be directly calculated by $\mathcal{F}(L(u), h_v) = 2^{h_v+1} \cdot \lfloor L(u)/2^{h_v+1} \rfloor + 2^{h_v}$. Hence, a node v at hight h_v is an ancestor of u iff $L(v) = \mathcal{F}(L(u), h_v)$.

We denote the PBiTree code for a node u as $L_P(u)$, and explain the way of coding in brief below. For any node u in the PBiTree, let l be the level of u (the root is at level 0) and α be the zero-based position index of element nodes from left to right at the level l, i.e. $\alpha \in [0, 2^l - 1]$, then the code of u is $L_P(u) = (\alpha, l)$, by which the in-order code of u can be easily computed as $(1 + 2\alpha) \times 2^{H-l-1}$, where H is the height of the tree. Note, H is recorded in a global variable and is not recorded as a part of the code for a particular node. When updates occur, if a node has more than two child nodes, the child nodes will be pushed down to a deeper level until this

level can hold all the child nodes. The PBiTree code has the top-down propagation property when XML updates occur.

PBiTree uses hash-based techniques to process containment join, and was reported in [16] outperforming the sort-based techniques used in the durable region scheme [12]. The PBiTree code can be easily converted into a region-based code and a prefix-based code. The reverse conversion is costly and requires extra information to be encoded, as indicated in [16].

3.4 Space Complexity

In [4], Cohen, Kaplan and Milo investigated the maximum length of a code, with and without clues, for dynamic numbering of trees when an insertion sequence is accompanied. Two types of clues were considered. When a node v is inserted, a subtree clue is to estimate on the final number of descendants of v, with a constant factor; and includes an estimate on the number of descendants of the future siblings of v. They studied region-based and prefix-based numbering schema, and showed that, $\Theta(n)$ with no clues, $\Theta(\log^2 n)$ with subtree clues and $\Theta(\log n)$ with sibling clues, for n insertions of nodes into a tree. PBiTree numbering scheme was not studied in [4]. In this paper, we show that there exists an equivalent prefix-based numbering scheme to PBiTree without any extra information to be coded. Indirectly, we show the space complexity addressed in [4] is applicable to PBiTree.

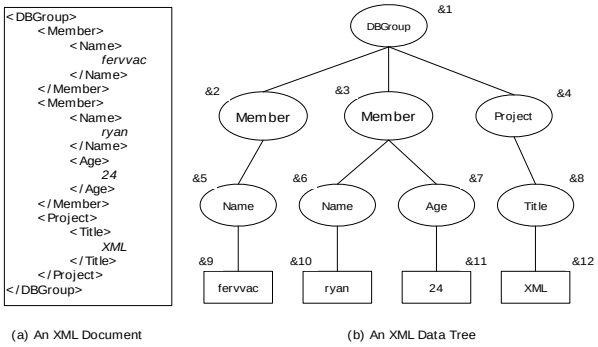


Figure 1: An XML document and its data tree given in [16]

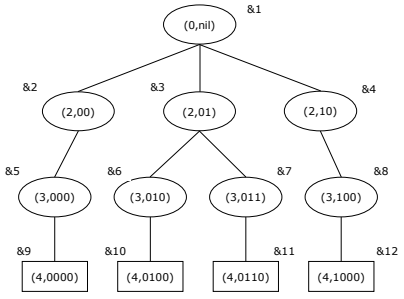


Figure 2: The P-PBiTree codes for Figure 1 (b)

4 P-PBiTree: A New Prefix-Based Numbering Scheme

In this paper, we propose a new prefix-based numbering scheme, called *prefix-based PBiTree* or in short *P-PBiTree*. Given a node v with k direct child nodes, $u_1, u_2, \cdots, u_k$. We assign a unique child code to u_i using m-bits such that $2^{m-1} < k \leq 2^m$, denoted $L'(u_i)$. Let the parent node v's code be $L(v)$, then the code of u_i is $L(u_i) = L(v).L'(u_i)$ where "." is a concatenation operator. The code of the root node is $\emptyset$. Let $length(u)$ denote the bit length of the code for

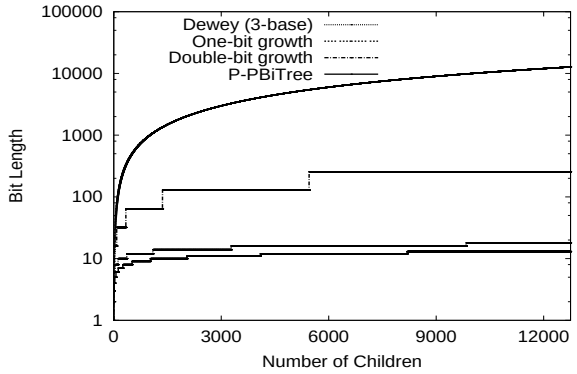


Figure 3: Bit Length Growth for Child Nodes (One-bit > Double-bit > Dewey > P-PBiTree)

node u. The new P-PBiTree numbering scheme is a pair $\langle L_{PP}, p\rangle$ where L_{PP} is the P-PBiTree code, and p is a predicate, such that $p(L_{PP}(v), L_{PP}(u))$ is true, i.e., v is an ancestor of u, iff $length(v) < length(u)$ and $L_{PP}(v)$ is a prefix of $L_{PP}(u)$. Note: it is not practical to count the bit length for a code. We keep the bit length as a part of the code for P-PBiTree as well as for the other prefix-based numbering schema.

$$\begin{aligned} L_{PP}(u) &= \langle u_l, u_p\rangle \\ L_{OB}(u) &= \langle u_l, u_p\rangle \\ L_{DB}(u) &= \langle u_l, u_p\rangle \end{aligned}$$

Here, u_p is the prefix-based code of u discussed above and u_l is the length of the corresponding code. We call u_p the *p-code* of prefix-based code, when the length is not concerned with. The prefix can be checked using a simple bit operation efficiently. We will discuss the main differences between P-PBiTree code and the other prefixed numbering schema.

Figure 2 shows the P-PBiTree codes for the XML data tree in Figure 1 (b). In Figure 2, the length of the root is zero. The root (`&1`) has three child nodes ($k = 3$). 2-bits are needed to encode the child-codes for the three children of the root, `00`, `01` and `10`. The p-code code of the three child nodes, `&2`, `&3` and `&4` are the same as their child-code, because the p-code of the root is empty. The P-PBiTree codes for `&1`, `&2`, `&3` and `&4` are then $\langle 0, \emptyset\rangle$, $\langle 2, 00\rangle$, $\langle 2, 01\rangle$ and $\langle 2, 10\rangle$, respectively. The node `&3` has two child nodes (`&6` and `&7`). Therefore, one more bit is needed. The P-PBiTree codes are $\langle 3, 010\rangle$ and $\langle 3, 011\rangle$, accordingly.

4.1 Space Consumption

As a prefix-based numbering scheme, P-PBiTree has the same space complexity as the other prefix-based approaches [4]. In addition, P-PBiTree performs the best in term of space consumption. We explain it below. Consider the p-code of a node v, coded $L(v)$, has k child nodes: $u_1, u_2, \cdots, u_k$. Then, the p-code of a child u_i is encoded as $L(u_i) = L(v).L'(u_i)$ where $L'(u_i)$ is a unique child code of u_i among its sibling and "." is a concatenation operator. The XMark benchmark [13] of 57M dataset setting uses an XML tree with the maximum degree of 12,750 (a node can have up to 12,750 child nodes). The bit length growths of child-code, ($L'(u_i)$ of $L(v).L'(u_i)$), assuming that $L(v)$ is the same, are shown in Figure 3, for the four prefix-based

numbering schema: Dewey-code[1], one-bit growth, double-bit growth and P-PBiTree. The bit length for the P-PBiTree is superior to all Dewey-code, one-bit growth and double-bit growth approaches in magnitude(s). Obviously, the longer the child-code is, the longer the prefix-based code is. It is worth noting that the bit length depends on both the hight of the tree and the maximum degree of any tree. The authors in [4] pointed out that the average depth of XML trees is low and the trees are balanced with relatively high degrees. As shown in Figure 3, our P-PBiTree numbering scheme can reduce the space significantly in particular when the number of child nodes is large.

The p-code of a node u_n specified by a path $p = \vec{u}_0.\vec{u}_1.\vec{u}_2.\cdots.\vec{u}_n$ is

$$u_p = L'(\vec{u}_0).L'(\vec{u}_1).L'(\vec{u}_2).\cdots.L'(\vec{u}_n)$$

where $L'(\vec{u}_i)$ is a child code. The length of p-code can vary. The question is how to allocate bits for recording the length, as a part of of the prefix-based code $L_{PP}(u_n) = \langle u_l, u_p \rangle$. There are two ways to handle the length component, using bit-unit or byte-unit. By name, it is to record the length in unit of bits or bytes. Consider the XMark benchmark of 57M dataset setting again where the hight of the tree is 12 and the maximum degree is 12,750. Suppose 13 bits are needed at max for a child code as shown in Figure 3. Then the bit-length of the p-code for u_n is $156 = (12 \times 13)$ at most. Suppose that we use one byte to record the u_l of the P-PBiTree code ($\langle u_l, u_p \rangle$) in bit-unit. The maximum bit length for a P-PBiTree code is 256. Suppose the height of the tree is 12 (as given in the XMark benchmark). Then 21 bits can be used for a child code, for $21 = \lceil 256 \div 12 \rceil$. The number of 21 implies that every node can have up to 2^{21} child nodes. When the bit-unit is not enough, we can use byte-unit with which we need to add a *terminator* bit '1' at the end of the p-code. For example, the P-PBiTree code (bit-unit) for the node `&7` is $\langle 3, 011 \rangle$. Its P-PBiTree code (byte-unit) is then $\langle 1, 011\underline{1}0000 \rangle$ where the last 1 bit $\underline{1}$ is the terminator. Suppose that we use one-byte to record u_l of the P-PBiTree code in byte-unit. The maximum bit length for a P-PBiTree code is $2,048 (= 256 \times 8)$. Suppose the height of the tree is 12 too. Every node can have up to 2^{170} child nodes where $170 = \lceil 2048 \div 12 \rceil$. Therefore, we can conclude that we only need one byte for recording the length part of the P-PBiTree code for most real applications.

Above, we showed that P-PBiTree consumes less than all Dewey-code, one-bit growth and double-bit growth prefix-based numbering schema. In fact, P-PBiTree code $L_{PP}(u) = \langle u_l, u_p \rangle$ consumes less even than any region-based numbering schema, which use 2 4-byte integers (64 bits). We will show it in our experimental studies.

4.2 P-PBiTree Updating

We show that P-PBiTree code is superior to the other prefix-based code, namely, Dewey-code, one-bit growth and double-bit growth, in terms of updating cost, for the following reasons. First, one-bit growth and double-bit growth perform the same in terms of renumbering cost, because they are designed to minimize the cost of appending a node as the last child of a node. The main idea is not to update any existing siblings. However, in real situations, insertions can occur at any positions. Therefore, when an insertion occurs at the i-th child of a node, then all

[1]We investigated several ways to encode Dewey decimal code, and found that the 3-based decimal representation is superior to other Dewey coding. Here, we use 2 bits for a 3-based digit, 0 (00), 1 (01), 2 (10), and "." (11). The decimal Dewey code 11.1 will be represented as 0100101101 ($11 = 1 \times 3^2 + 0 \times 3^1 + 2 \times 3^0$.

the subtrees of the right siblings need to be updated, which is costly. Dewey-code performs in a similar fashion like one-bit growth and double-bit growth. The numbering cost for the three Dewey-code, one-bit growth and double-bit growth are the same. Second, P-PBiTree allows *code preserving*. Suppose a node u has k child nodes where k is between 2^{m-1} and 2^m. By code preserving, we do not assign the k child codes consecutively. We preserve some child codes between every two child nodes. With code preserving, when a node is inserted at a position, we can minimize the cost of renumbering. That is, we do not need to renumber all its right siblings.

For example, a node has 5 child nodes, u_1, u_2, u_3, u_4 and u_5. P-PBiTree needs 3 bits for the child codes. Without code preserving, the child codes are $L'(u_1) = 000$, $L'(u_2) = 001$, $L'(u_3) = 010$, $L'(u_4) = 011$ and $L'(u_5) = 100$. When a new node is about to be inserted as the first child, all the five nodes need to be renumbered. With the code preserving, we assign the child codes for the 5 child nodes as $L'(u_1) = 000$, $L'(u_2) = 001$, $L'(u_3) = 011$, $L'(u_4) = 100$ and $L'(u_5) = 111$. When a new node is about to be inserted as the first child, we only need to shift the child code for the current u_1 and u_2 as $L'(u_1) = 001$ and $L'(u_2) = 010$, respectively, in order to give the new first child code as 000. We do not need to update the other siblings.

Furthermore, several code preserving strategies can be considered. One is to evenly assign the preserved codes between existing child nodes. The other is to preserve codes following DTD information. For instance, suppose the DTD for Figure 4 (a) is given as follows.

```
<!ELEMENT A(B,C+)>
<!ELEMENT B(D+)>
<!ELEMENT C(F,G+)>
```

Then, it is known that the root `a` can only have one `b` but many `c`'s. Also, in many cases, it is known the maximum elements can be inserted as a sub-element of the parent. In addition, when the bit is overflow, i.e., when a new element is inserted as the (k+1)-th child while $k = 2^m$, we can assign $m + n$ bits, for $n \geq 1$, rather than $m + 1$ bits, if it is known that more new child nodes will be inserted. It is very useful when dealing with bulk insertion.

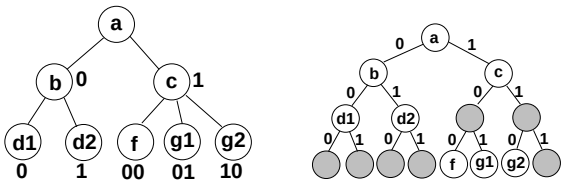
The code-preserving can be introduced in region-based numbering schema. But, because the region-based numbering schema propagate updates in a bottom-up fashion, the code-preserving at the upper level does not work efficiently. On the other hand, P-PBiTree propagates updates top-down, the code-preserving strategies work efficiently. The other three prefix-based numbering schema can possibly preserve code between child nodes. However, as seen in Figure 3, they already consume large space (bits) to encode, and they need even more space to support code-preserving, which is not practical.

4.3 P-PBiTree = PBiTree

In this section, we show that the dual property of the proposed P-PBiTree, as in fact it is equivalent to PBiTree [16]. We show an example in Figure 4 to explain. Figure 4 (a) shows a tree of hight 3 consisting of 8 nodes. The P-PBiTree child-codes are shown along with the corresponding nodes in the figure. Accordingly, their P-PBiTree codes are as follows. (Recall for a node u its P-PBiTree code is $L_{PP}(u) = \langle u_l, u_p \rangle$.) `a`: $\langle 0, \emptyset \rangle$, `b`: $\langle 1, 0 \rangle$, `c`: $\langle 1, 1 \rangle$, `d1`: $\langle 2, 00 \rangle$, `d2`: $\langle 2, 01 \rangle$, `f`: $\langle 3, 100 \rangle$, `g1`: $\langle 3, 101 \rangle$, and `g2`: $\langle 3, 110 \rangle$. On the other hand, Figure 4 (b) shows the corresponding PBiTree consisting of 15 nodes including 7 virtual nodes (shadow nodes). (Recall for a node u its PBiTree code is $L_P(u) = \langle l, \alpha \rangle$ where l is the level (The level of the root is 0) and α is the zero-based position index from left to right.) The PBiTree codes are shown below. `a`: $\langle 0, 0 \rangle$, `b`: $\langle 1, 0 \rangle$, `c`: $\langle 1, 1 \rangle$, `d1`: $\langle 2, 0 \rangle$, `d2`: $\langle 2, 1 \rangle$, `f`: $\langle 3, 3 \rangle$, `g1`: $\langle 3, 4 \rangle$, and `g2`: $\langle 3, 5 \rangle$.

We show that, given a node u, with two codes, $L_{PP}(u) = \langle u_l, u_p \rangle$ (P-PBiTree) and $L_P(u) = \langle l, \alpha \rangle$ (PBiTree), then $u_l = l$ and $u_p = \alpha$, and therefore $L_{PP}(u) = L_P(u)$. First, in order to show $u_p = \alpha$, we introduce an edge-code as shown in Figure 4 (b). We code the two outgoing edges of a node in a binary tree as 0 and 1. Then, any node will have an e-code as the concatenation of the edge codes along the path from the root. For example `f` in Figure 4 (b) has an e-code `100`. It is easy to observe that the e-code of a node is the binary representation of α in the PBiTree code, and at the same time the e-code is the same as u_p in the P-PBiTree code. Second, we show that $u_l = l$ because u_l is the length of e-code, and the length of e-code is the level of the corresponding node.

The P-PBiTree keeps all the advantages of PBiTree unchanged and is superior to PBiTree because P-PBiTree does not need to use any virtual nodes and does not need to binarize XML trees in an extra preprocessing step.

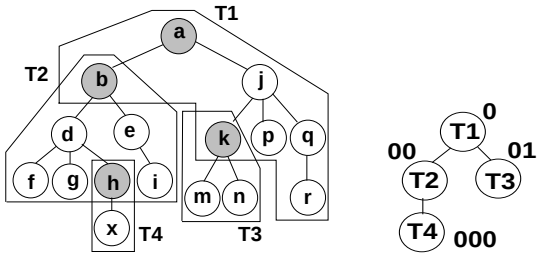


(a) P-PBiTree (b) PBiTree

Figure 4: P-PBiTree and PBiTree

4.4 Sort-Based and Hash-Based Query Processing

One of the most important issues of any numbering scheme is the performance for XML query processing. The unique feature of P-PBiTree code is that it can support both sort-based and hash-based containment join algorithms effectively and efficiently. Given a set of ancestors, A, and a set of descendants, D. First, we show that P-PBiTree numbering scheme is at least as good as the region-based schema. The region scheme processes containment join as a variation of sort merge join. It first sorts the nodes in A and D respectively according to their preorder, and checks ancestor-descendant relationship between nodes in set A and D using regions. P-PBiTree can process containment join in a similar way, because we can sort a set of nodes using their p-codes, and check ancestor-descendant relationship using prefix-containment efficiently. Second, because P-PBiTree is equivalent to PBiTree, we can use the hash-based containment join algorithms proposed with PBiTree in [16].



(a) L-Trees (b) G-Tree

Figure 5: Two Level P-PBiTree Code

4.5 Two-Level P-PBiTree

As a prefix-based code, P-PBiTree has the property of top-down propagation. If the root of a subtree is close to the root of the whole tree, then the cost of renumbering is high when the subtree is large. In paper, we propose a two-level P-PBiTree numbering scheme, denoted 2P-PBiTree. The main idea behind is to cluster nodes into a high-level node. In 2P-PBiTree, there exists a global-tree where each node in the global-tree is a local-tree. By doing so, instead of maintaining a single tree, we maintain a small global-tree and many local-trees. The motivation is to minimize the cost of renumbering in a local-tree where possible. The global-tree can be easily maintained in memory during query/update processing.

An example is shown in Figure 5. Figure 5 (a) shows a way of constructing four local-trees, T_1, T_2, T_3 and T_4. Figure 5 (b) shows the global tree where each node is one of the four local-trees. Suppose that the root node a will have a node to be inserted as the first child. With P-PBiTree, the codes for all the nodes except for the root a need to be renumbered. With 2P-PBiTree, we attempt to renumber the nodes in the local tree T_1 only.

In the following, we will describe how to code it and how to convert it to the P-PBiTree when processing a containment join. It is important to know that we can convert 2P-PBiTree codes to P-PBiTree code on the fly when the ancestor/descendant datasets are extracted from the XML tree, if needed.

4.5.1 2P-PBiTree numbering

Given a tree $T = (V, E)$ where V is a set of nodes and E is a set of edges. For a tree T, we denote the set of nodes and edges of T, as $V(T)$ and $E(T)$. A cluster, T_i, is a connected subtree of T. A cluster T_i may have a node(s), $u \in V(T_i)$, incident with a node in $V(T) - V(T_i)$. We call such u a *boundary node*. Given a set of clusters: $T_1, T_2, \cdots, T_n$, then $V(T) = \cup_{i=1}^{n} V(T_i)$, $E(T) = \cup_{i=1}^{n} E(T_i)$, and $E(T_i) \cap E(T_j) = \emptyset$ for $i \neq j$. Except for the root of the tree T, a boundary node, u, will appear in two clusters as a leaf node of the parent cluster (called leaf-representation and denoted $\downarrow u$) and as the root of the child cluster (called root-representation and denoted $\uparrow u$).

Figure 5 shows an example. There are four clusters T_1, T_2, T_3 and T_4. The shadow nodes are boundary nodes. The boundary node b appears as a leaf in T_1 and as the root in T_2.

In order to distinguish the node appearances, we use 2 bits for three different kinds of nodes: the root-representation of a boundary node (11), the leaf-representation of a boundary node (01), and a non-boundary node (00). We call it T-code of a node u. In addition to the T-code, a node, u, will have G-code and L-code. The G-code is the code of the cluster the node u belongs and the L-code is the *local* P-PBiTree code. The 2P-PBiTree code is a triple $L_{2PP}(u) = \langle u_t, u_g, u_l \rangle$ where u_t, u_g and u_l are the T-code, G-code and L-code of the node u. The codes for the tree in Figure 5 are shown in Table 1.

Our proposed 2P-PBiTree uses the similar ideas of clustering as proposed in [1]. But, there are some fundamental differences between the two approaches. In [1], the clustering is used to reduce the space complexity ($\log n + O(\sqrt{\log n})$ bit to each node of a tree sized n) rather than renumbering cost. They studied how to number a tree after converting the tree to a cluster-tree in a static environment, and restrict the number of boundary nodes in a cluster to be two, in order to form a binary tree on top of the clusters. In other words, the cluster, T_1, shown in Figure 5, is not allowed, because there are three boundary nodes. In this paper, we focus on

Node	P-PBiTree	T-Code	G-Code	L-Code
$\uparrow a$	$\langle 0, \emptyset \rangle$	11	0	$\langle 0, \emptyset \rangle$
$\downarrow b$	$\langle 1, 0 \rangle$	01	0	$\langle 1, 0 \rangle$
$\uparrow b$	$\langle 1, 0 \rangle$	11	0	$\langle 0, \emptyset \rangle$
d	$\langle 2, 00 \rangle$	00	00	$\langle 1, 0 \rangle$
e	$\langle 2, 01 \rangle$	00	00	$\langle 1, 1 \rangle$
f	$\langle 4, 0000 \rangle$	00	00	$\langle 3, 000 \rangle$
g	$\langle 4, 0001 \rangle$	00	00	$\langle 3, 001 \rangle$
$\downarrow h$	$\langle 4, 0010 \rangle$	01	00	$\langle 3, 010 \rangle$
$\uparrow h$	$\langle 4, 0010 \rangle$	11	00	$\langle 0, \emptyset \rangle$
i	$\langle 3, 010 \rangle$	00	00	$\langle 2, 10 \rangle$
j	$\langle 1, 1 \rangle$	00	0	$\langle 1, 1 \rangle$
$\downarrow k$	$\langle 3, 100 \rangle$	01	0	$\langle 3, 100 \rangle$
$\uparrow k$	$\langle 3, 100 \rangle$	11	0	$\langle 0, \emptyset \rangle$
m	$\langle 4, 1000 \rangle$	00	01	$\langle 1, 0 \rangle$
n	$\langle 4, 1001 \rangle$	00	01	$\langle 1, 1 \rangle$
p	$\langle 3, 101 \rangle$	00	0	$\langle 3, 101 \rangle$
q	$\langle 3, 110 \rangle$	00	0	$\langle 3, 110 \rangle$
r	$\langle 4, 1101 \rangle$	00	0	$\langle 4, 1101 \rangle$
x	$\langle 5, 00100 \rangle$	00	000	$\langle 1, 0 \rangle$

Table 1: P-PBiTree and 2P-PBiTree codes for Figure 5

minimization of renumbering costs in a dynamic environment. We study clustering strategies which allow more than two boundary nodes and can deal with cluster merging/splitting.

In the following, we first show how to convert 2P-PBiTree to P-PBiTree code, and then discuss two strategies: to restrict the number of clusters and to restrict the the size of a cluster.

4.5.2 Converting 2P-PBiTree to P-PBiTree

We store the 2P-PBiTree code on disk with nodes, and convert it to P-PBiTree code when extracting the nodes for further containment join processing. The conversion is straightforward. Given a node u, and suppose its local P-PBiTree code is $L"_{PP}(u) = \langle u_l, u_p \rangle$. Furthermore, suppose that u is a node in a cluster at level k in the global tree. There exists $k-1$ boundary nodes (leaf-representation) along the path from the root, $\downarrow u_1, \downarrow u_2, \cdots, \downarrow u_{k-1}$. Let the local P-PBiTree code for $\downarrow u_i$ be $L"_{PP}(\downarrow u_i) = \langle \downarrow u_{i_l}, \downarrow u_{i_p} \rangle$. Then, $L_{PP}(u) = \langle \downarrow u_{1_l} + \cdots + \downarrow u_{k_l} + u_l, \downarrow u_{1_p}. \cdots . \downarrow u_{k_p}.u_p \rangle$. Consider x in Figure 5. It is a node in the cluster T_4, and T_4 is a node at level 3 as shown in Figure 5 (b). There are two boundary nodes (leaf-representation), $\downarrow h$ in T_2 and $\downarrow b$ in T_1. The local P-PBiTree codes for $\downarrow b$, $\downarrow h$ and x are $\langle 1, 0 \rangle$, $\langle 3, 010 \rangle$ and $\langle 1, 0 \rangle$, respectively. Therefore, the global P-PBiTree code for x is $\langle 1 + 3 + 1, 00100 \rangle$. The conversion can be done efficiently via a small mapping table that can be kept in memory, when needed.

4.5.3 2P-PBiTree Numbering Scheme Updating

The clusters reduce the updates cost by reducing the top-down propagation effect. Therefore, when XML update occurs at a node, t_r, close to the root of the XML data tree, we do not need to update the whole subtree rooted at t_r, but only update the nodes in the subtree within a small cluster if any. The remaining issue is how to create such clusters for a given node, and maintain it. There are two strategies: *fixed-cluster-number* that restricts the number of clusters (denoted C_n) and *fixed-cluster-size* that restricts the the number of nodes in a cluster (denoted C_s). Note C_n and C_s are treated as soft constraints to guide node clustering.

We construct the 2P-PBiTree as follows. We parse an XML document using postorder traversal as supported in SAX (`http://sax.sourceforge.net`). That is, we can obtain the

Description	2MB	5MB	57M
Num Nodes	33,140	83,353	823,911
Num Nonleaf Nodes	9,183	22,680	227,365
Num Leaf Nodes	23,957	60,853	605,546
Max Degree	510	1,275	12,750
Min Degree	1	1	1
Avg Degree	3.6	3.7	3.7
Std Deviation	7.4	11.1	33.9
Max Depth	12	12	12
Min Depth	3	3	3
Avg Depth	5.6	5.5	5.6
Std Deviation	1.7	1.7	1.7

Table 2: The Characteristics of Target XML Data Trees

size of a subtree while parsing the tree without extra costs. For fixed-cluster-number (C_n), we estimate the number of nodes that exist in a cluster on average. The estimation does not necessarily be accurate. If the number of clusters being created is larger/smaller than C_n, we can further splitting/merging clusters. When the size of a subtree is greater than C_n, a cluster will be created. For fixed-cluster-size (C_s), unlike clustering fixed-cluster-number, some special attention needs to be paid, which needs to handle clustering with *virtual nodes*. We explain it below. Recall the average depth of most XML trees is low and the trees are balanced with relatively high degrees. As a result, a root of a subtree, v, may have a large number of direct child nodes, $u_1, u_2, \cdots, u_k$ where each subtree rooted at u_i is rather small in size ($\ll C_s$), but the size of the subtree rooted at v is large ($> C_s$). If we cluster each of subtree rooted at u_k, then we will create too many clusters. In order to avoid creating a large number of clusters with only one/two nodes inside, we introduce virtual nodes. A virtual node is created as the direct child of v and as the parent of a group of consecutive child nodes $u_i, u_{i+1}, \cdots, u_j$, in order to cluster the subtrees rooted at $u_i, u_{i+1}, \cdots, u_j$ into a single cluster. A virtual node is a special boundary node, which reduces the number of clusters to be created.

After the initial trees (global-tree and local-trees) being constructed, when XML updates occur, we insert/delete nodes from clusters. Because we maintain some statistics in each node in the global-tree, we can easily tell whether a cluster becomes too small/bit. and then we conduct merging/splitting.

5 Performance Studies

We have implemented the following algorithms: prefixed-based number schema (denoted `Pr`) for all one-bit growth, double-bit growth and Dewey code because they perform the same in terms of renumbering cost, durable region-based numbering scheme (denoted `DR`), as well as the newly proposed P-PBiTree and 2P-PBiTree numbering schema. For P-PBiTree and 2P-PBiTree, we implemented `PBn` (P-PBiTree without code preserving), `PBp` (P-PBiTree with code preserving), `PBnS` (2P-PBiTree with fixed-cluster-size strategy without code preserving), `PBpS` (2P-PBiTree with fixed-cluster-size strategy and code preserving), `PBnN` (2P-PBiTree with fixed-cluster-number strategy without code preserving), and `PBpN` (2P-PBiTree with fixed-cluster-number strategy and code preserving).

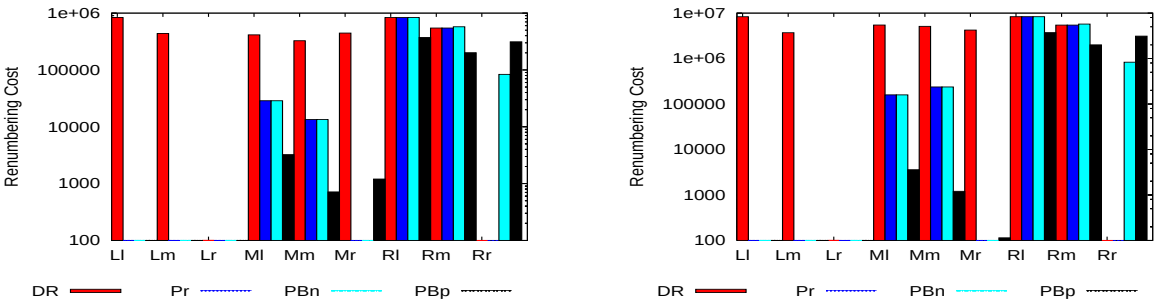
We use XMark dataset in our experimental studies [13]. The DTD of XMark benchmark carefully modeled an auction considering all hierarchical element structure, references and text generation. We selected three scale factors to generate XMark datasets (0.02, 0.05, 0.5), and created three datasets, 2MB, 5MB and 57MB. We use them to generate initial XML data trees

Scheme	2MB	5MB	57M
Durable	64	64	64
One-Bit	544	1,306	12,785
Double-Bit	64	76	81
Dewey	66	68	76
P-PBiTree	**39**	**41**	**45**
2P-PBiTree	47	50	53

Table 3: The Max Code Length (bit)

Scheme	2MB	5MB	57M
Durable	2,120,960	5,334,592	52,730,304
One-Bit	5,163,676	28,583,412	2,609,894,587
Double-Bit	1,229,501	3,660,480	42,624,458
Dewey	1,149,915	3,015,803	33,280,650
P-PBiTree	**856,454**	**2,291,274**	**25,378,188**
2P-PBiTree	1,145,540	2,996,590	31,757,978

Table 4: The Total of Code Length (bit)



(a) 9 Insertion Position (5M)

(b) 9 Insertion Position (57M)

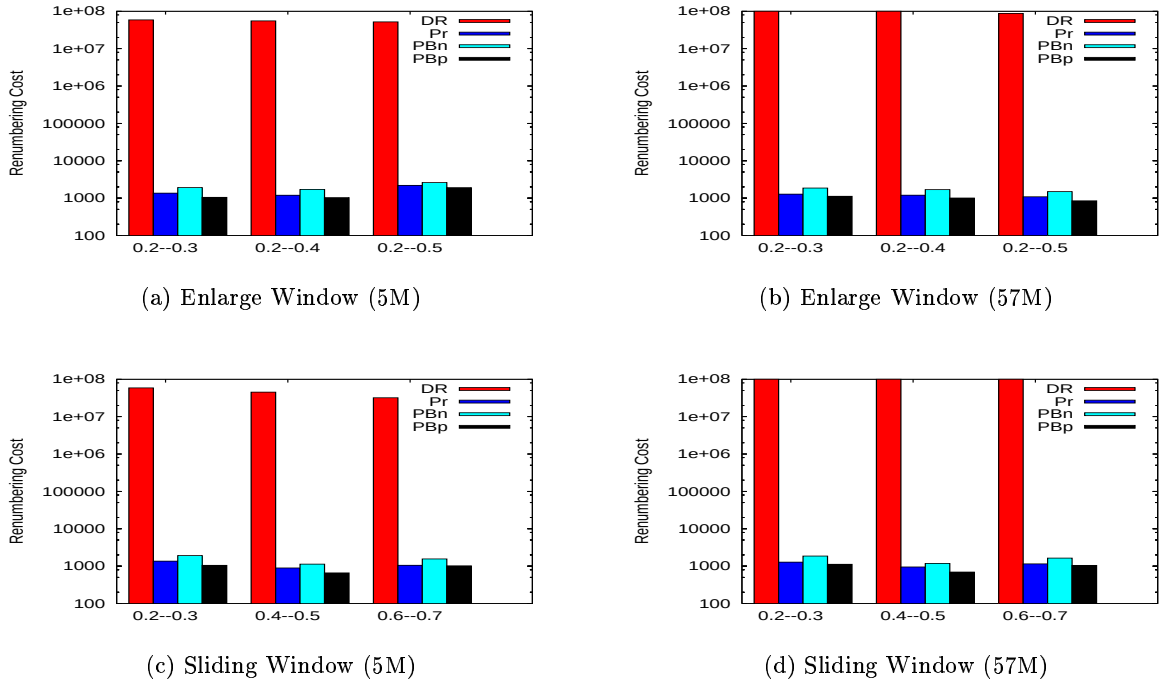
Figure 6: Insertion Positions

to be updated, known as *target XML data trees*. The characteristics of the three target XML data trees are summarized in Table 2. All the three trees have about 27% of nonleaf nodes and 73% leaf nodes. They are balanced with relatively high degrees. The reason of choosing XMark datasets is because that they share the characteristics of most XML documents (the average depth is low, and the depth is relatively high), as observed in [4].

We focus on the XML update costs in particular with the numbering scheme update. We report the renumbering cost for insertion in this paper. The metric of cost is the sum of renumbered nodes when insertions occur. We also measure the code-length of these approaches as one metric. The maximum code lengths are shown in Table 3, and the total sum of the code lengths are shown in Table 4. P-PBiTree code needs less memory space than all the others in terms of the max bit-length and the total sum of the bit-lengths. 2P-PBiTree is the second best.

5.1 Exp-1: Insertion Pattern Testing without Clustering

Figure 6 (a) and (b) show the total renumbering cost for 10 insertions, at 9 insertion positions using 5M/57M XMark datasets. Here, the 9 insertion positions are represented by two letters where `L/M/R` represent the leaf level, the middle level and the top level, respectively, and `l/m/r` represent the leftmost/middle/rightmost child at a certain level, respectively. Several observations can be made. First, `Pr`, `PBn` and `PBp` do not cost any at `Ll/Lm/Lr`. `RD` costs most



(a) Enlarge Window (5M)

(b) Enlarge Window (57M)

(c) Sliding Window (5M)

(d) Sliding Window (57M)

Figure 7: Insertion Windows

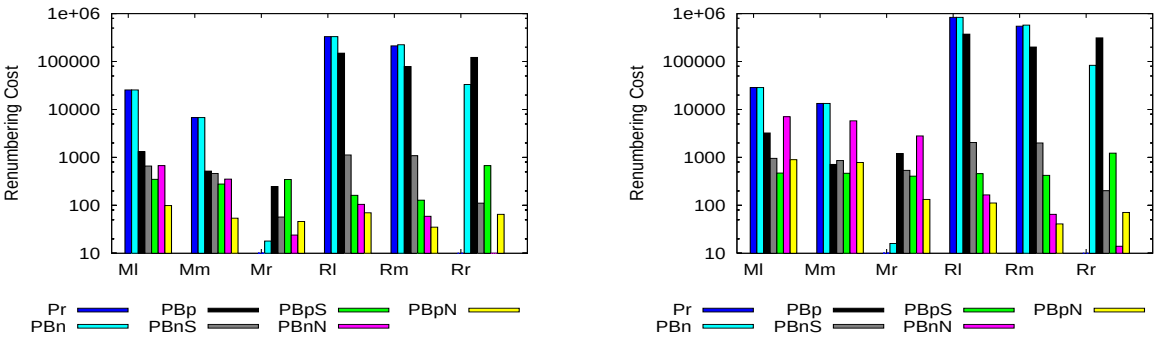
at `Ll` as expected, and cost more than expected at the top level, because it needs to reorder. Overall, `PBn` performs in a similar way as `Pr`. `PBp` significantly outperforms `Pr` in all the cases except `Rr`, because `Pr` does not cost any when appending a new node.

In Figure 7 (a) and (b), we tested several insertion windows, and report the total renumbering cost for 1,000 insertions where 80% of the insertions are in an insertion window W and 20% are of random insertions. The window is specified as below. First, we treat all nodes in a sequence following a traversal order. Second, the insertion window, for example 0.2–0.3, indicates the insertion is from the 0.2-quantile to 0.3-quantile in the whole sequence. Three windows are tested: 0.2–0.3, 0.2–0.4 and 0.2–0.5 (enlarging window). Also, Figure 7 (c) and (d) shows total renumbering costs for 1,000 insertions with a sliding window (0.2–0.3, 0.4–0.5, 0.6–0.7). `Pr` outperforms `PBn` marginally, and `PBp` outperforms `Pr` marginally. It is because about 70% of nodes in the windows are leaf nodes.

5.2 Exp-2: Insertion Pattern Testing with Clustering

We repeat the same testing for clusterings with two strategies and with/without code preserving. In following, we do not report `DR` because it does not perform well in all the testings. We concentrate on prefix-based numbering schema. For `PBnN` and `PBpN` we use $C_n = 5,000$ and for `PBnS` and `PBpS`, we use $C_s = 100$, as default. We will report the effectiveness of these parameters later.

Figure 8 shows the total renumbering cost for 10 insertions at each of the 6 insertion positions, because all the prefix-based numbering schema do not cost any when XML updates occur at leaf level. All `PBnN`, `PBpN`, `PBnS` and `PBpS` outperform `Pr` significantly. For example, for the insertion pattern of `Rm`, `Pr` = 212,707, `PBnS` = 1,085, `PBpS` = 128, `PBnN` = 59 and `PBpN` = 35. `PBnS` is worst among the other PBiTree codes, but costs only 0.5% of the renumbering cost of `Pr`. In most cases, clusterings with preserving outperform clusterings without preserving. But, in the case of



(a) 2M

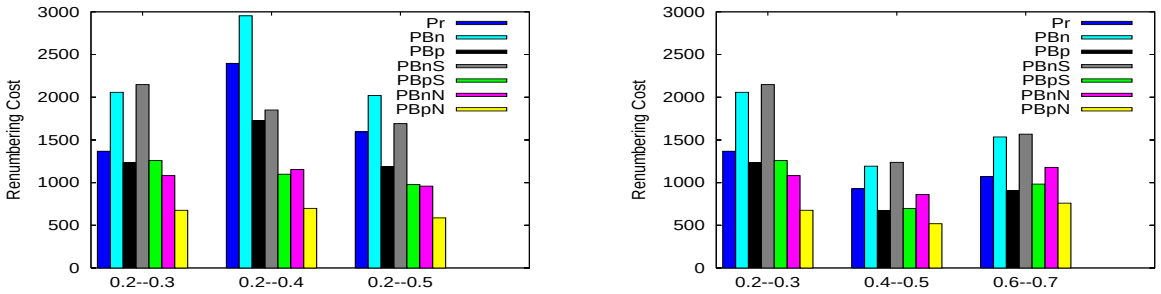
(b) 5M

Figure 8: Basic Insertion Patterns (Clustering)

`Rr` (inserting a node as the rightmost child of the root), clustering without preserving performs better, because it does not need to rebalance the codes among the siblings. With the same setting like Figure 8, we tested several insertion windows as we did for non-clustering causes. As shown in Figure 9, in all cases, clustering with preserving outperforms `Pr` significantly. However, there exist cases that clustering without preserving perform worse than `Pr`. For example, when 80% insertions occur in the window 0.2–0.3, `PBnS` performs worse than `Pr`. It is because most nodes in the window are leaf nodes. The benefits of having clusters are invisible. In order to test the effectiveness of clustering, we tested insertions at specific levels, specifically. Figure 10 shows the results, where 1,000 insertions occur at a certain level ranging from 0 to 4 (the high portion of the tree). In Figure 10 (a) and (c), we can see that `DR` performs the worst because the traversal order needs to be updated when an insertion occur. Both `PBn` and `PBp` outperform `Pr` and `DR`. Unlike `DR` which performs the same when the level increases, the renumbering cost for all the prefix-based numbering schema decreases while the level increases, as subtrees that need to be updated become smaller. Figure 10 (b) and (d) show the performance of `PBnS`, `PBpS`, `PBnN` and `PBpN` which all outperform `Pr` significantly, in particular when at level 0-1.

5.3 Exp-3: Clustering Tuning

In this experimental study, we show the effectiveness of the two parameters: C_n and C_s which control the number of clusters and the cluster size, respectively. In Figure 11, we run 1,000 insertions using the insertion pattern `Rm` (inserting at the middle child of the root). We vary C_n in the range of 500, 1,000, 2,000 and 5,000, and vary C_s in the range of 100, 500, 1,000 and 2,000. As shown in Figure 11 (a) and (b), with all C_s values, `PBpS` outperforms `PBnS`. Even when C_s is large (2,000), `PBnS` shows high performance, because it only renumbers 4 nodes on average per insertion. When C_s becomes smaller, the less number of nodes can reside in a cluster, the renumbering cost will become even smaller. From Figure 11 (a) and (b), we can see that a cluster sized of 1,000 nodes can achieve satisfactory level of performance. The similar observations can be made for C_n (Figure 11 (c) and (d)). When $C_n = 2,000$ (the total number of clusters is 2,000), the renumbering cost reduces to a minimal level, therefore, further increasing the number of clusters does not necessarily reduce the renumbering cost. Figure 12 shows the testing results using 1,000 insertion queries at 6 different insertion positions. Here, `PBnS`-n and `PBpS`-n means $C_s = n$, and `PBnN`-n and `PBpN`-n means $C_n = n$. Our clustering strategies perform significantly well at the top level. When the level becomes larger, most of insertions will fail into



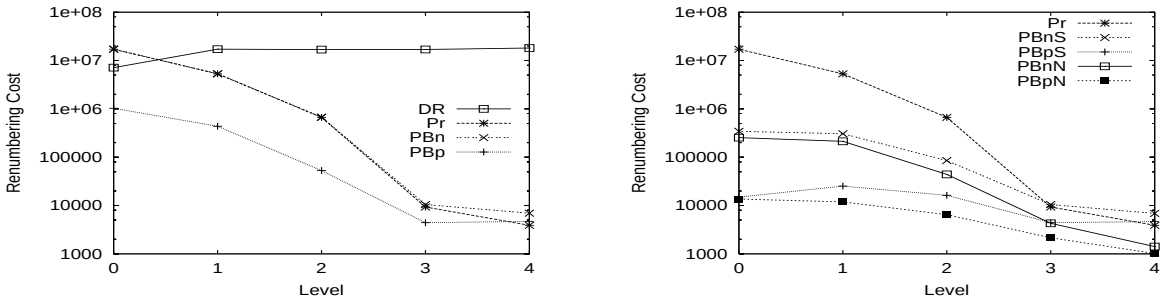
(a) Enlarge Window (2M)

(b) Sliding Window (2M)

(c) Enlarge Window (5M)

(d) Sliding Window (5M)

Figure 9: Three Insertion Windows with Clusterings



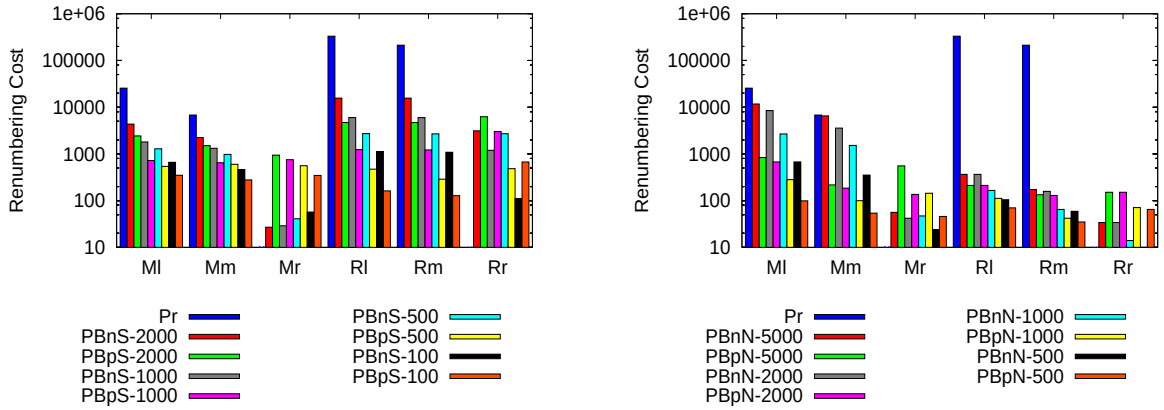
(a) Non-Clustering (2M)

(b) Clustering (2M)

(c) Non-Clustering (5M)

(d) Clustering (5M)

Figure 10: Varying Levels



(a) Fixed Size (2M)

(b) Fixed Number (2M)

(c) Fixed Size (5M)

(d) Fixed Number (5M)

Figure 11: Vary Clustering Parameters

C_s	CC	VN	C_{max}	C_{min}	C_{avg}	C_{std}
100	836	998	372	7	84.70	0.61
500	176	170	523	209	475.61	3.10
1,000	80	87	1,000	232	961.14	9.85
2,000	43	42	2,000	1,508	1943.60	16.98

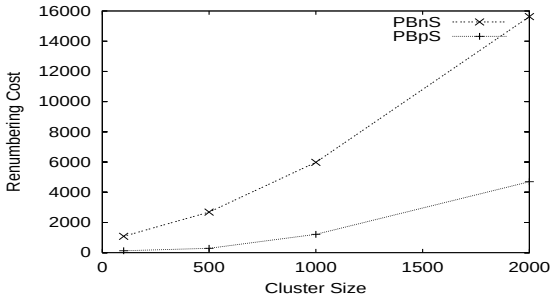
Table 5: Characteristics (Fixed-Cluster-Size) (5M)

a cluster, therefore the effectiveness of clustering is less visible. Note that if insertions occur at the rightmost position, `Pr` does not cost any, because it is designed in such a way.

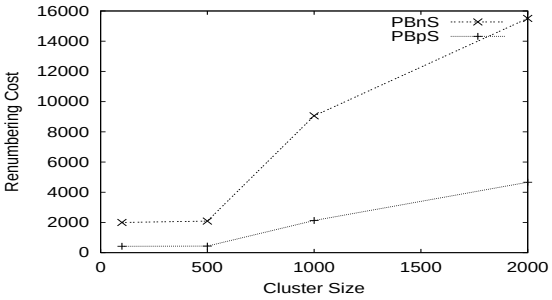
Some statistics about clustering are summarized in Table 5 and Table 6. In both Tables, the symbols have the following meanings: CC (the number of clusters being generated), C_{max}/C_{min} (the maximum/minimum number of nodes in a cluster), C_{avg} and C_{std} are the average number of nodes in a cluster and its standard deviation. In terms of quality of clustering, clustering using fixed-cluster-size is superior to clustering using fixed-cluster-number. It is because the standard deviation of the average number of nodes are different. The standard deviation in fixed-cluster-size is rather small, whereas the standard deviation in fixed-cluster-number is considerably large, which mean some clusters are very big in size. The quality of clustering with fixed-cluster-size is ensured using the virtual nodes. Table 5 shows the number of virtual nodes being generated.

C_n	CC	C_{max}	C_{min}	C_{avg}	C_{std}
2,000	468	16,733	13	179.49	51.11
1,000	75	22,083	63	1114.76	423.00
500	14	27,883	63	5967.57	2134.57
100	10	28,447	584	8354.20	2693.37

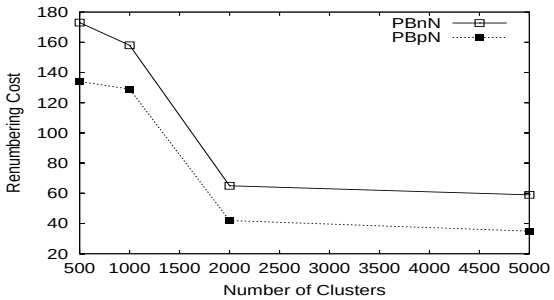
Table 6: Characteristics (Fixed-Cluster-Number) (5M)



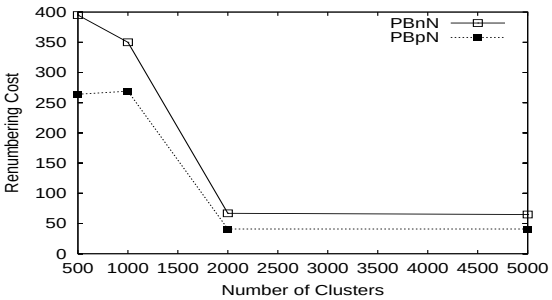
(a) Cluster Size (2M)



(b) Cluster Size (5M)



(c) Cluster Number (2M)



(d) Cluster Number (5M)

Figure 12: Effectiveness of Cluster Numbers and Sizes

6 Conclusion

In this paper, we proposed new numbering schema which minimize the code-length, minimize the renumbering costs on average, and support both sort-based and hash-based containment joints. The significant reduction on renumbering costs is achieved by a combination of code-preserving and clustering. Our experimental studies verified that our proposed numbering schema outperform the existing numbering schema significantly in terms of storage space consumption and updating performance. As our future work, we will continue our investigations on maintaining suboptimal clusters that can further minimize the renumbering costs.

References

[1] Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In *Proceedings of SODA'02*, 2002.

[2] D. Chamberlin, D. Florescu, and J. Robie et al. XQuery: A query language for XML. In *W3C Working Draft*, `http://www.w3.org/TR/xquery`, 2001.

[3] J. Clark and S. DeRose. XML path language (XPath). In *W3C Recommendation 16 November 1999*, `http://www.w3.org/TR/xpath`.

[4] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proceedings of PODS'02*, 2002.

[5] H.V. Jagadish, Laks V.S. Lakshmanan, Divesh Srivastava, and Keith Thompson. Taxz: A tree algebra for xml. In *Proceedings of 8th International Workshop on DBLP*, 2001.

[6] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of SODA'02*, 2002.

[7] D. D. Kha, M. Yoshikawa, and S. Uemura. An XML indexing structure with relative region coordinate. In *Proceedings of ICDE'01*, 2001.

[8] W. Eliot Kimber. HyTime and SGML: Understanding the HyTime HYQ query language 1.1. Technical report, IBM Corporation, 1993.

[9] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1973.

[10] Y. K. Lee, S. J. Yoo, and K. Yoon. Index structures for structured documents. In *Proceedings of ACM First International Conference on Digital Libraries*, 1996.

[11] S. Lei and G. ãzsoyoglu Z. M. ãzsoyoglu. A graph query language and its query processing. In *Proceedings of ICDE'99*, 1999.

[12] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of VLDB'01*, 2001.

[13] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. The XMark: A benchmark for XML data management. In *Proceedings of VLDB'02*, 2002.

[14] Igor Tatarnov, Zachary G. Ives, Along Y. Halevy, and Daniel S. Wed. Updating XML. In *Proceedings of SIGMOD'01*, 2001.

[15] Igor Tatarnov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and quering ordered XML using a relational database system. In *Proceedings of SIGMOD'02*, 2002.

[16] W. Wang, H. Jiang, H. Lu, and J.X. Yu. Pbitree coding and efficient processing of containment join. In *Proceedings of ICDE'03*, 2003.

[17] M. YoshiKawa and T. Amagasa. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1), 2001.

[18] C. Zhang, Jeffrey F. Naughton, David J. DeWitt, Q. Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD'01*, 2001.