An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML

Changqing Li and Tok Wang Ling

Department of Computer Science, National University of Singapore
{lichangq, lingtw}@comp.nus.edu.sg

Abstract. A number of labeling schemes have been designed to facilitate the query of XML, based on which the ancestor-descendant relationship between any two nodes can be determined quickly. Another important feature of XML is that the elements in XML are intrinsically ordered. However the label update cost is high based on the present labeling schemes. They have to re-label the existing nodes or re-calculate some values when inserting an order-sensitive element. Thus it is important to design a scheme that supports order-sensitive queries, yet it has low label update cost. In this paper, we design a binary string prefix scheme which supports order-sensitive update without any re-labeling or re-calculation. Theoretical analysis and experimental results also show that this scheme is compact compared to the existing dynamic labeling schemes, and it provides efficient support to both ordered and un-ordered queries.

1 Introduction

The growing number of XML [7] documents on the Web has motivated the development of systems which can store and query XML data efficiently. XPath [5] and XQuery [6] are two main XML query languages.

There are two main techniques to facilitate the XML queries, viz. structural index and labeling (numbering) scheme. The structural index approaches, such as dataguide [9] in the Lore system [11] and representative objects [13], can help to traverse through the hierarchy of XML, but this traverse is costly. The labeling scheme approaches, such as containment scheme [2, 10 16, 17], prefix scheme [1, 8, 11, 14] and prime scheme [15], require smaller storage space, yet they can efficiently determine the ancestor-descendant and parent-child relationships between any two elements of the XML. In this paper, we focus on the labeling schemes.

One salient feature of XML is that the elements in XML are intrinsically ordered. This implicit ordering is referred to as document order (the element sequence in the XML). The labeling scheme should also have the ability to determine the order-sensitive relationship.

The main contributions of this paper are summarized as follows:

- This scheme need not re-label any existing nodes and need not re-calculate any values when inserting an order-sensitive node into the XML tree.

- The theoretical analysis and experimental results both show that this scheme has smaller storage requirement.

The rest of the paper is organized as follows. Section 2 reviews the related work and gives the motivation of this paper. We propose our improved binary string prefix scheme in Section 3. The most important part of this paper is Section 4, in which we show that the scheme proposed in this paper need not re-label any existing nodes and need not re-calculate any values when updating an ordered node. The experimental results are illustrated in Section 5, and we conclude in Section 6.

2 Related Work and Motivation

In this section, we present three families of labeling schemes, namely containment [2, 10 16, 17], prefix [1, 8, 11, 14] and prime [15].

Containment Scheme. Agrawal et al [2] use a numbering scheme in which every node is assigned two variables: “start” and “end”. These two variables represent an interval [start, end]. For any two nodes u and v, u is an ancestor of v iff label(u.start) < label(v.start) < label(v.end) < label(u.end). In other words, the interval of v is contained in the interval of u.

Although the containment scheme can determine the ancestor-descendant relationship quickly, it does not work well when inserting a node into the XML tree. The insertion of a node may lead to a re-labeling of all the nodes of the tree. This problem may be alleviated if we increase the interval size with some values unused. However, it is not so easy to decide how large the interval size should be. Small interval size is easy to lead to re-labeling, while large interval size wastes a lot of values which causes the increase of storage.

[3] uses real (float-point) values for the “start” and “end” of the intervals. It seems that this approach solve the re-labeling problem. But in practice, the float-point is represented in computer with a fixed number of bits which is similar to the representation of integer. As a result, there are a finite number of values between any two real values [14].

Prefix Scheme. In the prefix labeling scheme, the label of a node is its parent’s label (prefix) concatenates the delimiter and its own label. For any two nodes u and v, u is an ancestor of v iff label(u) is a prefix of label(v). There are two main prefix schemes, the integer based and the binary string based.

DeweyID [14] is an integer based prefix scheme. It labels the nth child of a node with an integer n, and this n should be concatenated to the prefix (its parent’s label) to form the complete label of this child node.

On the other hand, Cohen et al use Binary strings to label the nodes (*Binary*) [8]. Each character of a binary string is stored using 1 bit. The root of the tree is labeled with an empty string. The first child of the root is labeled with “0”, the second child with “10”, the third with “110”, and the fourth with “1110”, etc. Similarly for any node u, the first child of u is labeled with label(u).“0”, the second child of u is labeled with label(u).“10”, and the ith child with label(u).“$1^{i-1}0$”.

Compared to the containment scheme, the prefix scheme only needs to re-label the sibling nodes after this inserted node and the descendants of these siblings, which is more dynamic in updating.

Prime Number Scheme. Wu et al [15] proposed a novel approach to label XML trees with prime numbers (*Prime*). The label of a node is the product of its parent_label and its own self_label (the next available prime number). For any two nodes u and v, u is an ancestor of v iff label(v) *mod* label(u) = 0.

Furthermore, Prime utilizes the Chinese Remainder Theorem (CRT) [4, 15] for the document order. When using the Simultaneous Congruence (SC) value in CRT to mod the self_label of each node, the document order for each node can be calculated. When new nodes are inserted into the XML tree, Prime only needs to re-calculate the SC value for the new ordering of the nodes instead of re-labeling.

In addition, Prime uses multiple SC values rather than a single SC value to prevent the SC value to become a very very larger number.

The prefix and prime schemes are called *dynamic* labeling schemes, and we only compare the performance of *dynamic* labeling schemes in this paper.

Motivation. The Binary and DeweyID prefix schemes both need to re-label the existing nodes when inserting an order-sensitive node.

Although Prime is a scheme which supports order-sensitive updates without any re-labeling of the existing nodes, it needs to re-calculate the SC values based on the new ordering of nodes. The SC values are very large numbers and the re-calculation is very time consuming.

In addition, the Prime scheme skips a lot of integers to get the prime number, and the label of a child is the product of the next available prime number and its parent's label, which both make the storage space for Prime labels large.

Thus the objective of this paper is to design a scheme 1) which need not re-label any existing nodes and need not re-calculate any values when inserting an order-sensitive node (Section 4.1 and 5.3), and 2) which requires less storage space for the labels (Section 3.2 and 5.1).

3 Improved Binary String Prefix Scheme

In this section, we elaborate our Improved Binary string prefix scheme (ImprovedBinary). Firstly we use an example to illustrate how to label the nodes based on our ImprovedBinary. Then we describe the formal labeling algorithm of this scheme. Also we analyze the size requirements of different labeling schemes.

In prefix schemes, the string before the last delimiter is called a prefix_label, the string after the last delimiter is called a self_label, and the string before the first delimiter, between two neighbor delimiters or after the last delimiter is called a component.

Example 3.1. Figure 1 shows our ImprovedBinary scheme. The root node is labeled with an empty string. Then we label the five child nodes of the root. The prefix_labels of these five child nodes are all empty strings, thus the self_labels are exactly the complete labels for these five child nodes. The self_label of the first (left) child node is "01", and the self_label of the last (right) child node is "011". We use

“01” and “011” as the first and last sibling self_labels because in this way, we can insert nodes before the first sibling and after the last sibling without any re-labeling of existing nodes. See Section 4.1.

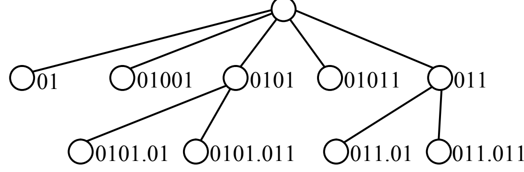

Fig. 1. ImprovedBinary scheme.

When we know the left and right self_labels, we can label the middle self_label and 2 cases will be encountered: *Case (a) left self_label size ≤ right self_label size, and Case (b): left self_label size > right self_label size.* For Case (a), the middle self_label is that we change the last character of the right self_label to “0” and concatenate one more “1”. For Case (b), the middle self_label is that we directly concatenate one more “1” after the left self_label.

Now we label the middle child node, which is the third child, i.e. $\lfloor (1+5)/2 \rfloor = 3$. The size of the 1st (left) self_label (“01”) is 2 and the size of the 5th (right) self_label (“011”) is 3 which satisfies Case (a), thus the self_label of the third child node is “0101” (“011” → “010” → “0101”).

Next we label the two middle child nodes between “01” and “0101”, and between “0101” and “011”. For the middle node between “01” (left self_label) and “0101” (right self_label), i.e. the second child node ($\lfloor (1+3)/2 \rfloor = 2$), the left self_label size 2 is smaller than the right self_label size 4 which satisfies Case (a), thus the self_label of the second child is “01001” (“0101” → “0100” → “01001”). For the middle node between “0101” (left self_label) and “011” (right self_label), i.e. the fourth child ($\lfloor (3+5)/2 \rfloor = 4$), the left self_label size 4 is larger than the right self_label size 3 which satisfies Case (b), thus the self_label of the fourth child is “01011” (“0101” ⊕ “1” → “01011”).

Theorem 3.1. The sibling self_labels of ImprovedBinary are lexically ordered.

Theorem 3.2. The labels (prefix_label ⊕ delimiter ⊕ self_label) of ImprovedBinary are lexically ordered when comparing the labels component by component.

Example 3.2. The self_labels of the five child nodes of the root in Figure 1 are lexically ordered, i.e. “01” $\prec$ “01001” $\prec$ “0101” $\prec$ “01011” $\prec$ “011” lexically. Furthermore, “0101.011” $\prec$ “011.01” lexically.

3.1 The Formal Labeling Algorithm

We firstly discuss the AssignMiddleSelfLabel algorithm (Figure 2) which inserts the middle self_label when we know the left self_label and the right self_label. If the size of the left self_label is smaller than or equal to the size of the right self_label, the self_label of the middle node is that we change the last character of the *right* self_label to “0” and concatenate one more “1”. Otherwise, the self_label of the middle node is the *left* self_label concatenates “1”.

```
Algorithm 1: AssignMiddleSelfLabel
Input: left self label self_label_L, and right self label self_label_R
Output: middle self label self_label_M, such that
                    self_label_L ≺ self_label_M ≺ self_label_R lexically.
begin
1: calculate the size of self_label_L and the size of self_label_R
2: if size(self_label_L) ≤ size(self_label_R)
     then self_label_M = change the last character of self_label_R to "0",
                    and concatenate (⊕) one more "1"
3: else if size(self_label_L) > size(self_label_R)
     then self_label_M = self_label_L ⊕ "1"
end
```

Fig. 2. AssignMiddleSelfLabel algorithm.

Next we discuss how to label the whole XML tree. Figure 3 shows the Labeling algorithm. We firstly get all the sibling child nodes of a node. If there is only one sibling, the self_label of this node is "01". Otherwise, the self_label of the first sibling node is "01" and the self_label of the last sibling node is "011". We use the SubLabeling function to get all the self_labels of the rest sibling nodes.

```
Algorithm 2: Labeling
Input: XML document
Output: Label of each node
begin
1: for all the sibling child nodes of each node of the XML document
2: for the first sibling child node, self_label[1]="01" //self_label is an array
3: if the Number of Sibling nodes SN > 1
     then self_label[SN]="011"
          self_label=SubLabeling(self_label, 1, SN)
4: label = prefix_label ⊕ delimiter ⊕ each element of self_label array
end

SubLabeling
Input: self_label array, left element index of self_label array L, and right element index of self_label array R
Output: self_label array
begin
1: M = floor((L+R)/2)  //M refers to the M^th element of self_label array
2: if L+1<R
     then self_label[M]=AssignMiddleSelfLabel(self_label[L], self_label[R])
          SubLabeling(self_label, L, M)
          SubLabeling(self_label, M, R)
end
```

Fig. 3. Labeling algorithm.

SubLabeling is a recursive function, the input of which is a self_label *array*, the left element index of self_label "L" and the right element index of self_label "R". This function assigns the middle self_label (self_label[M]) using the AssignMiddleSelfLabel algorithm (Figure 2), then it uses the new left and right self_label positions to call the SubLabeling function itself, until each element of the self_label array has a value.

Finally the label of each sibling node is the prefix_label concatenates the self_label.

3.2 Size Analysis[1]

In this section, we analyze the size required by the DeweyID, Binary, Prime and our ImprovedBinary. The "D", "F" and "N" are respectively used to denote the maximal depth, maximal fan-out and number of nodes of an XML tree.

DeweyID. The maximal size to store a single self_label is $\log(F)$ (all the self_labels of DeweyID use this size). When considering the prefix, the maximal size to store a complete label (prefix_label ⊕ self_label) is $D \times \log(F)$ since the maximal depth is D and there are at most (D-1) delimiters in the prefix_label. Thus the maximal size required by DeweyID to store all the nodes in the XML tree is

$$N \times D \times \log(F) \quad \textbf{(1)}$$

Binary. The size of the first sibling self_label is only 1, the second is 2, ···, and the Fth is F. Thus the actual total sibling self_label size is $1+2+\cdots+F = (1+F)\times F/2 = F^2/2+F/2$, and the average size for a single self_label is $F/2+1/2$. Thus the maximal size to store all the nodes in the XML tree is

$$N \times D \times (F/2+1/2) \quad \textbf{(2)}$$

From formulas (1) and (2), we can see that the size of Binary is larger than the size of DeweyID.

Prime. According to the size analysis of Prime in [15], the maximal size required to store all the nodes in the XML tree is

$$N \times D \times \log(N \times \log(N)) \quad \textbf{(3)}$$

Comparing formulas (1) and (3), F is definitely less than $N \times \log(N)$, thus DeweyID requires smaller label size than Prime when considering the worst case. This is intuitive when we notice that Prime skips many integers to get the prime number and uses the product of two numbers.

ImprovedBinary. Finally we consider the size required by our ImprovedBinary.

Example 3.3. For the 5 sibling self_labels of the child nodes of the root in Figure 1, the first and last sibling self_labels are "01" and "011" with size 2 and 3 bits respectively. The middle self_label between "01" and "011" is "0101" with size 4 bits. Then for the two middle nodes "01001" and "01011" (between "01" and "0101", and between "0101" and "011"), their sef_label sizes are both 5, and so on.

Table 1 shows the relationship between the size of a label and the number of labels with this size. There is one label with size 2, one label with size 3, 2^0 label with size 4, 2^1 labels with size 5, 2^2 labels with size 6, 2^3 labels with size 7, ···, and 2^n labels with size n+4. The number of sibling nodes F is equal to $1+1+2^0+2^1+2^2+2^3+\cdots+2^n=2^{n+1}+1$. Therefore $2^n = (F-1)/2$, and $n+4 = \log(F-1)+3$. Thus the total sibling self_label size is

$$\begin{aligned} & 2+3+1\times 4+2^1\times 5+2^2\times 6+2^3\times 7+\cdots 2^n\times(n+4) \\ &= 2+3+1\times 4+2^1\times 5+2^2\times 6+2^3\times 7+\cdots (F-1)/2\times(\log(F-1)+3) \end{aligned}$$

[1] The size in this paper refers to bits and the log in this paper is used as the logarithm to base 2.

$= F\log(F-1) + 2F - \log(F-1) + 1$

Hence the average size for a single self_label is

$= \log(F-1) + 2 - \log(F-1)/F + 1/F$

Accordingly the maximal size required to store all the nodes in the XML tree is

$$N \times D \times (\log(F-1) + 2 - \log(F-1)/F + 1/F) \quad \textbf{(4)}$$

Table 1. Number of sibling nodes and single sibling self_label size of ImprovedBinary.

Number of labels with this size	Size (bits)
1	2
1	3
1 (2^0)	4
2 (2^1)	5
4 (2^2)	6
8 (2^3)	7
...	…
2^n	n+4

It can be seen from formulas (1) and (4) that the size required by ImprovedBinary is as small as the size required by DeweyID. In addition, DeweyID uses fixed length for all the self_labels. On the other hand, our ImprovedBinary uses variable length, therefore the self_label size of our ImprovedBinary will not always employs the maximal fan-out *F*. As a result, the actual total label size of our ImprovedBinary should be smaller than the actual total label size of DeweyID. Consequently the size required by our ImprovedBinary is smaller than the size required by Binary and Prime. This will be confirmed in Section 5.1 by the experimental results.

4 Order-Sensitive Update and Query

The most important part of this paper is Section 4.1, in which we show that our ImprovedBinary scheme need not re-label any existing nodes and need not re-calculate any values when inserting an order-sensitive node. In Section 4.2, we briefly introduce how to answer order-sensitive queries based on different schemes.

4.1 Order-Sensitive Update

The deletion of a node will not affect the ordering of the nodes in the XML tree. Therefore in this section, we discuss the following three order-sensitive insertion cases.

Case (1): Insert a node before the first sibling node. The self_label of the inserted node is that the last character of the first self_label is changed to “0” and is concatenated with one more “1”. After insertion, the order is still kept.

Case (2): Insert a node at any position between the first and last sibling node. We use the AssignMiddleSelfLabel algorithm introduced in Section 3.1 to assign the self_label of the new inserted node. After insertion, the order is still kept.

Case (3): Insert a node after the last sibling node. The self_label of the inserted node is that the last self_label concatenates one more "1". After insertion, the order is still kept.

In the above three cases, the prefix_labels of the inserted nodes are the same as the prefix_labels of the sibling nodes.

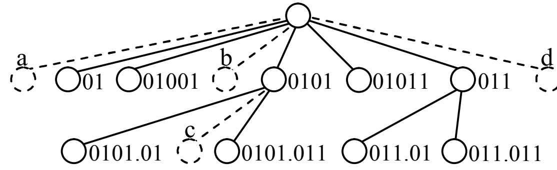

Fig. 4. Order-sensitive update for ImprovedBinary.

Example 4.1. When inserting the node "*a*" (see Figure 4), it is Case (1), thus the self_label (label) of "a" is "*001*" ("01" $\rightarrow$ "00"$\rightarrow$ "001"). When inserting the node "*b*", it is case (2) and we use the AssignMiddleSelfLabel algorithm to assign the self_label of "b". The left self_label of "b" is "01001" with size 5 and the right self_label of "b" is "0101" with size 4, therefore we directly concatenate one more "1" after the left self_label ("01001" $\oplus$ "1" $\rightarrow$ "010011"), then the self_label of "b" is "*010011*". When inserting the node "*c*", it is still case (2), but the left self_label ("01") size < the right self_label ("011") size, therefore the self_label of "c" is "0101" ("011" $\rightarrow$ "010" $\rightarrow$ "0101"), and the complete label of "c" is "*0101.0101*". When inserting the node "*d*", it is case (3), thus the self_label of "d" is "*0111*" ("011" $\oplus$ "1" $\rightarrow$ "0111"). After insertion, the orders are still kept, i.e. label(a) $\prec$ "01", "01001" $\prec$ label(b) $\prec$ "0101", "0101.01" $\prec$ label(c) $\prec$ "0101.011", and "011" $\prec$ label(d) lexically.

It can be seen that for all the above three cases, ImprovedBinary need not re-label any existing nodes and need not re-calculate any values.

On the other hand, DeweyID and Binary need to re-label all the sibling nodes following the inserted node and all the descendant nodes of the following sibling nodes for Case (1) and (2). Prime needs to re-calculate the SC values for the new ordering.

4.2 Order-Sensitive Query

Besides the ancestor-descendant and parent-child relationship determinations, there are the following order-sensitive queries.

1) position = i:

Selects the ith node within a context node set. For example, the query "/play/act[2]" will retrieve the second act of the play.

2) preceding-sibling or following-sibling:

Selects all the preceding (following) sibling nodes of the context node. For example, the query "/play/act[2]/preceding-sibling::act" will retrieve all the *acts ("::act")* which are sibling nodes of act[2] and are before act[2].

3) preceding or following:

Selects all the nodes before (after) (in document order) the context node excluding any ancestors (descendants). For example, the query "/play/act[2]/following::*" will retrieve all the *nodes ("::*")* after act[2] in document order and these nodes should not be the descendants of act[2].

The Prime scheme uses the SC value and the self_label to calculate the order of each node, then it can fulfill these three types of order-sensitive queries.

From the labels only, the prefix schemes (including DeweyID, Binary and ImprovedBinary) can determine the sequence of nodes, hence they can fulfill these three order-sensitive queries.

It should be noted when inserting a node, DeweyID and Binary need to re-label the existing nodes and Prime needs to re-calculate the SC values before they can process the order-sensitive queries.

5 Performance Study

We conduct three sets of experiments (storage, query and update) to evaluate and compare the performance of the four dynamic labeling schemes, namely DeweyID, Binary, Prime and ImprovedBinary. All the four schemes are implemented in Java and all the experiments are carried out on a 2.6GHz Pentium 4 processor with 1 GB RAM running Windows XP Professional. We use the real-world XML data available in [18] to test the four schemes. Characteristics of these datasets are shown in Table 2 which shows the depths of real XMLs are usually not too high (confirmed by [12]).

Table 2. Test datasets.

Datasets	Topics	# of files	Max fan-out for a file	Max depth for a file	Total # of nodes for each dataset
D1	Bib	18	25	4	2111
D2	Club	12	47	3	2928
D3	Movie	490	38	4	26044
D4	Sigmod Record	988	26	6	39058
D5	Department	19	257	3	48542
D6	Actor	480	368	4	56769
D7	Company	24	529	4	161576
D8	Shakespeare's play	37	434	5	179689
D9	NASA	1882	1188	6	370292

5.1 Storage Requirement

The label size in Figure 5 refers to the total label size for all the nodes in each dataset. As expected, ImprovedBinary has the smallest label size for each of the nine datasets. Furthermore, the total label sizes of all the nine datasets for Binary, Prime and our ImprovedBinary are 3.97, 2.00 and 0.78 times of that of DeweyID.

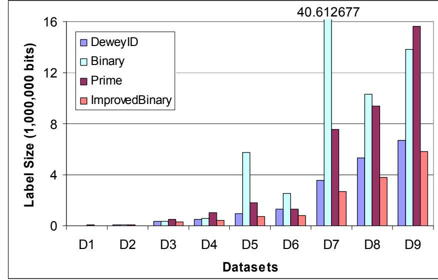

Fig. 5. Storage space for each dataset.

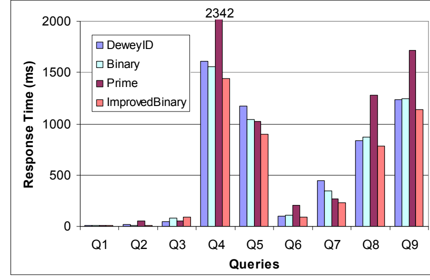

Fig. 6. Processing time of the nine queries.

5.2 Query Performance[2]

In this experiment, we test the query performance of the four schemes based on all the XML files in the Shakespeare's play dataset (D8). In order to make a more sizeable data workload, we scale up (replicate) D8 10 times as described in [14]. The ordered and un-ordered queries and the number of nodes returned from this scaled dataset are shown in Table 3. Except Q3, ImprovedBinary works the fastest for the rest 8 ordered and un-ordered queries (see Figure 6).

Table 3. Test queries on the scaled D8.

Queries		# of nodes returned
Q1	/play/act[4]	370
Q2	/play/act[5]//preceding::scene	6110
Q3	/play/act/scene/speech[2]	7300
Q4	/play/*/*	19380
Q5	/play/act//speech[3]/preceding-sibling::*	30930
Q6	/play//act[2]/following::speaker	184060
Q7	/play//scene/speech[6]/following-sibling::speech	267050
Q8	/play/act/scene/speech	309330
Q9	/play/*//line	1078330

5.3 Order-Sensitive Update Performance

The elements in the Shakespeare's plays (D8) are order-sensitive. Here we study the update performance of the Hamlet XML file in D8. The update performance of other XML files is similar. Hamlet has 5 acts, and we test the following six cases: inserting an act before act[1], inserting an act between act[1] and act[2], ···, inserting an act between act[4] and act[5], and inserting an act after act[5]. Figure 7 shows the number of nodes for re-labeling when applying different schemes.

[2] The query time and re-labeling (re-calculation) time in this paper refer to the processing time only without including the I/O time.

DeweyID and Binary have the same number of nodes to re-label in all the six cases. The Hamlet XML file has totally 6636 nodes, but DeweyID and Binary need to re-label 6595 nodes when inserting an act before act[1].

For Prime, the number of SC values that are required to be re-calculated is counted in Figure 7. Because we use each SC value for every three[3] labels, the number of SC values required to be re-calculated is 1/3 of the number of nodes required to be re-labeled by DeweyID and Binary. (Note that all the act nodes are the child nodes of the root play.)

In all the six cases, ImprovedBinary need not re-label any existing nodes and need not re-calculate any values.

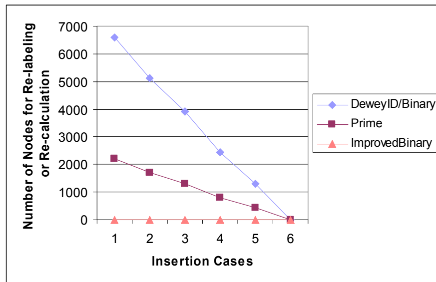

Fig. 7. Number of nodes or values for re-labeling or re-calculation.

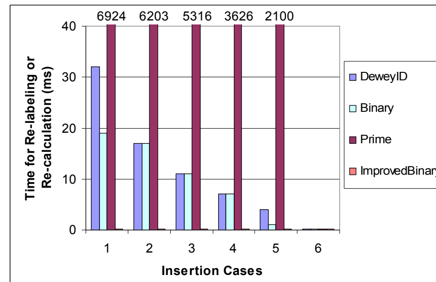

Fig. 8. Processing time for re-labeling or re-calculation.

Next we study the time required to re-label nodes or re-calculate SC values. The time in Figure 8 shows that the time required by Prime to re-calculate the SC values is at least 337 times larger than the time required by DeweyID and Binary to re-label the nodes. In contrast, our ImprovedBinary needs 0 milliseconds (ms) for the insertion in all the six cases.

6 Conclusion and Future Work

When an order-sensitive node is inserted into the XML tree, the present node labeling schemes need to re-label the existing nodes or re-calculate some values to keep the document order which is costly in considering either the number of nodes for re-labeling (re-calculation) or the time for re-labeling (re-calculation). To address this problem, we propose a node labeling scheme called ImprovedBinary in this paper, which need not re-label any existing nodes and need not re-calculate any values when inserting order-sensitive nodes into the XML tree.

In the future, we will further study how to efficiently process the delimiters of the prefix schemes and decrease the label size, as well keep the low label update cost.

---

[3] The SC values for every 4 or more nodes will be very large numbers which can not be stored using 64 bits in Java for calculation, for every 1 or 2 nodes will cause more SC values to be re-calculated.

References

1. Serge Abiteboul, Haim Kaplan, Tova Milo: Compact labeling schemes for ancestor queries. SODA 2001: 547-556
2. Rakesh Agrawal, Alexander Borgida, H. V. Jagadish: Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. SIGMOD Conference 1989: 253-262
3. Toshiyuki Amagasa, Masatoshi Yoshikawa, Shunsuke Uemura: QRS: A Robust Numbering Scheme for XML Documents. ICDE 2003: 705-707
4. James A. Anderson and James M. Bell, Number Theory with Application, Prentice-Hall, New Jersey, 1997.
5. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simon. XML path language (XPath) 2.0 W3C working draft 16. Technical Report WD-xpath20-20020816, World Wide Web Consortium, Aug. 2002.
6. Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simon. XQuery 1.0: An XML Query LanguageW3C working draft 16. Technical Report WD-xquery-20020816, World Wide Web Consortium, Aug. 2002.
7. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (XML) 1.0 third edition W3C recommendation. Technical Report REC-xml-20001006, World Wide Web Consortium, Oct. 2000.
8. Edith Cohen, Haim Kaplan, Tova Milo: Labeling Dynamic XML Trees. PODS 2002: 271-281
9. Roy Goldman, Jennifer Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB 1997: 436-445
10. Quanzhong Li, Bongki Moon: Indexing and Querying XML Data for Regular Path Expressions. VLDB 2001: 361-370
11. Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, Jennifer Widom: Lore: A Database Management System for Semistructured Data. SIGMOD Record 26(3): 54-66 (1997)
12. Laurent Mignet, Denilson Barbosa, Pierangelo Veltri: The XML web: a first study. WWW 2003: 500-510
13. Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, Sudarshan S. Chawathe: Representative Objects: Concise Representations of Semistructured, Hierarchial Data. ICDE 1997: 79-90
14. Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, Chun Zhang: Storing and querying ordered XML using a relational database system. SIGMOD Conference 2002: 204-215
15. Xiaodong Wu, Mong-Li Lee, Wynne Hsu: A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. ICDE 2004: 66-78
16. Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, Shunsuke Uemura: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Trans. Internet Techn. 1(1): 110-141 (2001)
17. Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, Guy M. Lohman: On Supporting Containment Queries in Relational Database Management Systems. SIGMOD Conference 2001
18. The Niagara Project Experimental Data. Available at: http://www.cs.wisc.edu/niagara/data.html