A Prime Number Labeling Scheme for Dynamic Ordered XML Trees

Xiaodong Wu Mong Li Lee Wynne Hsu
School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117943
{wuxiaodo, leeml, whsu}@comp.nus.edu.sg

Abstract

Efficient evaluation of XML queries requires the determination of whether a relationship exists between two elements. A number of labeling schemes have been designed to label the element nodes such that the relationships between nodes can be easily determined by comparing their labels. With the increased popularity of XML on the web, finding a labeling scheme that is able to support order-sensitive queries in the presence of dynamic updates becomes urgent. In this paper, we propose a new labeling scheme that takes advantage of the unique property of prime numbers to meet this need. The global order of the nodes can be captured by generating simultaneous congruence values from the prime number node labels. Theoretical analysis of the label size requirements for the various labeling schemes is given. Experiment results indicate that the prime number labeling scheme is compact compared to existing dynamic labeling schemes, and provides efficient support to order-sensitive queries and updates.

1. Introduction

The growing number of XML repositories on the World Wide Web has provided the momentum for the development of systems that can store and query XML data efficiently. Query languages such as XPath [6] and XQuery [4] have been designed to process XML data. Given the tree structure of XML data, path and tree pattern matching algorithms play crucial roles in the processing of XML queries. Techniques to carry out path and tree pattern matching include containment joins and structural joins whereby the pattern tree is composed by matching ancestor and descendant pairs, or parent and child nodes within lists of nodes.

In order to facilitate the determination of relationships among the nodes, nodes in XML tree are typically labeled in such a way that the ancestor-descendant relationships between any two nodes can be established quickly. Hence, a good and compact labeling scheme is crucial to efficiently process XML queries. This labeling scheme should have the following characteristics:

a) Deterministic: The relationships between two nodes can be uniquely and quickly determined simply by examining their labels.
b) Dynamic: Updating XML files will not require the re-labeling of nodes in the XML trees.
c) Compact: The size of the labels should be minimal in order to fit in the main memory.
d) Flexible: The scheme can be used to support all kinds of XQuery/XPath functions.

Early works on labeling schemes are typically range-based [11, 16]. A depth-first traversal of the XML tree is carried out to assign to each node a pair of values that cover the range of values in the labels of its descendant nodes. A test for ancestor relationship is equivalent to an interval containment test on the node labels. However, XML documents on the Web are subjected to frequent changes. As a result, such static interval-based labeling schemes require a re-labeling of the entire XML tree when frequent insertions and deletions of nodes occur.

[1, 7, 10] design a prefix-based labeling scheme to handle dynamic XML trees. The nodes in an XML tree are labeled such that the ancestor relationship test is determined by whether one label is a prefix of the other. New nodes can be inserted without affecting the labels of the existing nodes. [15] gives a labeling scheme that can be used to support order-sensitive queries. However, to the best of our knowledge, none of the existing labeling schemes are able to support dynamic updates when order is a concern.

In this paper, we propose a novel labeling scheme for XML trees that is based on the property of prime numbers. Each node is labeled by an integer, and the labeling scheme ensures that each label can only be divided exactly by its own ancestor in an XML file.

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

With the prime number labeling scheme, we are able to support dynamic updates and answer queries with ordered XQuery/XPath function. The contributions of this paper include:

a) Propose a new, dynamic, and scalable XML labeling scheme.
b) Generate a simultaneous congruence table to maintain the global order among nodes. This allows the support of ordered XQuery/Xpath functions. Together with the proposed labeling scheme, the cost for updating ordered XML is much smaller than existing schemes.
c) Construct the size model of the proposed prime number labeling scheme. Compared to existing dynamic labeling schemes, the size of the prime number labeling scheme is less affected by the fan-out of the tree. Hence, we can use fixed-length labels resulting in better storage utilization.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the property of prime number and the proposed prime number labeling scheme. An analysis of the size the prime number labeling scheme and its optimizations are also presented in this section. Section 4 discusses the incorporation of order in this labeling scheme. Section 5 gives the results of our experiments, and we conclude in Section 6.

2. Related Work

XML query is initially processed by tree traversal. In the Lore system [12], the DataGuide [9] is utilized as a summarization for the path information in the XML file. Piloted by DataGuide, the query processing system can carry out a vertical tree traversal to determine whether there exists any ancestor-descendant relationship between two nodes. However, such tree traversal-based XML query systems are costly.

To overcome this problem, a number of researchers [11, 16] propose the use of XML labeling scheme such that by comparing the labels assigned to the nodes in the XML tree, it is possible to determine the relationship between any two nodes. For example, XISS [11] employs a numbering scheme in which every node is assigned two variables: "order" and "size". These two variables represent an interval (order, order + size). For any two nodes x and y, x is an ancestor of y if and only if order(x) < order(y) < order(x) + size(x).

The value of order for a node is obtained by an extended preordering scheme. However, it is not clear how one assigns a large enough value for "size". To address the issue of assigning suitable value for "size", [16] proposes a different kind of interval-based labeling scheme. It initializes a counter to 1 and carries out a depth-first tree traversal of the XML tree. If a node is seen for the first time, it is assigned the value of the counter as its "start-point". When a node is encountered again, it is assigned the counter value as its "endpoint". The counter is always incremented by 1 each time its value is assigned to a node.

While interval-based schemes are effective in supporting XML query processing, they cannot handle dynamic updates. Insertion or deletion of nodes into a labeled XML tree may result in a total re-labeling of the XML tree. This problem may be alleviated somewhat by reserving enough space for anticipated insertions. However, it is hard to predict the actual space requirements. Thus, re-labeling after updates is inevitable for interval-based labeling schemes which are not suitable for labeling XML documents in update-intensive applications such as biding.

[2] proposes to use floating point numbers to replace integers as the labels in interval-based labeling scheme. In theory, it solves the problem of updates because one can always insert a number between any two floating point numbers. Unfortunately, in practice, the representation of a floating point number is constrained by the number of bits in the mantissa. Once again, when the number of insertions exceeds certain limits, re-labeling is necessary.

Recently, there is a trend towards dynamic labeling schemes [7, 15] where the nodes inherit their parents' labels as the prefix to their own labels. This allows one to determine the existence of an ancestor-descendant relationship by simply examining whether the prefix relationship exists in the labels of the two nodes.

The integer-based prefix labeling scheme [15] labels the n^{th} child of a node with an integer n. Each label inherits its parent's label as its prefix. However when the fan-out is larger than 10, the labeling scheme breaks down. For example, if there are two nodes with labels "2" and "21" respectively. The 11^{th} child of the first node and the first child of the second node will have the same labels, that is, "211". In this case, it will not be possible to differentiate the parent-child relationship correctly. One way to solve this problem is to use some delimiter [15]. If we use comma as the delimiter, the labels for the 11^{th} child of the first node and the first child of the second node will be "2,11" and "21,1" respectively. However, the delimiter must be stored with the label, which incurs significant overhead.

Another major prefix labeling scheme encodes the node labels as binary strings [7]. To reduce the size required for the labeling scheme, [10] proposes a

compressed prefix scheme. First, the tree is partitioned into paths to transform the tree into a balanced tree. Although this scheme is good for trees with many levels, it does not reduce the storage requirement for trees with large fan-out.

Besides the ancestor-descendant type of queries, XQuery and XPath have also provided order-based functions. Users can issue queries on the order information in XML. Hence, it becomes important to have a labeling scheme that can support this type of queries. [15] designs three kinds of labeling schemes to support ordered queries. The global approach gives the best performance but incurs high update cost. Although the local approach is the least costly for updates where order is a concern, it is unable to support all types of ordered query. The Dewey approach is similar to the dynamic prefix labeling scheme, and achieves a good tradeoff between query performance and dynamic updates.

To date, none of the existing labeling schemes can support updates for ordered XML data at low cost. When a new node is inserted into an ordered XML tree, all the existing labeling schemes require a re-labeling of the tree. A detailed survey on the various labeling schemes can be found in [5].

3. Prime Number Labeling Scheme

In this section, we describe the proposed labeling scheme that exploits the property of prime numbers. We also analyze the size requirements of the prime number labeling scheme compared to the existing labeling schemes. Finally, we discuss various optimizations that can be carried out on the proposed labeling scheme to further reduce its storage space.

Property 1 [Divisibility]: *If an integer A has a prime factor which is not a prime factor of another integer B, then B is not divisible by A.*

For example, 6 is not a factor of 10 because "3" is a prime factor of 6 but it is not a factor of 10.

We observe that in XML trees, if a node A has a descendant which is not a descendant of another node B, then A cannot be a descendant of node B. Therefore, if we label the leaf nodes in XML by prime numbers and the non-leaf nodes as a product of the labels of its child nodes, then we can easily determine the ancestor-descendent relationship by using the "divisible" property of prime numbers.

Figure 1 illustrates the basic *bottom-up* prime number labeling scheme. We start from the leaf nodes and assign prime numbers to each leaf node. For each subsequent level, we assign the parents' labels as the product of their children's labels.

Property 2 [BottomUpMod]: *In a bottom-up prime number labeling scheme, for any nodes x and y in an XML tree, x is an ancestor of y if and only if label(x) mod label(y) = 0.*

Making use of Property 2, we can quickly determine the ancestor-descendant relationship between any two nodes. It is obvious that the bottom-up approach can quickly result in relatively large numbers being assigned to nodes at the top of the tree. In addition, special handling is required for those nodes that have only one child.

Alternatively, we may consider the leaf node as the integer in Property 1 and the ancestor of this leaf node as the prime factor of the integer. This gives rise to the top-down variant of the prime number labeling scheme (see Figure 2). Clearly, the label of a node is divisible only by its ancestor's label. In this top-down scheme, each non-leaf node is given a unique prime number and the label of each node is the product of its parent nodes' label and its own label. Thus each label is a product of two factors: first factor is the number that is inherited from the label of its parent. We call it "parent-label". The second part is the value that is assigned to the node by the labeling scheme. We call it "self- label.

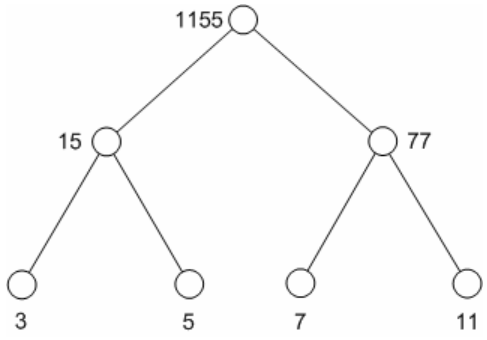

Figure 1. Bottom-up labeling scheme

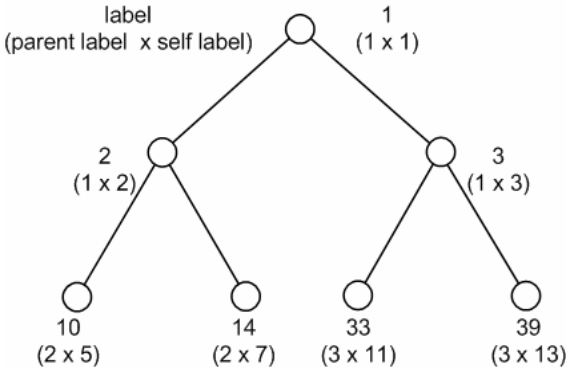

Figure 2. Top-down labeling scheme

For example, in Figure 2, the "parent-label" is 2 for the node whose label is "10", while its "self-label" is 5. Note that the top-down prime number labeling scheme is good for dynamic updates. When a new node is inserted, it is easy to simply assign a prime number that has not been assigned before as the self-label for the newly inserted node. No re-labeling is required.

In addition, the size of the label is mainly dependent on the depth of the XML tree and hardly affected by the fan-out (as in the case of bottom-up prime labeling). This makes sense as previous studies have shown that the depths of XML are usually not too high [13]. Furthermore, if the XML tree is indeed high, the tree decomposition method described in [10] can be used to reduce the height of the XML.

For the rest of this paper, the term "prime number labeling scheme" refers to the top-down labeling method.

3.1. Size Estimation

In this section, we discuss the size requirements of the various labeling schemes. This is important because the storage requirement of a labeling scheme has a direct impact on the performance of XML query processing. We use "D", "F" and "N" to denote the maximal depth, maximal fan-out and number of nodes of an XML tree respectively.

In the interval-based labeling scheme, each node is assigned two numbers that denotes the start and end points of an interval. The maximum value that these numbers can take is N where N is the number of nodes in the XML tree. In other words, the maximum size of a label for the interval-based labeling scheme is 2(1+ log (N)) [1] bits.

The binary-based prefix labeling scheme first labels the i^{th} child of a node with a binary string "$1^{i-1}0$". Next, it adds the label of the parent node as a prefix to the label of the child node. We refer to this basic prefix labeling scheme as Prefix-1. Clearly, the size of this scheme increases directly with the node fan-out in the XML tree. In fact, the maximum size of a label in Prefix-1 is

$$\text{Prefix-1: } L_{max} = D*F \qquad (1)$$

A simple optimization to reduce the overall maximal size of the prefix labeling scheme is to increment the binary representation of the labels of sibling nodes [7]. If a node label consists of all ones, then its length can be doubled by adding the same number of zeros to the label. Thus, the labels for sibling nodes will be as follows: 0, 10, 1100, 1101, 1110, 11110000. We refer to this optimized prefix labeling scheme as Prefix-2. [7] shows that the maximum size for a label in Prefix-2 is

$$\text{Prefix-2: } L_{max} = D*4\log F \qquad (2)$$

In the prime number labeling scheme, we carry out a depth-first traversal of the XML tree and assign to each node a prime number. The maximum number of bits required for a label is determined by the total number of the nodes in the XML file. Given an XML file with N nodes, we use θ_N to denote the maximal prime number that has been used to label the nodes. If the maximum level in the corresponding XML tree is D then the maximum number of bits required by the node labels at each level is given by $D\log(\theta_N)$. We assume that the bit length of the product of two numbers is the sum of the bit lengths of the two numbers.

From the characteristics of prime numbers, we know that for an integer N, the number of prime numbers that is smaller than or equal to N is N(1/log(N)). Hence, the N^{th} prime number approximately is Nlog(N) and the number of bits needed to represent the N^{th} prime number is log(Nlog(N)). Note that the error ratio for using log(Nlog(N)) to predict the length of the binary representation of the N^{th} prime number is the logarithm of the difference between Nlog(N) and the N^{th} actual prime number. Therefore, although there is fluctuation in the difference between the actual prime number and the estimated prime number, the error ratio is small. Figure 3 shows the difference between the length for the binary representation of the first 10000 actual prime numbers and the estimated prime numbers.

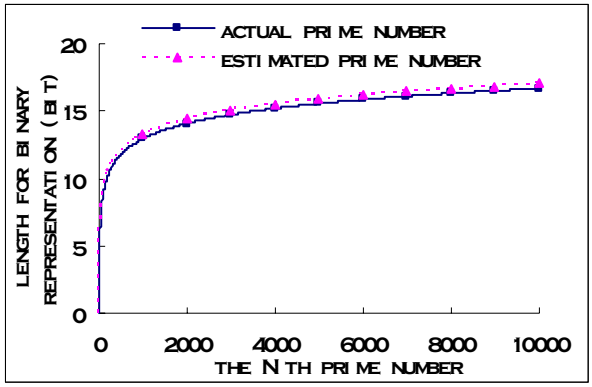

Figure 3. Actual vs. estimated prime number

Assuming the worst case for prime number labeling scheme where the XML tree is a perfect tree with fan-

[1] log is used as the logarithm to base 2.

out F and depth D, the number of nodes N in T is $\sum_{i=0}^{D} F^i$. This implies that the maximum size of a node label in T is given by

$$\text{Prime: } L_{max} = D \log((\sum_{i=0}^{D} F^i) \log(\sum_{i=0}^{D} F^i)) \quad (3)$$

Comparing the maximum label size for the three dynamic labeling schemes Prefix-1, Prefix-2, and Prime, we observe that the maximum size of a node label is determined by the product of the depth of the XML tree and the maximum size of node's self label. The latter is influenced by the fan-out of the node's parent.

From the above label size formulas, we plot two graphs to visualize the effects of fan-out and depth on the maximum size of a self label (see Figures 4 and 5). We observe that Prefix-1 increases linearly with the fan-out while the prime number labeling scheme is hardly affected by the increase in fan-out. In contrast, both Prefix-1 and Prefix-2 are not affected by the change in depth, while the prime number labeling scheme increases linearly with the depth on perfect tree. In the un-optimized prime number labeling scheme, each node is labeled with a distinct prime number as its self label. Thus, the maximum size of a self label will grow with the total number of nodes in the XML tree. This increase in label size is faster when the depth increases.

Hence, when an XML tree has a large fan-out, even for one node, but limited depth, the prime number labeling scheme can outperform the prefix labeling scheme in terms of the storage space requirement. On the other hand, if the XML document has a large depth and limited fan-out in the nodes, the prefix labeling wins. In practice, the depth of real world XML data is typically low with relatively high fan-out. [13] performs a statistical analysis on 200,000 XML documents, and discovers that 99% of these documents have less than 8 levels of nestings. Further, these documents have fan-out that can be as large as 10,000.

Overall, we see that the interval-based labeling schemes have smaller size requirements compared to the dynamic labeling schemes. However, among the dynamic labeling schemes, our prime labeling scheme gives the most compact size requirement that is the least affected by the structure of the XML tree. In other words, it is possible to use a fixed length representation for storing the labels. In so doing, we can take advantage of the standard DBMS functions for XML query processing.

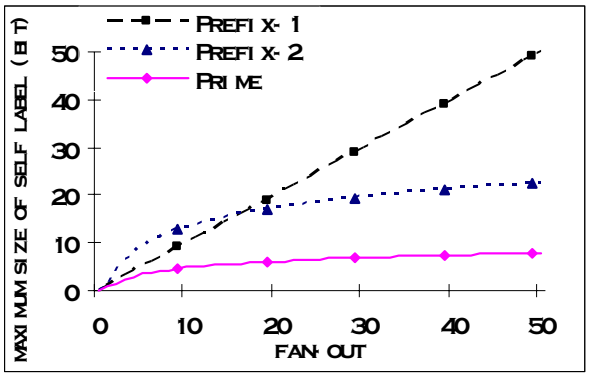

Figure 4. Effect of fan-out on size label (depth=2)

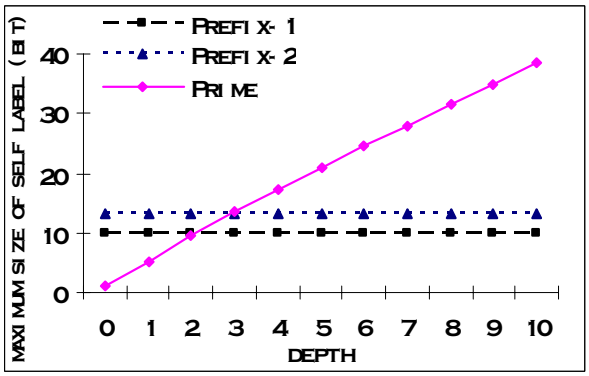

Figure 5. Effect of depth on size label (fanout=15)

3.2. Optimizations

One disadvantage of the prime number labeling scheme is that each prime number can only be used once. Hence, the self-label of a node that is subsequently inserted is always larger than self-labels of existing nodes. This implies the size of the labels will increase when the smaller prime numbers are used up. To overcome this problem, three optimization techniques are proposed to further reduce the size requirement of the prime labeling scheme.

First, we observe that the node labels that are nearer to the root of the tree have more influence on the size requirement because they are inherited by their descendants. Thus, we reserve a set of small prime numbers for labeling the root node and the nodes one level below the root.

Second, we note that the number 2 is the only even prime number. Thus, we can use even numbers such as 2^1, 2^2, …, 2^n to label the self-labels of leaf nodes, and the labels for the non-leaf nodes are odd numbers.

Since there is no ancestor–descendent relationship between the leaf nodes, we need to modify the criterion to test for ancestor–descendent relationship between two nodes as shown in Property 3.

Property 3 [OptimizedMod]: *In the optimized top-down prime number labeling scheme, for any two nodes x and y in an XML tree T, x is an ancestor of y if and only if odd(label(x)) and label(y) mod label(x) = 0.*

This optimization makes full use of the smaller prime numbers to label the nodes, thus ensuring that the space requirement is kept low. Note that this optimization causes the size of the leaf nodes to increase as fast as the prefix labeling scheme. However, it is more flexible and controllable than the prefix labeling scheme. When the size of a label in a leaf node reaches some pre-determined threshold, we can use other prime numbers instead of powers of 2 to label the remaining siblings.

Third, we discover that in many real-world XML files, some paths may occur multiple times. Consider Figure 6 where the book element has 3 authors. The path book/author in Figure 6(a) can be combined to form the XML tree in Figure 6(b). By collapsing the paths that occur many times in the XML tree, we can reduce redundancy and further decrease the size of the labels. In the case where order is important, we can maintain the position information at the leaf nodes to indicate their orders among the siblings. Note that this optimization is only applicable for answering "ancestor-descendant" queries. If order among sibling nodes is important, then this optimization cannot be applied. Since this optimization is relevant for all kinds of XML labeling schemes, we will not use it in our comparative study.

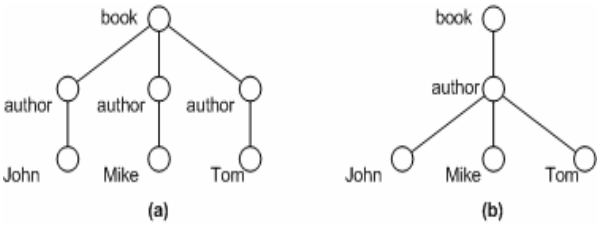

Figure 6. Combining repeated patterns

Finally, we observe that the prime number labeling scheme can also benefit from the tree decomposition approach described in [10] when the depth of the tree is high. In the prime number labeling scheme, each node inherits its parent's label. Thus, we can decompose an XML tree into several sub-trees. The nodes in each sub-tree are first labeled separately. A global tree that comprises of the root nodes of these sub-trees is constructed and labeled. [10] finds that this tree decomposition approach can effectively reduce the label size of dynamic labeling schemes for trees with great depths.

Figure 7 gives the details of the algorithm to label nodes with the prime number labeling scheme. The algorithm incorporates the two optimization techniques described in the previous section, namely, reserving a small set of prime numbers to label the top-level nodes in the tree, and using powers of 2 to label the leaf nodes.

Three functions are called in the algorithm: *getReservedPrime()* returns a prime number from the set of reserved small prime numbers for the top level nodes in the XML tree; *getPrime()* returns the next smallest prime number to be used for the node label; and *getPower2(n)* returns the number 2^n to label the n^{th} leaf node.

```
Algorithm: PrimeLabel
Input:  XML document
Output: Label for each node
begin
for each node n in the XML document do
   n.childNum = 0;
 if (n is the root node ) then n.label = 1;
 else      parent = n.parent;
       if (n is a non-leaf node)
       then  if (this node is a top level nodes)
                then n.selfLabel= getReservedPrime();
                else  n.selfLabel = getPrime();
       else   parent.childNum++;
              n.selfLabel=GetPower2(parent.childNum);
  output (parent.label*n.selfLabel);
end
```

Figure 7. Algorithm PrimeLabel

4. Labeling Ordered Trees

The elements in XML are intrinsically ordered. Suppose we have the XML document on a book with 3 chapters. We will have 3 chapter tags in the XML document. Looking at the order of occurrences of these 3 chapter tags, we may infer that the first chapter tag that occurs right after the start element of the XML node gives the details of the first chapter of the book. The next chapter tag that occurs after the first chapter tag describes the details of the second chapter of the book, etc. An example of an ordered XML tree is shown in Figure 8.

Order in XML is important as users may be interested in issuing order-sensitive query. For example, the query book[author[2]='John'] retrieves a list of books whose *second* author is "John".

In general, order-sensitive queries in XML can be divided into three types:

a) Preceding, Following:

This class of queries selects all the nodes before (after) the context node excluding any descendants (ancestors). For example, the query

paper/title[1]/Following::author

will retrieve all the elements following "paper/title[1]".

b) Preceding-sibling or Following-sibling:

This class of queries selects all the preceding (or following) sibling nodes of the context node. For example, the query

paper/author[2] /Following-sibling::*

will retrieve the sibling nodes of "paper/author[2]". The order of the requested node should be larger than the context node.

c) Position=n:

This class of queries selects the n^{th} node within a context node set. For example, the query

book/author[2]

will retrieve the second author of this book.

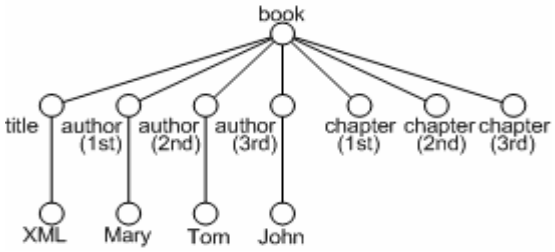

Figure 8. Example of an ordered XML tree

This need to maintain order within XML creates a problem for many XML labeling schemes. For example, if we need to insert a new author as the second author to the XML tree in Figure 8, then we would have to push Tom and John to the 3rd and 4th sibling positions respectively. For interval-based labeling schemes and prefix labeling schemes, a re-labeling is required.

To the best of our knowledge, none of the existing labeling schemes is able to handle the three types of order-sensitive queries in the presence of dynamic updates effectively. In this paper, we make use of the Chinese remainder theorem to maintain order in our prime number labeling scheme. This allows our prime number labeling scheme to answer order-sensitive XML queries, and cope with updates at low cost.

Definition 1 [GCD]: Let N be the set of integers. Given a set of integers $m_1, m_2, \ldots m_k$, the GCD ($m_1, m_2, \ldots m_k$) = max ({f| m_1 mod f = 0, f ∈ N} ∩ { f | m_2 mod f = 0, f ∈ N}…∩ { f | m_k mod f = 0, f ∈ N}).

Theorem 1 [Chinese Remainder Theorem] [3]**:** Let M = [$m_1, m_2, \ldots m_k$] and N = [$n_1, n_2, \ldots, n_k$] be two lists of integers. If the GCD ($m_1, m_2, \ldots m_k$) = 1, then the simultaneous congruence SC (M, N) = x satisfies $\begin{cases} x \bmod m_1 = n_1 \\ x \bmod m_2 = n_2 \\ \ldots\ldots \\ x \bmod m_k = n_k \end{cases}$, and there is exactly one solution x between 0 and C, where $C = \prod_{i=1}^{k} m_i$.

There are many algorithms to compute the simultaneous congruence of two sets of integers. One of these methods is to use the following Euler's quotient function:

$$X = \left(\sum_{i=1}^{k} (C/m_i) * n_i * \phi(m_i)\right) \bmod C$$

where $C = \prod_{i=1}^{k} m_i$ and $\phi(x)$ is Euler's totient function [3] which is defined as the total number of integers which are smaller than x and relatively prime to x. The complexity of Euler's totient function is O(n). Therefore, the cost to compute simultaneous congruence of two sets of integers is acceptable.

4.1. Ordering with Simultaneous Congruence Values

Given a list of prime numbers P = [3, 4, 5], and a list of integers I = [1, 2, 3], the Chinese remainder theorem states that there exists a number x = 58 where

$$\begin{cases} x \bmod 3 = 1 \\ x \bmod 4 = 2 \\ x \bmod 5 = 3 \end{cases}$$

This allows us to generate a one-to-one mapping between the elements in P and I. Thus, when the prime numbers in P are self-labels of the nodes in an XML tree, the integers in I actually depict the ordering of these nodes. We can use the number SC(P, I) = x to capture the global ordering for an XML document.

Figure 9 shows an XML tree that has been labeled using the prime number labeling scheme. The order number of a node in the XML is given by the integer within the node. The order number of the root node is defined to be 0.

The SC value that is generated from self-labels and the order number of the nodes for this XML tree is 29243. Thus, the global ordering for each node can be subsequently derived from the formula: SC mod (self-label).

For example, the order number for the node whose self-label is 5 is 3, that is, 29243 mod 5.

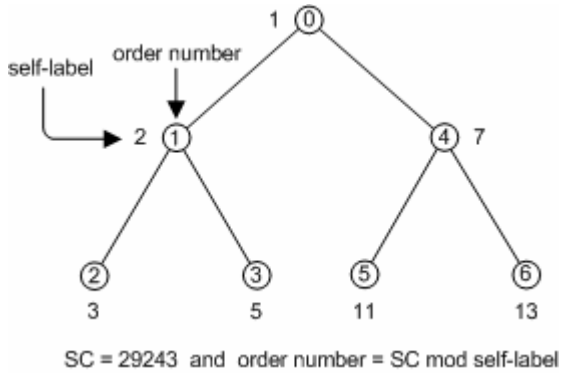

Figure 9. Capturing order by an SC value

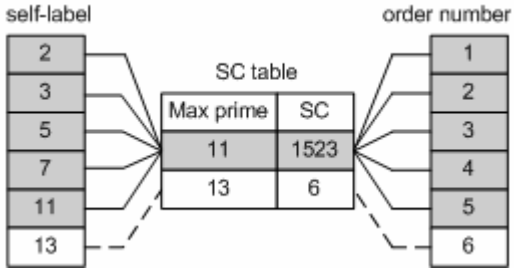

Figure 10. SC table for XML tree in Figure 9

In practice, the XML tree may be large, thus requiring a large SC value to capture the ordering of the nodes. An alternative way is to use a list of SC value instead of a single SC value. Each SC value maintains the global ordering of a subset of the nodes in the XML tree.

Consider Figure 10 where we use two SC values to capture the ordering of the nodes in the XML tree in Figure 10. The SC value for the self-labels of the first 5 nodes is 1523, while the SC value for the single 6th node is 6. At the same time, we record the maximum prime number for each SC value in the SC table. These maximum prime numbers will indicate the set of nodes whose ordering is captured by the corresponding SC value.

4.2. Ordering after Updates

In this section, we discuss how the ordering of the nodes can be easily maintained by using the SC values when new nodes are inserted. Note that the deletion of nodes from an XML tree does not affect any node ordering.

Suppose we insert one node into the XML tree in Figure 9. Figure 11 shows the updated XML tree with new nodes in the dashed line rectangle. The order number for the new node whose self-label is 17 is 3. We search for the largest maximum prime number that is stored in the SC table, and update it to 17. The corresponding SC value for this record is also updated to the new simultaneous congruence value that satisfies the following two equations: $\begin{cases} x \bmod 13 = 7 \\ x \bmod 17 = 3 \end{cases}$

The order numbers for the nodes that comes after the newly inserted node will be increased by 1. Thus, the SC values associated with these nodes need to be updated accordingly. In this example, the first record of the SC table contains the order number that need to be changed. The new simultaneous congruence value for this record is computed according to the following equations: $\begin{cases} x \bmod 2 = 1 \\ x \bmod 3 = 2 \\ x \bmod 5 = 4 \\ x \bmod 7 = 5 \\ x \bmod 11 = 6 \end{cases}$

Figure 12 shows the updated SC table for the XML tree in Figure 11. Since an SC value can capture the order numbers for several nodes in an XML tree, updating the ordering information of these nodes can be performed by updating the SC value. Therefore, the cost of updating the SC table is relatively low compared to the cost of updating the order number directly.

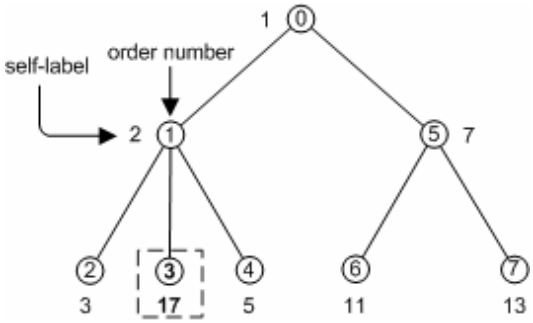

Figure 11. Updated XML tree

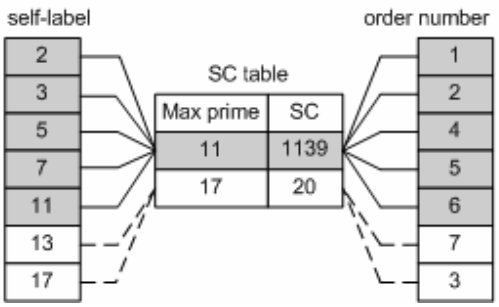

Figure 12. SC table for the XML in Figure 11

4.3. Answering Ordered Queries

Finally, we illustrate how the three types of ordered queries can be answered using the proposed prime number labeling scheme and the SC table.

The Preceding, Following query can be answered by simply comparing the ordering of the nodes. The Position=n, Preceding-sibling and Following-sibling queries can be evaluated using the following strategy.

Consider the query "paper/author[2]". We first retrieve all the author nodes who are the descendants of "paper". Next, we generate the order numbers for these author nodes using their self-labels and the SC table. The author nodes are sorted first according to their order numbers. Finally, we return the author node that is in the second position.

Similarly, for the query "paper/author/Following-sibling::author[2]", we first obtain a list of author nodes as before, and output the nodes whose positions are larger than 2 in this list.

5. Performance Study

We implemented in JAVA the three labeling schemes: interval-based labeling scheme (Interval) [11], prefix-based labeling scheme (Prefix-2) [7], and the optimized prime number labeling scheme (Prime). Note that Interval is representative of existing static labeling schemes, such as floating point or integer-based interval labeling scheme, while Prefix-2 is representative of dynamic labeling schemes, such as Dewey or prefix labeling scheme. We incorporate two optimizations in Prime, namely, reserving a set of small prime numbers for the top level nodes, and labeling the leaf nodes with different powers of 2. Since the optimizations of combining repeated paths and tree decomposition are not restricted to just the prime number labeling scheme, we do not include them in our comparative study.

The three labeling schemes are used to label the 6224 real-world XML files available in [14]. Table 1 shows the characteristics of the various datasets used.

Four sets of experiments are carried out. The first set examines the space requirements. The second set compares the processing time for ordered and unordered queries of the dynamic labeling schemes. The third set evaluates effect of un-ordered updates. The last set of experiments studies the effect of order-sensitive updates.

In the experiments, the labels of the XML are stored in a relational database with a table structure that is similar to that in [15]. All the queries are first transformed into SQL using the approach in [15]. Operations that are used by interval-based labeling scheme such as ">","<", and the prime number labeling scheme such as "mod", ">", "<", "=" are directly supported by the database system. The operation "check prefix" used in the prefix labeling scheme is defined as a user-defined function. Experiments were carried out on an Intel i586 PC 1.6 GHz with 256 MB RAM. The buffer pool used is 128 MB.

Table 1. Characteristics of datasets

Dataset	Topic	Max. # of nodes
D1	Sigmod record	41
D2	Movie	125
D3	Club	340
D4	Actor	1110
D5	Car	2495
D6	Department	2686
D7	NASA	4834
D8	Shakespears' Plays	6636
D9	Company	10052

5.1. Space Requirements

In this section, we give the results of two sets of experiments. The first set examines the effect of the optimizations on the label size. The second set compares the label size for the three labeling schemes.

5.1.1. Sensitivity Experiment

We evaluate the space requirements for the original top-down prime number labeling scheme and the three optimizations which are described in Section 3.2, that is, reserve a set of small prime numbers for the top-level nodes (Opt1), use powers of 2 to label the leaf nodes (Opt2), and combine the same paths in the XML tree (Opt3). Figure 13 shows the result.

Compared to the original prime number labeling scheme, the improvement for Opt1 is limited. The latter works best when the XML document contains a large number of nodes. In this situation, the original prime number labeling scheme will label a top level node of large XML with a big prime number. Since each node inherits its ancestor's label, the strategy in Opt1 will decrease the maximum size of the node label by using small prime numbers for the top-level nodes.

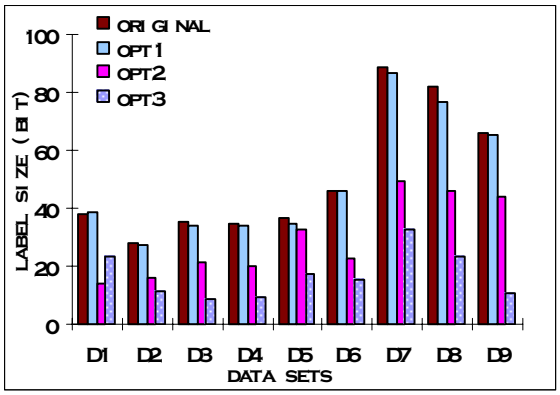

Figure 13. Effect of optimizations on space requirement

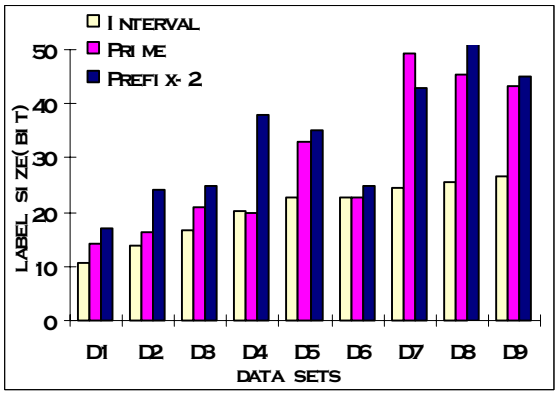

Figure 14. Space requirements for the various labeling schemes

On the other hand, Opt2 will greatly decrease the size of the node labels. It is able to achieve up to a 63% reduction in the maximum label size. The main reason for this vast reduction is that the majority of the nodes in the various XML datasets are leaf nodes. The ratio of an internal node to a leaf node is about 2.6 : 1.

The optimization Opt3 can further decrease the size of the node labels. Combining the same paths can reduce up to 83% of the maximum label size. This is because many real world datasets conform to some DTD, and have many repeating patterns.

5.1.2. Comparative Experiment. Next, we study the space requirements for the three labeling schemes. We compare the size of fixed length labels, that is, the length of label is determined by the maximal length of labels in the data set. The results for the 9 datasets are shown in Figure14.

As expected, the maximum label size for the interval-based labeling scheme is smaller compared that in the prefix and prime number labeling schemes. Although the optimization in Prefix-2 is able to reduce its storage requirement, the prime number labeling scheme shows the best savings in storage space for the majority of the datasets.

Careful examination reveals that the movie dataset D4 contains a list of movies for an actor. This dataset has a huge fan-out. As a result, the prefix labeling scheme suffers badly. In contrast, dataset D7 is the NASA document that has a high depth with low fan-out. This structure is ideal for the prefix labeling scheme.

From the experiments, we see that prime number labeling scheme is the most compact among all the dynamic labeling schemes, and is the least affected by the structure of XML tree.

5.2. Response Times

In this set of experiments, we investigate the performances of the various labeling schemes when processing ordered and unordered queries. We use the Shakespears' Play dataset (D8) in this experiment. To ensure that the number of nodes retrieved is substantially large, we replicate the Shakespears' Play dataset (D8) 5 times as carried out in [15].

Table 2 shows the 9 queries issued on this dataset.

Table 2. Test queries

Query		Number of Nodes Retrieved
Q1	/play//act[4]	185
Q2	/play// act[3]//Following::act	370
Q3	/play//act//persona	969
Q4	/act[5]//Following::speech	60105
Q5	/speech[4]//Preceding::line	66946
Q6	/play//act[3]//line	108500
Q7	/act// Following-Sibling::speech[3]	143725
Q8	/play//speech	154755
Q9	/play//line	538955

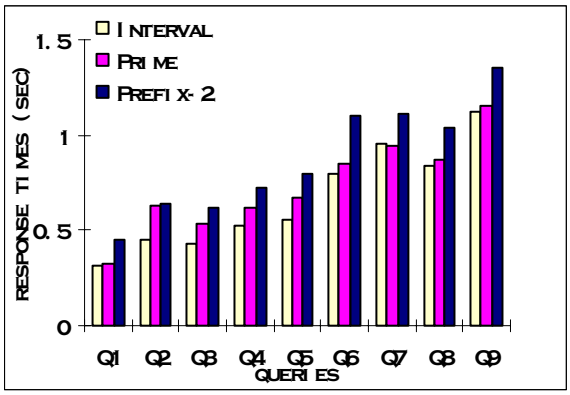

Figure 15. Response time for queries

The results of the experiments are shown in Figure 15. In general, the prime number labeling and the interval-based labeling scheme can process ordered queries faster compared to the prefix-based labeling schemes. This is because the prefix labeling schemes use a user-defined function to retrieve data. Further, the node labels in the prefix labeling schemes are relatively large, and may incur additional disk I/Os.

5.3. Effect of Un-ordered Updates

In this set of experiments, we compare the number of existing nodes that need to be re-labeled when a new node is inserted to the XML tree using different labeling schemes. The deletion of nodes does not affect the labels of other nodes. Hence, we do not need to be concerned about deletion here.

We select 10 XML files whose size ranges from 1000 to 10,000 nodes. We first examine the performance of updating the leaf nodes. We add a new node as the sibling of the node on the deepest level in the XML tree and count the number of nodes whose labels need to be re-labeled after the insertion.

Figure 16 shows the comparison of the cost of updates on the leaf nodes. The dynamic labeling schemes, both prime and prefix, are not affected by the size of the XML file. The number of nodes that need to be re-labeled for the prefix labeling scheme is 1, which is essentially the inserted node. The optimized prime number labeling scheme needs to re-label 2 nodes, that is, the newly inserted node and its parent. This is because the parent node is previously a leaf node, and has been labeled using 2^n for some n. After insertion, we need to change this label to a prime number. Note that the original prime number labeling scheme will only need to re-label the new node. In contrast, the number of nodes that need to be re-labeled in the interval-based labeling scheme grows dramatically with the increase in the number of nodes in the XML file.

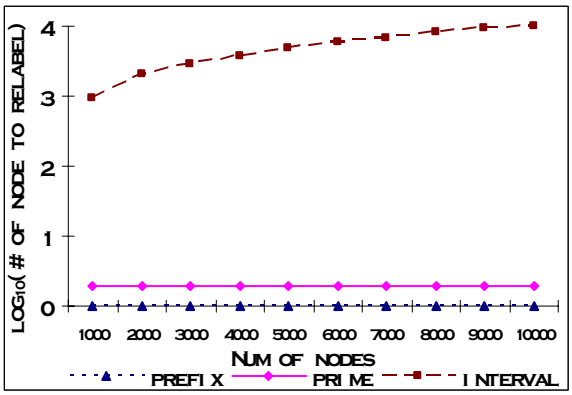

Figure 16. Update on leaf nodes

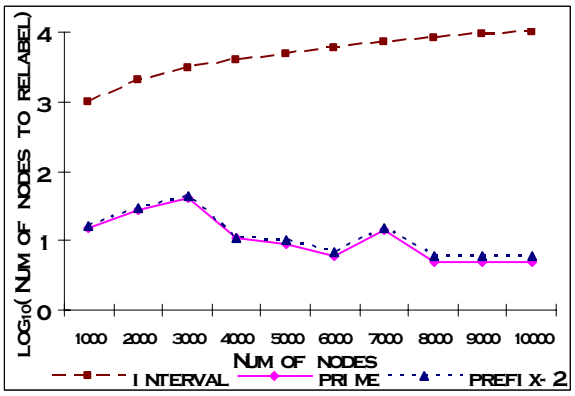

Figure 17. Update on non-leaf nodes

Next, we examine the performance of the three labeling schemes when updating the non-leaf nodes. We insert a node as a parent of the first level 4 nodes in the SAX parse order and count the number of nodes that needs re-labeling. Figure 17 shows the experiment results.

For the interval-based labeling scheme, all the nodes that come after the newly inserted node in SAX parse order require re-labeling. For both the prefix and prime number labeling schemes, only the descendents of the newly inserted node need to be re-labeled. This is clearly a subset of the nodes that require re-labeling in the interval-based labeling scheme.

From this experiment, it is clear that dynamic labeling schemes perform better than static labeling schemes in the presences of updates. The number of nodes that require re-labeling is almost the same for the prefix and prime number labeling schemes.

5.4. Effect of Order-Sensitive Updates

In the final set of experiments, we compare the performances of order-sensitive updates using the various labeling schemes. We update the Hamlet XML file in the Shakespears' Play dataset. Since the file contains a list of ordered ACT element nodes, we insert a new ACT node between each of these nodes in the list. We count the number of nodes that needs to be re-labeled after each insertion. For this experiment, we use one SC value to maintain the order of 5 nodes. We consider a record update in the SC table as a node that requires re-labeling.

The result is shown in Figure 18. Note that the number of nodes that require re-labeling for an order-sensitive query is very high for the prefix and interval-based labeling schemes. It is clear show that none of the existing labeling schemes is able to handle order-sensitive updates efficiently. For the prime number labeling scheme, only the SC table needs to be updated. Since one SC number can capture the ordering of several nodes, the cost to update SC table is much less than the cost to update the node labels. Therefore, the SC table in the prime number labeling scheme reduces the cost for order-sensitive updates.

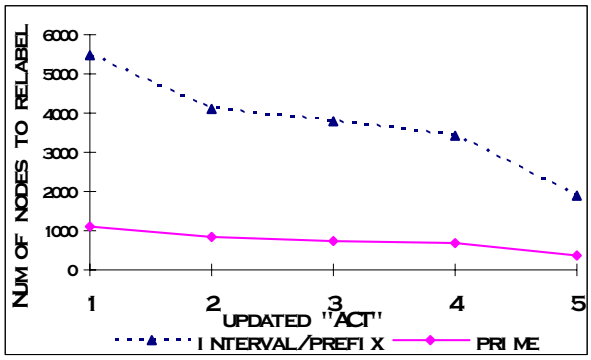

Figure 18. Order-sensitive updates

6 Conclusion

The objective for designing labeling schemes for XML trees is to allow quick determination of the relationships among the element nodes without actually accessing the XML file. Motivated by the need to efficiently support queries and updates in ordered XML trees, we have developed a prime number labeling scheme that utilizes the unique characteristics of prime numbers to capture the ancestor-descendant relationship between two nodes. An analytical study of the size requirements of the prime number labeling scheme, and the existing dynamic prefix-based labeling schemes indicate that the proposed scheme is compact and hardly affected by the fan-out of the XML trees. Several optimizations have also been designed to further reduce the size of the scheme.

We also apply the Chinese Remainder Theorem to generate a simultaneous congruence table to maintain the global order among nodes. This allows ordered queries to be answered efficiently. Experiment results indicate that the proposed prime number labeling scheme, together with the simultaneous congruence table, is able to efficiently process both ordered and unordered queries while maintaining low update costs.

References

[1] S. Abiteboul, H. Kaplan and T. Milo, Compact Labeling Schemes for Ancestor Queries, In SODA, 2001.

[2] T. Amagasa, M. Yoshikawa and S. Uemura, QRS: A Robust Numbering Scheme for XML Documents (Poster), In ICDE, 2003.

[3] J. Anderson and J.M. Bell, Number Theory with Application, Prentice-Hall, New Jersey, 1996.

[4] D. Chamberlin et.al, XQuery 1.0: An XML Query Language, W3C Working Draft, 2001.

[5] V. Christophides, D. Plexousakis, M. Scholl and S. Tourtounis, On Labeling Schemes for the Semantic Web, In WWW, 2003.

[6] J. Clark and S. DeRose, XML Path Language (XPath) Version 1.0, W3C Recommendation, 1999.

[7] E. Cohen, H. Kaplan and T.Milo, Labeling Dynamic XML Tree, In PODS, 2002.

[8] B. Cooper, N. Sample, MJ. Franklin and GR. Hjaltason, A Fast Index for Semistructured Data, In VLDB, 2001.

[9] R. Goldman and J. Widom, Dataguides: Enabling Query Formulation and Optimization in Semistrucutred Databases, In VLDB, 1997.

[10] H. Kaplan, T.Milo and R. Shabo, A Comparison of Labeling Schemes for Ancestor Queries, In SODA, 2002.

[11] Q. Li and B. Moon, Indexing and Queryring XML data for Regular Path Expressions, In VLDB, 2001.

[12] J. McHugh et al, Lore: A Database Management System for Semistructured Data, In SIGMOD, 1997.

[13] L. Mignet, D. Barbosa and P. Veltri, Web XML: A First Study, In WWW, 2003.

[14] Niagara Project, http://www.cs.wisc.edu/niagara/

[15] Tatarinov, S.D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita and C. Zhang, Storing and Querying Ordered XML Using a Relational Database System, In SIGMOD, 2002.

[16] M. Yoshikawa and T.Amagasa, XRel: A Path-based Approach to Storage and Retrieval of XML Documents

Using Relational Databases, In ACM Transactions on Internet Technology (TOIT), 2001

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE