A Research Study on the Issues Related to XML Updates

Manesh Subhash Ni Yuan Sun Chong
National University of Singapore
School of Computing
{*maneshsu, niyuan, sunchong*}*@comp.nus.edu.sg*

Abstract

Element labeling and structural indexes for XML data are the important techniques provided for XML query processing, which should be consistent with the corresponding XML document. However, XML data on the Web are subjected to frequent updates. The updates on XML data may incur the changes on element labels and structural indexes. In order to avoid the expensive cost for re-labeling elements or reconstructing indexes, several approaches have been proposed to handle the maintenance of these issues during XML updates. In this report, we study these approaches in detail and compare the performance of them.

1 Introduction

XML (eXtendible Markup Language) [2] has become the *de facto* standard for data representation and information exchange over the Internet. A large amount of data on the Web is in the format of XML, which provides a momentum for the development of querying XML document. W3C has recommended several XML query languages such as XPath [1], XQuery [3]. The efficiency for XML query processing is very important. In order to facilitate the query processing, two kinds of additional information for the XML document are provided, i.e. labels for XML element and structural index built on XML document.

Labels are assigned to the nodes in XML document in such a way that the ancestor-descendant relationships between any two nodes can be determined quickly. The labeling mechanism is widely used in structural joins [4, 6] approaches for XML query processing. One problem for the labeling scheme which is used in those approaches is that the updates on the original XML document may incur to relabel a large part of nodes in the document, which is very expensive.

Building structural index is another way to accelerate XML query processing. Several structural indexes have been proposed, i.e. 1-index [13], A(k)-index [12], D(k)-index [14], and M(k)-index [9], which summarize the structure of XML data to support the evaluation of path expressions. Since the structural index is a summary of the XML data, it should be consistent with the XML data. However, the updates on the original XML document may cause the significant change on the structural index, even the update is just to insert an edge in the original XML graph. To make the index consistent with the XML document, one naive approach is to rebuild the structural index which is also very expensive.

The common requirements for both labels and structural indexes are that they should be consistent with XML document, while the size of them should be small. Several labeling schemes which are suitable for updates have been proposed [7, 18, 5, 19]. The PREFIX1 and PREFIX2 proposed by Cohen et al [7] can avoid relabeling completely, and prime number labeling scheme [18] provides good performance in the case of large fan-out. In this report, we study these labeling schemes and compare the *pros and cons* of those schemes. XPath query can be divided into *ordered* query and *unordered* query [1], and the processing of *ordered* query will also be affected by the updates, therefore we also

study an algorithm to maintain the order information of XML nodes. Finally, we study the algorithm to maintain the structural indexes during updates.

The rest of the report is organized as follows. In section 2, we introduce the background information, such as the basic labeling schemes and the structural index. In section 3, we study the XML update issues. We first discuss the schemes for labeling dynamic XML trees. Then we present the order maintenance of XML updates. At the end of the section, we study the maintenance of structural indexes. In section 4, we study and compare the performance of different algorithms. The conclusion is given in section 5.

2 Background

2.1 Labels for XML Document

The query processing in XML database always involves determining ancestor-descendant structural relationship in addition to parent-child structural relationship, since users may not know the exactly structure of the XML document in the absence of DTD. In the navigation based query processing, the nodes which matched with the ancestor have to been kept for a long time to wait for the matching descendants. In the index based query processing, there are two options: one is that we maintain only parent-child node pairs and obtain ancestor-descendant node pairs through repeated joins, which will take too much query processing time; the other is that we maintain all ancestor-descendant relationship, which will lead too much space cost.

To handle the difficulty of determining ancestor-descendant structural relationship, the approach that uses labels to represent the positions of the elements and string values in XML document has been proposed, which makes the checking ancestor-descendant structural relationships as easy as checking parent-child structural relationships. There are two main types of labeling strategies, i.e. *interval* based and *prefix* based.

2.1.1 Interval based labeling for XML document

The interval based labeling scheme is first suggested by Santoro and Khatib [15]. Yoshikawa and Amagasa [21] also proposed a variant of interval based XML labeling scheme. The principles for interval based labeling scheme is that numbering leaf nodes from left to right in ascending order, and then label each node with a pair consisting of the numbers of its smallest and largest leaf descendants. The checking of ancestor-descendant structural relationship is converted to determine the containment of two intervals.

An interval based XML labeling scheme, which is used by many structural join algorithms as a representation for XML document [4, 6], is to represent the position of an element occurrence in XML document as a 3-tuples (DocID, StartPos : EndPos, Level), where DocID is the identifier of the document; StartPos and EndPos can be generated by counting word numbers from the beginning of the document DocID until the start and end of the element, respectively; Level is the depth of the element in document DocID. Then node $(D_1, S_1 : E_1, L_1)$ is an ancestor of node $(D_2, S_2 : E_2, L_2)$ iff $D_1 = D_2$ and $S_1 < S_2$ and $E_2 < E_1$.

2.1.2 Prefix based labeling for XML document

Some researchers have proposed the prefix based scheme to label the elements [10]. For prefix based labeling scheme, the label for a node consists of two parts: one part is inherited from its ancestors and the other part is the label assigned to the node itself. The prefix based scheme requires that for a certain node u, u's label is the prefix of labels of all its descendant, while u's label should not be the prefix of the label of any node which is not u's descendant. Then to determine whether node n_1 is an ancestor of node n_2, the prefix based labeling scheme checks whether label for n_1 is a prefix of label for n_2.

2.2 Indexes for XML Document

XML is kind of *semi-structured* data which is "schema-free", it sometimes may have some kind of structure, but it is usually too varied, irregular to be easily mapped to a fixed schema. Without a schema or some structure information, the basic operation of querying would become computationally expensive and would resort to exhaustive searches in most cases. Indexes have been such an efficient traditional technique to facilitate the queries on a large database, it is possible to be built based on the pre-defined schema like in relational database and Object Oriented databases, however, it is not as quite as simple in the case of semi-structured data, some novel indexes have been proposed based on the structure of the XML data not on the value of the data elements, such as DataGuides [8], 1-index [13], A(K)-index [12], D(K)-index [14] and so on. The basic concept of these indexes is to summarize the graph structure of the original XML data and keep the information in a concise graph with much fewer nodes and labels. Thus queries that should be implemented on the original XML data, now could be efficiently processed on the newly built data structure or index.

In this section, we first introduce several indexes. According to the relationship and characteristic of the indexes, they would be separated into two parts:

- Strong DataGuides. This is one of the first structure summaries.
- 1-index, A(k)-index, D(k)-index. These three indexes are in one family and all are based on the similarity or simulation.

2.2.1 Strong DataGuide

The Strong DataGuide [8] is one of the first structure summaries that are used to facilitate graphical browsing and queries.It presents a graph structure dynamically generated directly from a database without regard to any fixed schema. The Strong Data Guides is based on the Object Exchange Model(OEM). In OEM, each object contains an object identifier(oid) and a value. A value can be many kinds of values such as integers, reals, strings, images, programs, or any other data considered indivisible.These atoms values could be combined together to form a complex value.Each object in the OEM is connected to the parent via a descriptive label.

Based on the data model, the Data Guides is defined as following: A *DataGuide* for an OEM source s is an OEM object d such that every label path of s has exactly one data path instance in d, and every label path of d is a label path of s. This definition is to make sure that each label path in the source appears exactly once in each DataGuide and neither DataGuide introduces any new paths that do not exist in the source. From the perspective of computational theory,creating a DataGuides over a database is equivalent to conversion of a non-deterministic finite automaton(NFA) to a deterministic finite automaton(DFA). As far as we know, a single NFA may have many equivalent DFAs. Similarly,one OEM source database may have multiple DataGuides. While the Strong DataGuide is one of the multiple DataGuides in which each set of label paths that shares the same target set in the DataGuide is the set of label paths that shares the same target set in the source. **Fig** 1 shows an example for Strong DataGuide.

2.2.2 Index Family

DataGuide records the existing labels in the original database and could be used as an index, while it is only suitable for much simple expressions. For more complex expressions, new structure summaries based on bi-similarity are introduced. The basic idea of indexes based on the simulation is still the same as that of DataGuide: preserving all the paths in the data graph in the structure summary while keeping far fewer nodes and edges. The basic concept for building an index is as following: grouping database objects into equivalent classes with objects that are indistinguishable to a certain path class. How to build the equivalent classes decides the difference of the indexes. To compute the complete equivalent relation is considered as PSPACE complete while finer equivalence classes based on bi-simulation would be efficiently computed.1-index is based on the bi-similarity; Both A(k)-index

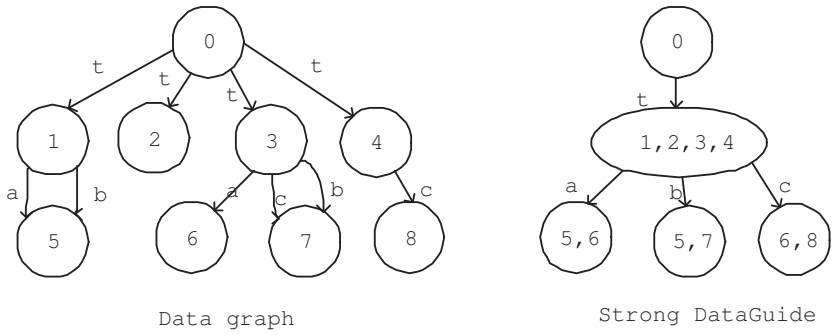


Figure 1: Strong DataGuide

and D(k)-index are based on the k-bisimilarity, but D(k)-index is more sensitive to the work load and could greatly improves the efficiency of the query evaluation.

Data model XML or other semi-structured data could be model as a directed labeled graph G=(V_G, E_G, root, $\sum_G$,*label*,*oid*, *value*). Each edge in the E_G indicates an object-subobject or IDREF relationship. "Simple" nodes in the V_G have no outgoing edges and are given a value via the *value* function. Each node in the V_G is labeled with a string-literal from $\sum_G$ via the *label* function and with a unique identifier defined by the *oid* function, with simple objets given the distinguishable label, **value**. There is a single root element with the distinguished label, **root** which has no incoming edges.

Here, we would introduce some terminology for the paths and path-evaluation.

- A *node path* in G is a sequence of nodes, $n_0...n_p$ such that an edge exists between node n_i and node n_{i+1}, for 0≤i≤p-1.
- A *label path* is a sequence of labels $l_0...l_p$ and a node path matches a label path if label $n_i = l_i$, for 0 ≤i≤ p

Then for path evaluation, the *extent* of a node A ($ext^I(A)$) is a subset of V_G, which includes the nodes which have equivalence relationship with node A.

- Safe: if $l_0l_1...l_k$ is a label path which matches a path to node v in G, there must be some node A in *I(G)* for which $l_0l_1...l_k$ matches a path to A and $v \in ext^I(A)$.
- Precise: if the reverse hold for the safe.

The safe is used to make sure that the index could include all the labels , thus the query result of the original data would always be contained in that of query result using the index. We could be sure that none of the result would be left out based on the extent mapping. Safe is the necessity for an index to some certain degree. While precise is a much more strict constraint. Sometimes, the more precise, the much larger the size of the index. Generally, it is much more suitable to find the tradeoff for the precise and size of the index. Because the size is a very important property of index and would have great influence on the speed of the queries on the indexes.While if the index is not exact precise, a validate procedure would be necessary. It would be important to find the balance the size and precise. 1-index is more precise and much larger compared with A(k)-index and D(k)-index. Later we would make a much more detailed comparison.

Similarity Simulation is the base for 1-index, A(k)-index and D(k)-index. According to a certain kind of simulation, these indexes combine nodes into small sets, therefore cutting lots of edges. The simulation is always achieved in local area of the data graph, while reaches a much large scope with certain constraint. The following are the definitions for the bisimilarity and k-bisimilarity:

- Bisimilarity: A symmetric, binary relation $\approx$ on V_G is called a bisimulation if, for any two nodes u and v with $u \approx v$, we have that:
  1. u and v have the same label, and
  2. if u'is a parent of u , then there is a parent v'of v such that $u' \approx v'$, and vice-versa.
  Two nodes u and v in G are said to be bisimilar, denoted by $u \approx^b v$, if there is some bisimulation $\approx$ such that $u \approx v$.

- K-bisimilarity: It is defined inductively.
  1.For any two nodes, u and v, $u \approx v$ have the same label.
  2.Node $u \approx v$ *iff* $u \approx^{k-1} v$ and for every parent u'of u , there is a parent v' of v such that $u' \approx^{k-1} v'$, and vice-versa.

Bisilarity is the base for 1-index. It guarantees that for every node in the extent of one node, not only themselves have the same label, but their parents also have the same label.Naturally,this constraint would go up recursively until the root. So at last, the incoming path starting from the root would be the same for all the nodes in one equivalent class or a node set. This is a very precise index. The equivalence relationship in A(k)-index and D(k)-index is built according to the k-bisimilarity.

Locally K-bisimilarity also needs that the nodes in one equivalence class have the same label, but when reaching a larger scope, the nodes may not have the same incoming path according to the parameter k. In other words, all the incoming paths within length k are the same for the nodes in one class or node set in A(k)-index. Compared with 1-index, A(k)-index would not be that precise, but still would be efficient enough when most of the query pathes are less than k. In this case, most results would be produced without another validation process. The parameter k determines the precise degree and size of the index. A suitable k should be point of the tradeoff. In A(k)-index, k would usually be a fixed value, while in practical case, the XPath of the query show some variability. That is to say, not all the structure in the data has the same equivalent significance. Depending on the query load,some types of the nodes may be frequently queried by long pathes, while some other nodes may always be accessed by short pathes.

1-index and A(k)-index can not adaptively adjust the index according to the different structure complexity of the equivalence class required by the query load. D(k)-index is a dynamic structure and could tune effectively for specific query load to achieve reduced index size and improved performance. Without a fixed k, D(k)-index would try to use different and most efficient local similarity for equivalence class according to the query load.

A(k)-index could be considered as 1-index when the parameter k is big enough, while D(k)-index would be A(k)-index if all the local nodes adopt the same k-bisimilarity and 1-index if the k is very large. Based on the above analysis, D(k)-index could be called the generalization of 1-index and k-index.So these indexes are regarded as one family based on the similarity and differ with each other according to the similarity degree. This could be shown in detail in the **Fig** 2. The node with label E in D(k)-index have 3-bisimilarity while other nodes in that index have 1-bisimilarity. We can easily find that in that figure, A(4)-index is the same as the 1-index showing that the interrelationship of index family members.

Compared with index family, the strong dataguide though keeping all the paths of the original data, just simply combines all the nodes with the same incoming label. If one node has more than one incoming label, then the node would be redundantly included in several node sets like in **Fig** 1 which would not happen in the index family. To some certain degree, strong dataguide is a little like A(1)-index just having one step similarity. But the dataguide is one structure considered from the perspective of the labels while the index family is from the nodes. For every label in the strong dataguide, the structure combine all the nodes that could be reached from this label. In the index family, for every node, all the other nodes that having the same labels with this node would be combined together.

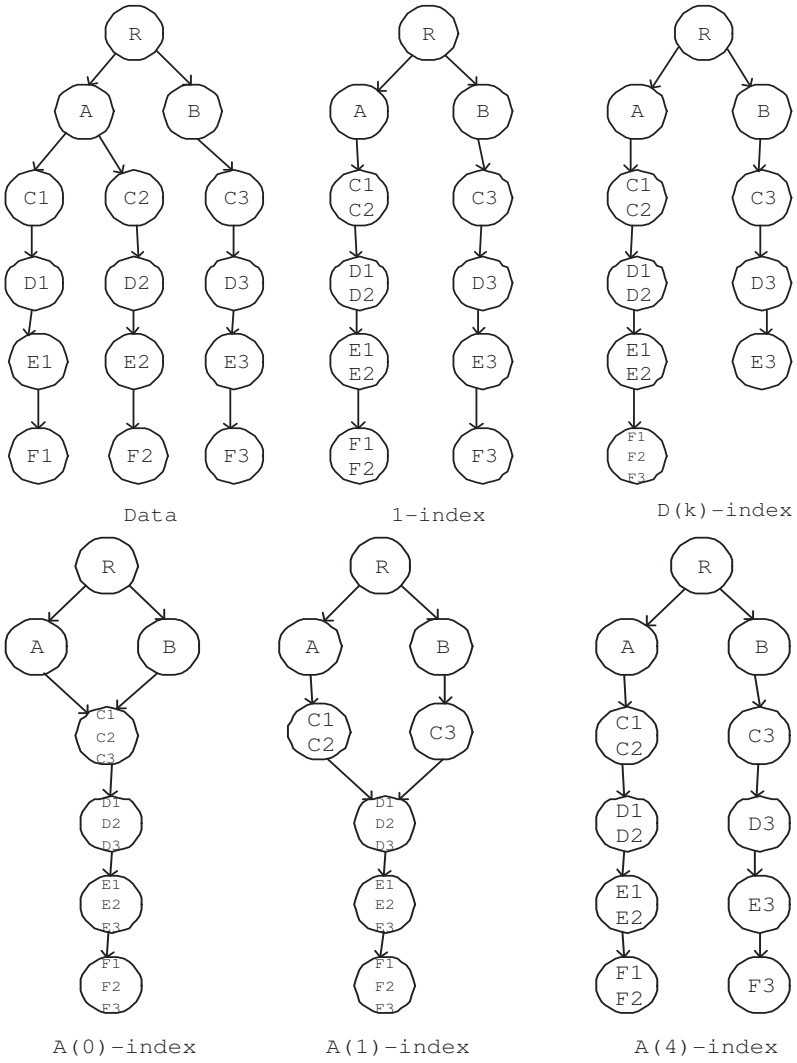


Figure 2: Index family

3 XML Update Issues

XML documents on the Web are subjected to frequent updates. When updates are performed on the XML document, there are three aspects which are needed to take care.

- Labels : the labels which are assigned to each element in the XML document will also change.
- Orders : a node which is the second child of its parent becomes the third child because of the insertion of another node.
- Structural index : the insertion of an edge may cause the structural index to change a lot.

As aforementioned, there are two types of labels, i.e. range-based labels and prefix based labels. For the case of range-based labels, when we insert new nodes into the XML document, there will be no available space for the new inserted nodes. The current labels for the nodes are needed to be changed because of the new inserted node. As an example, we use the XML tree in **Fig** 3 . The original XML

tree contains node 1, 2, 3, and 4 and the labels for these nodes are labelled beside the node. When we insert node 5 as a child of node 2 into the tree, nodes 1, 2, 3, and 4 are all needed to be relabelled.

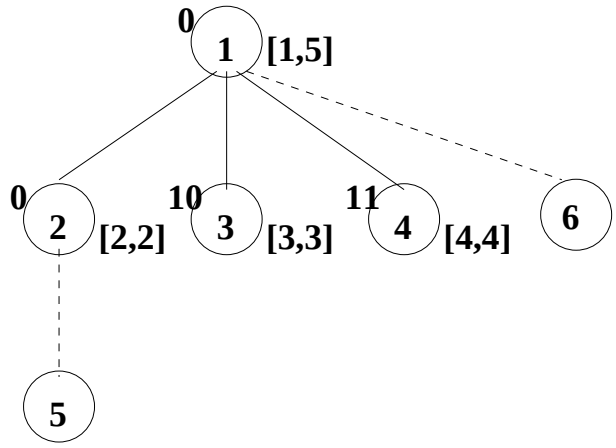


Figure 3: Range-based labels for an XML tree

For the case of prefix-based labels, when we insert new nodes into the XML document, there will be no valid prefix-free labels for the new inserted node. We can still use the XML tree in **Fig** 3 to show an example. The number on the up-left corner of each node is the label assigned to the node. When we want to insert node 6 as a child of node 1 into the tree, we will find that there is no prefix-free string can be assigned to the new inserted node. At least, we have to relabel node 4 for the insertion of node 6. However, re-labeling the XML tree is expensive, especially for the frequent updates.

The query on XML document can be divided into *ordered* query and *unordered* query. For the query /publication/book[pos()=2], if we insert a book element before the second book child of publication, the query result will change. The problem for *ordered* query is how to maintenance the order information of each node with low cost during update.

As discussed in the above section, structural index is built on the XML document for query evaluation or result size estimation. However, when the XML documents is updated, the structural index may not be consistent with the underlying data. To guarantee the quality, a naive approach is to rebuild the structural index upon the updates. However, rebuilding the structural index is expensive if the updates are frequent. Therefore, we need an incremental maintenance of structural index during updates, which provides both the *efficacy* and *efficiency*.

In the following parts of this section, we first discuss some mechanisms for labeling dynamic XML document, then we present an algorithm which maintains the order information of nodes, and finally we discuss the approaches to incremental maintain the structural index, i.e. Strong DataGuide, 1-index and A(k)-index.

3.1 Label Maintenance of XML Updates

As aforementioned, the re-labeling is expensive, therefore we need to design schemes which will not cause re-labeling a large part of the XML tree during updates. In this section, we discuss and compare some XML labeling approaches which are suitable for the documents that are subjected to updates.

3.1.1 Extending range-based scheme

One simple approach is to leave some "gaps" for range based labels. For example, initially we label the node 1 in **Fig** 3 as (1,8), node 2 as (2,5), node 3 as (6,6), node 4 as (7,7), then the insertion of node 5 will not cause re-labeling. However, it may not be easy to determine how many "gaps" we should

leave for each node and if one part of the document is frequently updated, the available numbers may be run out and the re-labeling is unavoidable.

Xing and Tseng improve the above approach by handling the overflow with a prefix based scheme [19]. In their scheme, the label is a integer pair, which specifies the range allocated to the node, prefixed with a sequence of integers, denoted as $p_1p_2 \dots p_n(s, e)$. The range allocated to a node is also left some "gaps". When we insert a new node into the XML tree, there are two cases :

- If the range is available, then the node is inserted in the same way as the range based scheme.
- If the range is not available, suppose the label for the parent of the node is $p_1p_2 \dots p_n(s, e)$, then the node first inherits $p_1p_2 \dots p_n s$ from its parent as the prefix, and then we assign the node a range (s', e') which is used to create a new pseudo-root of a new subtree, making insertion to the newly created node the same way as insertion to the previous nodes.

Although combing prefix based scheme and range based scheme can avoid re-labeling during updates, it may cause relatively longer length of labels compared with other schemes discussed in the following section.

3.1.2 Using floating point numbers as labels

Amagasa et al [5] proposed a novel labeling scheme, called Quartering-Regions Scheme (QRS). QRS also uses a pair of numbers consisting of the start and end points of the corresponding node in an XML tree like other labeling scheme. The main difference is that QRS uses floating point numbers instead of the integer numbers. When a subelement is inserted into an element, the range of the element is evenly divided into four parts. The end of the first part is the start of the label for the new inserted element, and the end of the third part is the end of the label of the new inserted element. For example, suppose an element is to be inserted between two sibling elements labelled (00, 01) and (10, 11), then the range between 01 and 10 is divided into four parts by the break points 01.01, 01.10, 01.11 and the range (01.01, 01.11) is assigned to the new inserted element.

Theoretically, we can always insert a number between any two floating point numbers. However, in practice the representation of a floating number is constrained by the number of bits in the mantissa. When the limitation for mantissa is reached, the re-labeling is necessary for the following insertion. As we know, every insertion consumes 2bits, then about 20 times updates at a particular point will consume the whole bits for the mantissa. Therefore, this approach is not strong enough for the insertion of elements which is composed of a lot of subelements.

3.1.3 Designing persistent prefix free scheme

For the prefix based scheme introduced in the background section, when we insert new elements into the tree, we can not find a prefix free string to label them. However, we can purposely design a prefix based scheme which always allows us to find a prefix free label for the new inserted element at the cost of longer labels.

Tatarinov et al [17] introduced a prefix labeling scheme which labels the n_{th} child of a node with integer n. The label for each element also inherits the label from its parent as the prefix. **Fig** 4 gives an example for this labeling scheme, the number beside the node is the labels for the node.

However, this scheme only supports the fan-out which is less than 10. When the fan-out of some nodes are larger than 10, different elements may have the same labels. The label for the node which is the 11th child of the node which is the second child of the root node is "1211", and the first child of the node which is the 21th child of the root node is also "1211". To handle this problem, this approach takes advantage of some delimiter. Suppose we use comma as the delimiter here, then the label for the 11th child of the second child of the root becomes "1, 2, 11", and the label for the first child of the 21th child of the root becomes "1, 21, 1". However, the delimiter must be stored with the labels together, which incurs significant cost.

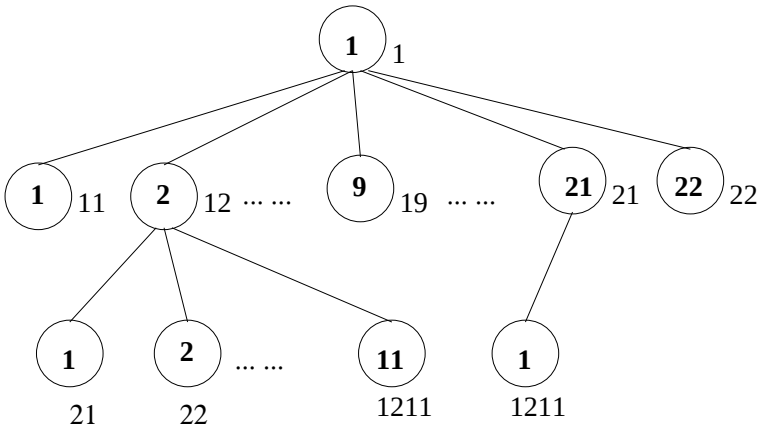


Figure 4: An example for prefix labeling scheme

Cohen et al proposed two prefix based labeling schemes [7], in which the labels need not be changed when the document is updated. The schemes proposed by Cohen et al avoid the storage cost for delimiter, since there is no label collision in the scheme.

The first prefix labeling scheme, which we call PREFIX1, is as follows. We label the root node with the empty string. The first child of the root is labelled with "0", the second child of the root is labelled with "10", the third child of the root is labelled with "110", the fourth child is "1110", and the ith child is "$1^{i-1}0$", etc. In general, suppose the label for node u is L(u), then the label for the first child of u is L(u)·0; the label for the second child of u is L(u)·10, and the label for the ith child is L(u)·$1^{i-1}0$". For a node v, we can compute L(v) using the equation 1

$$L(v) = L(u) \cdot 1^{i-1}0, \quad v \ is \ the \ i^{th} \ child \ of \ u \tag{1}$$

In another prefix based labeling scheme given by Cohen et al in [7] (denoted as PREFIX2), the label of a node v consists of two parts as other prefix based schemes. The first part is inherited from the parent of v, and the second part is a string which is assigned in the way shown as follows. Suppose v is the i^{th} child of its parent, the string assigned to v is denoted as s(i) and the string assigned to the $i-1^{th}$ child is denoted as s(i-1). Then if s(i-1)+1 is not all ones, we assign s(i-1)+1 to s(i), otherwise we double the length of s(i-1)+1 by concatenating a sequence of zeros and assign the result string to s(i). **Fig** 5 shows an example for this prefix based scheme. For a node v which is the i^{th} child of its parent, we can compute the string assigned to v with equation 2.

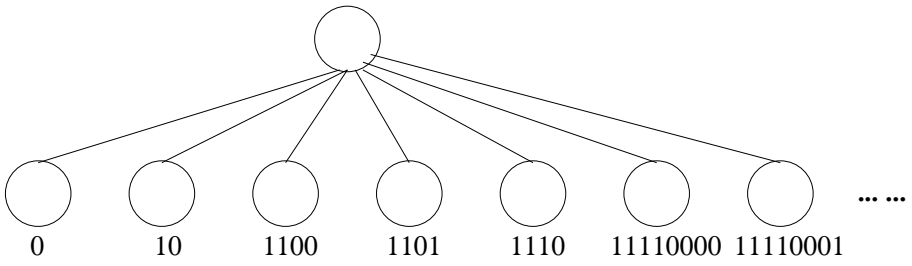


Figure 5: An example for prefix based labeling scheme

$$s[i] = \begin{cases} s[i-1]+1 & \text{if s[i-1]+1 is not all ones} \\ 11...100...0 & \text{s[i-1]+1 is all ones, 0s=1s} \end{cases} \tag{2}$$

For the two prefix based persistent labeling schemes, i.e. PREFIX1 and PREFIX2, proposed in [7], the length of labels in PREFIX1 increases linearly with the fan-out of a node while the length of labels in PREFIX2 increases sub-linearly with the fan-out of a node. The heuristics of PREFIX2 scheme is that the more children that a node already has, the more likely for it to get additional children. Therefore, rather than allocating the new inserted child the shortest possible available prefix-free string, we assign a longer string instead. However, it is likely to be paid off as the forthcoming nodes can obtain shorter labels. In PREFIX1, the length of assigned prefix free string grows by one bit for each new inserted child, while in PREFIX2 the label length may grow several bits at once, and then it can stay the same for more future inserted nodes. For the XML document in real life, we observe that the trees are usually balanced with low depth and relatively high fan-out, therefore, PREFIX2 should be better than PREFIX1.

Cohen et al in [7] also suggested some additional information which is associated with the inserted node and can reduce the bound for the label length. The additional information is modelled as clues, which can be derived from DTD or the statistics of the similar document obeyed the same DTD. There are two kinds of the clues, i.e. *subtree clues* and *sibling clues*. The subtree clue is the estimation of the number of future descendants of the inserted node, and the sibling clue is the estimation of the number of future descendant together with the number of descendants of future siblings of the inserted node. The additional information will restrict the number of possible final trees, therefore we can design shorter labels using the additional information.

Table 1 shows the bounds of labels with different labeling schemes. We can see that for the static XML document, the bounds for the label length is $\Theta(\log n)$. For dynamic XML document, if we do not have any additional information, the bounds for the label length is $\Theta(n)$; if we have subtree clues, we can achieve the bounds of $\Theta(\log^2 n)$; if we get sibling clues further, we can reduce the bounds to $\Theta(\log n)$, which is the same with the static XML case.

problem	static	dynamic		
		no clues	subtree clues	sibling clues
bounds	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log^2 n)$	$\Theta(\log n)$

Table 1: The bounds comparison of different labeling schemes

The above table indicates that any persistent labeling scheme, which can be called *immutable* labeling scheme, requires $\Omega(n)$ bits per label, where n is the number of nodes in XML trees. Since XML trees always contain a large number of nodes, $\Omega(n)$ bits per label is relatively long comparing with the static document. The long labels incur high storage overhead, and they also affect the query processing efficiency, since processing long labels is expensive than processing short labels. Although Cohen et al argue that the use of sibling clues can reduce the bound for label size to $\Theta(\log n)$, it is not clear how to estimate the sibling clues exactly, and the wrong estimation will also cause longer labels.

3.1.4 Utilizing the properties of prime numbers

Wu et al [18] proposed a prime number labeling scheme which uses integers and several properties of integers such as prime factors, relatively prime, congruence equations etc. The prime number labeling scheme is suitable to label dynamic XML trees.

The basic[1] prime number labeling scheme uses a set of primes to label all the nodes of the XML data tree. Each label of a node is a product of two values, the first value is called the self-label and is equal to a unique prime number. The second value is called the parent-label and is equal to the label of its parent.

Given any labeling scheme, it is desirable to have the deterministic property, i.e. given the labels of any two nodes, the structural relationship between them can be easily determined. In this scheme, a node is an ancestor of another only if its label exactly divides the label of the other. To illustrate

[1]The optimized version relaxes this requirement for leaf nodes

the labeling scheme let us look at some interesting properties of integers and an example applying the labeling to an XML document

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.*

Example:*Suppose A = 55 (5*11), B = 265 (5*53), 11 not prime factor of B, hence A does not divide B.*

Property 2 [**TopDownMod**]: In a top-down 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(y) mod label(x) = 0.

Example: *Lets say x is a node with label 4895(1*5*11* 89) and y is another node with label 104522935 (1*5*11 * 89 * 131 * 163). Clearly, label(y) mod label(x) = 0.*

A labeling example: Consider a XML document D, we can see in **Fig** 6 the above properties are easily satisfied. For example, the node labeled 995 is not an ancestor of the node labeled 302395 where as the node labeled 985 is.

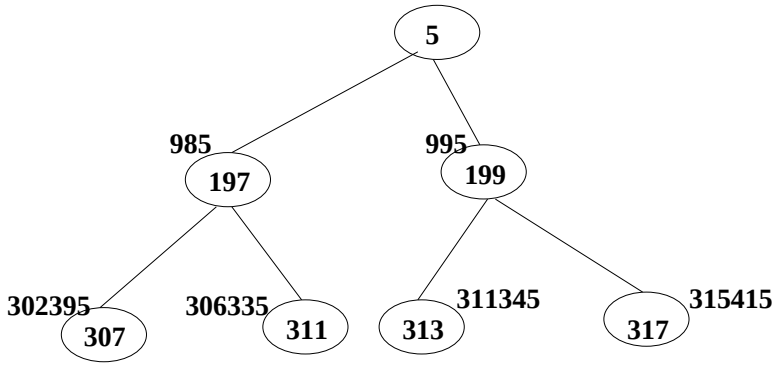


Figure 6: Example of the prime number labeling scheme

We can easily see that 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. As in any labeling scheme the size of the labels is an important factor in determining its feasibility. This scheme is largely unaffected by XML trees with large fan out, and is quite compact for shallow XML trees.

We use D, F, and N to denote the maximal depth, maximal fan-out and number of nodes of an XML tree respectively. We find the PREFIX1 has a maximum label size of

$$\text{PREFIX1} : L_{max} = D * F \tag{3}$$

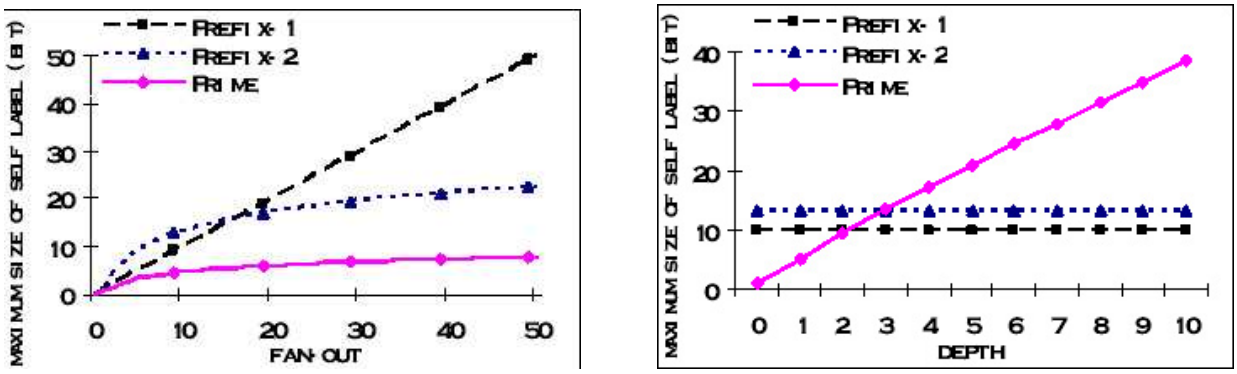
The PREFIX2 scheme is scales much better and has a maximum label size of

$$\text{PREFIX2} : L_{max} = D * 4logF \tag{4}$$

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. 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. The maximum number of bits required by the node labels at each level is given by Dlog(θ_N). 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)). Thus, the prime number labeling scheme has a maximum label size of

$$Prime : L_{max} = Dlog((\sum_{i=0}^{D} F^i)log(\sum_{i=0}^{D} F^i)) \tag{5}$$

The above equations enable us to compare the three labeling schemes that is characterized by the growth of the L_{max}. We find 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 nodes self label. The latter is influenced by the fan-out of the nodes parent. Using the formulae given above, the three schemes have been studied in the paper and the results obtained have been summarized in **Fig** 7. The interval based scheme that is used for static document labeling is the smallest with label size equal to 2log(N), however, considering dynamic documents, we observe that with limited depth the prime number labeling has lower label sizes compared to the other two.



(a) **(b)**

Figure 7: Comparison of the three schemes for varying depth and fan-out

The basic prime number labeling scheme has a few short comings, as the prime numbers cannot be reused, larger and larger primes need to be used for each node. In addition to this, the labels of the leaf nodes tend to be very large because of the presence of parent-labels. To reduce the effect of the above mentioned short comings, the authors proposed three important optimization steps

1. Reserve the smaller primes for the nodes near the root.
2. Relax the requirement for all the leaf nodes to be primes.
3. Collapse common paths reducing the number of nodes to be labeled.

Optimization 1 ensures that the size of the labels of descendant nodes increase slowly. Optimization 2 uses the property that two is the only even prime. Two is not used in the labeling of non-leaf nodes resulting in all the non-leaf nodes having odd labels. Two and its powers are used as self-labels for the leaf nodes. This results in property 2 being modified as shown below.

Property 3 [OptimizedMod]: *In the optimized topdown 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.*

Optimization step three is common to other labeling schemes too. In this step, we can fold many common paths into a single path, however, this is cannot be applied where sibling ordering is important. Let us now look at an example that applies these optimization principles.

As shown in **Fig** 8 we find that the usage of lower primes towards the root and the use of powers of two for the leaf nodes results in smaller label sizes as compared to the example of **Fig** 6. The optimization with collapsing of common paths has not been applied.

Now let us look at the **Alg** 1 to label the XML tree using the optimized prime number labeling scheme.

The *PrimeLabel* algorithm incorporates the two optimization techniques described in the previous section, using the function *getReservedPrime()* we obtain small prime numbers for the top level nodes

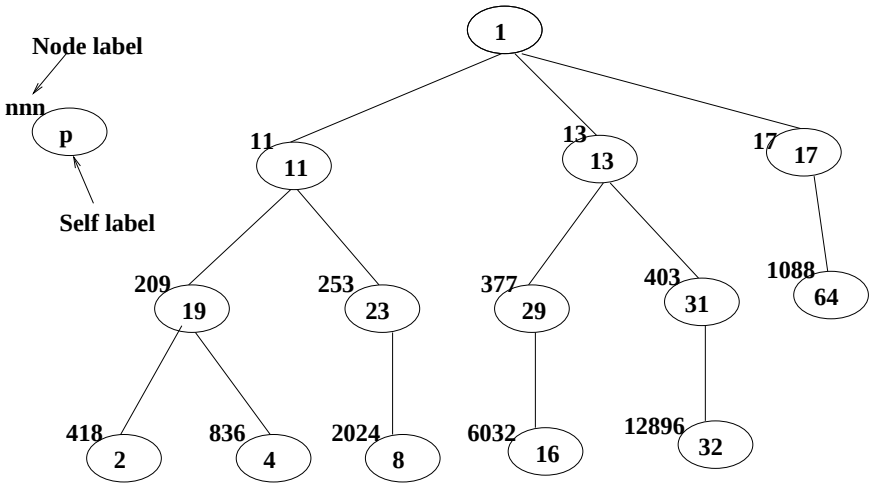


Figure 8: Example of the optimized prime number labeling scheme

of the XML tree and the function *getPower2(n)* returns the number 2^n to label the n^{th} leaf node. For all other nodes, the function *getPrime()* returns the next smallest prime number to be used as its self-label.

3.1.5 Maintenance with additional data structure

As proved in [7], any *immutable* labeling scheme requires $\Omega(n)$ bits per label, which is relatively long. Silberstein et al [16] proposed to associate dynamic labels with *immutable label IDs* (LID) to each node. The *immutable label IDs* which keep the same during updates is duplicated freely in the database, and the dynamic labels are stored in only one place so that update is efficient. This approach makes use of a weight-balanced B-tree (W-BOX) to maintain the dynamic labels, and utilizes a heap file called *immutable label ID file* (LIDF) to reference the *immutable label IDs* to their dynamic labels. **Fig** 9 shows the structure of the LIDF and W-BOX.

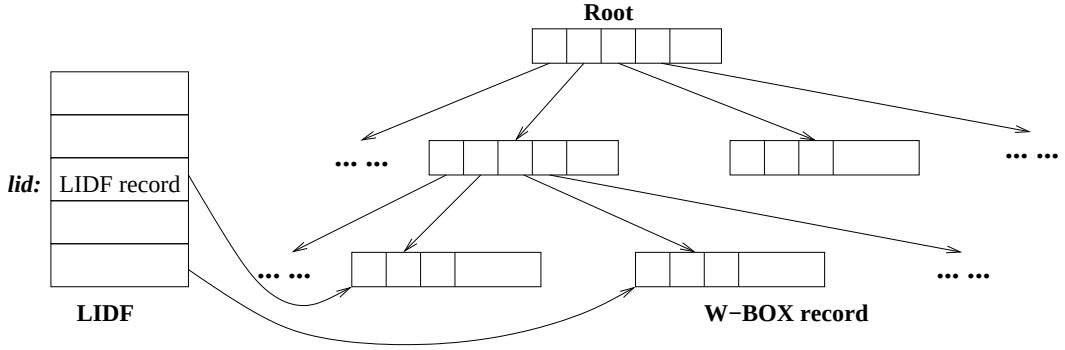


Figure 9: The structure for LIDF and W-BOX

The W-BOX is a weight-balanced B-tree of W-BOX records with label values as search keys. A W-BOX internal node contains a list of child pointers separated by search key values; each child pointer points to the child of this internal node. A W-BOX leaf node contains a list of W-BOX records, each of which stores the value and LID of a label.

The LIDF is a heap file containing a list of LIDF records. Each element e has two LIDF records, one is for the start label and the other is for the end label. Their record numbers serve as e's LIDs.

Algorithm 1 PrimeLabel

```
1: Input: XML document
2: Output: Label for each node
3: begin
4: for each node n in the XML document do
5:    n.childNum = 0;
6:    if n is the root node then
7:       n.label = 1;
8:    else
9:       parent = n.parent;
10:   if n is a non-leaf node then
11:      if this node is a top level node then
12:         n.selfLabel= getReservedPrime();
13:      else
14:         n.selfLabel = getPrime();
15:   else
16:      parent.childNum++;
17:      n.selfLabel=getPower2(parent.childNum);
18:   output (parent.label*n.selfLabel);
19: end
```

The LIDF records store pointers to the block containing corresponding BOX records. For a given element with its LID, we can use LIDF to direct access its LIDF record, and use the pointer of LIDF to find its label value in the BOX.

When a new XML element e is inserted into the document, we allocate two new records in LIDF, i.e. one of e's start label and the other for e's end label. The LIDs for an element will never change once assigned, hence they can be freely used in indexes or as element IDs. Then the LIDs should be inserted in the W-BOX, which is similar with inserting a node in a B-tree. The insertion may cause some nodes in the B-tree to split, and even to reallocate the range of some nodes in the B-tree. The split and range reallocation will cause some records in leaf nodes to be re-labeled.

The labeling scheme proposed by Siberstein et al is a trade-off between the update and lookup costs. When we want to retrieve the label value of a node, we have to use the LIDs of the label to find the LIDF records, and use the pointers to retrieve the label values in BOX, which introduces an extra cost for read-heavy queries. However, for updates, the LIDs will keep the same, we only need to do the insertion to a B-tree. It can be proved that given a LIDF record, the cost of retrieving the label from a W-BOX is one I/O, and the amortized cost to insert and delete a label in a W-BOX is O($log_B N$) and O(1), respectively. However, the bound of label length in this scheme achieve O($logN$), which is the same with labeling static XML document.

3.2 Order Maintenance of XML Updates

In the earlier section on labeling schemes we looked at how the prime number labeling scheme can be applied to an XML document. In this section we shall study how the prime number labeling scheme supports ordered queries over a ordered tree and how updates such as inserts and deletes are supported. Let us first look at labeling ordered trees.

Elements in a XML document are intrinsically ordered. There are several types of queries one can execute such that only those results that satisfy ordered constraints are returned. For efficient ordered query processing, the labeling scheme could extend itself to include order information. The earlier mentioned schemes [10, 7] have lacked in this regard. In the prime number labeling scheme an global order of all the nodes of the XML document is maintained.

The prime numbers used as self labels for the nodes are unique in the XML tree, we can use the

fact that all the self-labels are relatively prime to each other to establish a global order among all the nodes. This is achieved using the property of the chinese remainder theorem.

Theorem 1 [Chinese Remainder Theorem] : Let M = [m_1, m_2, m_k] and N = [n_1, n_2, , n_k] be two lists of integers. If the GCD (m_1, m_2, m_k) = 1, then the simultaneous congruence SC (M, N) = x satisfies x mod m_1 = n_1, x mod m_2 = n_2 ...,x mod m_k = n_k. There exists exactly one solution x to this set of congruence equations and its value lies between 0 and C, where C = $\prod_{i=1}^{i=k}(M_i)$. The paper [18] suggests the usage of Euler's quotient function

$$X = (\sum_{i=1}^{k}(C/M_i) * n_i * \phi(m_i))modC \tag{6}$$

where $\phi(m_i)$ is the number of integers relatively prime to m_i. From our observations we found that, this equation does not result in a correct solution. instead of equation 6 we suggest the following equation to be used.

$$SC = \begin{cases} n_i * b_i * (C/m_i) + .. + n_r * b_r * (C/m_r), \text{given C} = m_1 * m_2 * .. * m_r \\ \text{where } b_i \text{ is determined from } b_i*(C/m_i) = 1 \text{ mod } m_i \end{cases} \tag{7}$$

To illustrate how the global ordering is computed, we present an example using **Fig** 10.

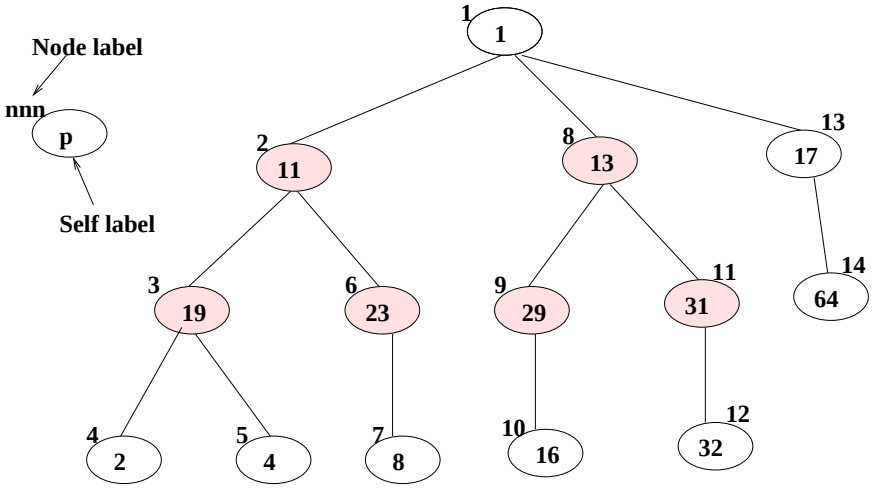


Figure 10: Example of determining ordering

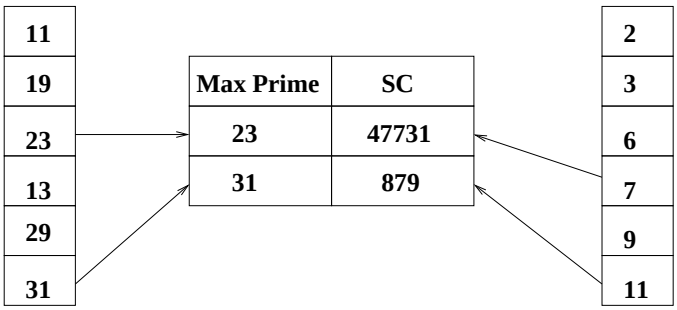


Figure 11: SC table determining ordering of XML tree in **Fig** 10

In **Fig** 10 the XML tree has been labeled using the prime number labeling scheme. The number within the node is the self-label and the number outside the label is the order. The root node is order

position 1. Using the set of self-labels as m_i, the global order value as n_i and the property of the chinese remainder theorem, we could arrive at a very large SC value, but one that can determine the entire order of the XML tree. Instead, we could also compute several SC values by dividing the set of nodes into several smaller sets, each of them specifying the n_i value corresponding to the global order. For example as shown in **Fig** 11, consider the set of dark shaded nodes, suppose we want to compute two SC values for these nodes, Let M_1 be the set of self label values (11, 19, 23, 13) and N_1 be (2, 3, 6, 8) representing the order. Let M_2 be the set of self label values (29, 31) and N_2 be (9, 11). Thus we obtain two SC values $SC_1 = 47731$ and $SC_2 = 879$. These SC values are maintained in a table.

To see how this works, consider the node with self-label 23, 47731 mod 23 gives you 6, which is the order of the node. Thus the complexity to determine the order given a label $\Theta(log(l))$, where l is the number of SC values.

One aspect of this ordering scheme that has not been talked about in the original paper is the ordering of the leaf nodes when powers of 2 are used as its self labels. This voids the assumption that all the M_i values are relatively prime, hence, they cannot be used to compute the SC values . Our understanding is that, the SC table is used only for the ordering of the non-leaf nodes and separate ordering information is captured for the leaf nodes.

The prime number labeling scheme is suited for dynamic XML documents, it does not require re-labeling of nodes upon updates and upon deletion of nodes nothing needs to be done. When a node is deleted, the previously computed global order is still valid as the sequence of ascending order numbers is sufficient to determine the actual order. Insertion of nodes to the the document entails some effort. The SC value of the group of primes to which it is added needs to be recomputed, and well as all those SC blocks that follow it. Similarly, the global order values of all the nodes that follow it need to be incremented by one as well. This overhead is less expensive compared to the option of re-labeling the affected nodes. To illustrate this, we insert a node with self label '37' to the XML tree in **Fig** 10 as the second child of node with self-label '13', this results in the new node having a global order number of 11. The order numbers that follow this node are incremented by one and the SC tables updated to reflect the change. The resulting XML tree and the SC table are shown in **Fig** 12.

3.3 Structural Index Maintenance of XML Updates

We have already discussed several structural indexes of XML document. Unlike schema, the structure index built on the XML data is not prescriptive and pre-defined. It is just a summary of the structure in the data graph and should change with the update of the data, so that the subsequent queries have a view of the summarized data that is consistent with the updated document. Based on the content, the update would fall into two kinds:

- Value update. The value of the elements in the data may change.
- Structure update. Some nodes or labels may change which would affect the structure information of the data.

The value update would not be the focus of the following part, the index is the structural summary of the XML graph and the value change of a node would not affect the structure , thus still keeping the index stable. While the structure update would need more attention. the impact of a slight change could be arbitrarily large in terms of the change in the size of the index. According to the ways of the updating, the update has the following cases:

- Add(or delete) an edge.
- Add(or delete) a subgraph.

Though, the process of a subgraph could be decomposed into processing edges step by step, a few more efficient ways would be proposed in the later section. The strategies for the update generally include the following:

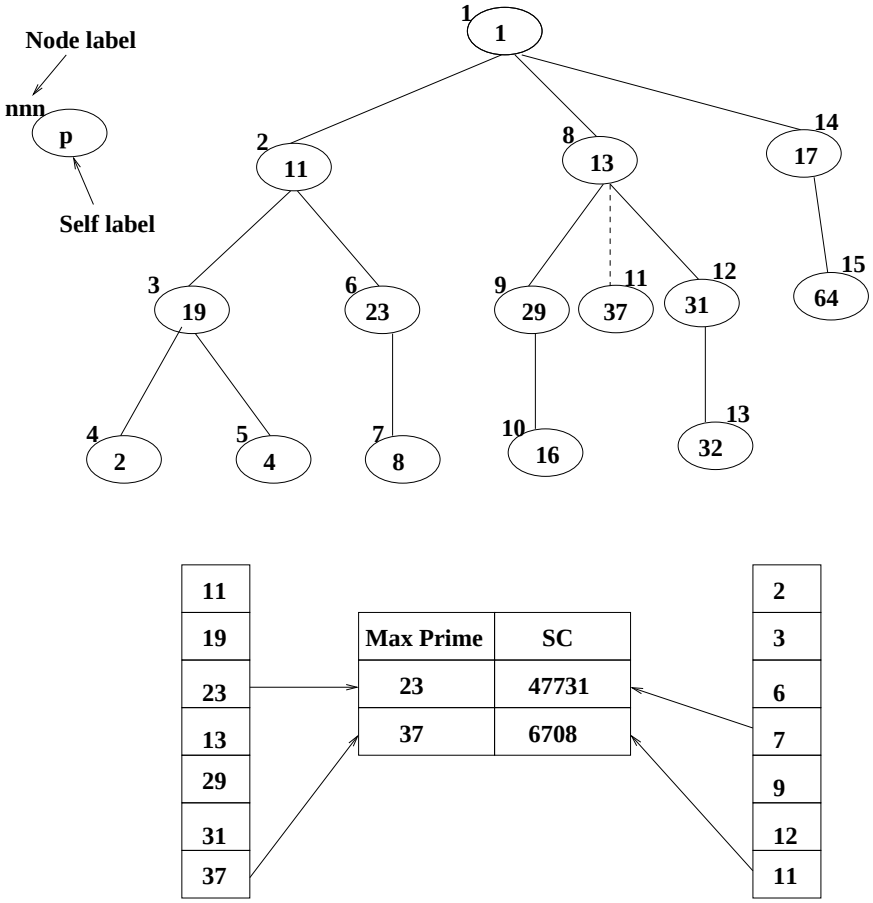


Figure 12: Updated XML tree and SC table for new node with self-label '37'

- Rebuild the index. When some nodes or labels in the data are changed, the whole index should be rebuilt.

- Incrementally maintain the index. When some nodes of labels in the data are changed, only related parts in the index would be updated.

The simple reconstruction strategy, though always keeping the index consistent with the updated document, would be obviously expensive from the computation perspective even when there are not many updating cases. Compared with the rebuild strategy, the incremental maintenance is much more efficient and would be mainly talked about in the following. To precisely evaluate the update algorithms, the most important measure are *efficacy* and *efficiency*:

- *efficacy* means preserving the quality of the structural summary. For the same underling data, there are lots of correct structure summaries with different sizes. The algorithm should make sure that the updated index is not expanded unnecessarily.

- *efficiency* means the algorithm itself should be efficient. To some degree, it means that the updating process should not be very expensive.

The reconstruction strategy provides perfect efficacy, but severely lacks efficiency. A good update algorithm should be able to optimally find the delicate balance between efficacy and efficiency.

3.3.1 Update Maintenance for Strong DataGuide

DataGuide is a structure summary, while the data could be changing from time to time. To keep the consistency, the DataGuide should be updated accordingly. The DataGuides updating algorithm adopts incremental maintenance strategy. It builds two important hash tables objectHash and targetHash to keep necessary information for the update. ObjectHash is used to map a source object o to all the DataGuide objects that correspond to target sets containing o. So whenever an edge $u.l.v$ from object u to object v via l is deleted from or added to the source, the object u would be treated as the *update point*. Through the objectHash, we can get the according object in the DataGuide which is called "sub-DataGuide" describing the potential changing structure of the object in the source target set. The tagetHash is used to avoid excessive re-computation; if we encounter a target set that already has a corresponding DataGuide object, the recursion would be halted. We explain the updating algorithm by using an edge insertion as an example.

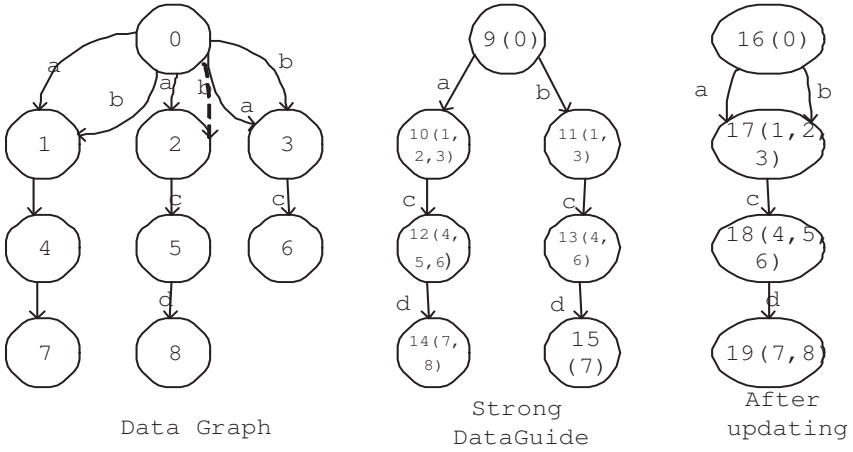


Figure 13: Update for Strong DataGuide

Fig 13 (a) without the dashed b edge between node 0 and node 2, is the original data. Edge b is going to be inserted. **Fig** 13 (b) is the strong dataguide for this source. After this insertion, Object 0 is the only update point(where some update would be done). While object 9 is the only target set in the strong dataguide that object o is part of according to the objectHash. Then we would check all the children of the update point, label by label. First, via label a, we reach a target set 1,2,3. From the targetHash, we find that object 10 corresponds to this set and there has been an edge with label a between object 9 and object 10. So no additional work would be done for label a, Then check the children via label b, the target set is also 1,2,3. Hence, we just need delete the original label b and add a new edge between object 9 and object 10 with label b. Then we get the final strong dataguide after inserting an edge. when deleting an edge, the according target set would check if all the nodes in this set still have the deleted label, if nodes would not have the label, they should be separated into another target set while inheriting all the labels and descendants of the original target set.

The computational cost greatly depends on the data structure. If the update point is near the root, then lots of the edges or nodes in the strong dataguide would be changed, while if the update point is at the leaf of the data, very few computation would be needed.

3.3.2 Update Maintenance for Index Family

Structural index of XML data is not a schema predefined and just a structure summary from the original data. While the data could be changed gradually, the index should be updated accordingly to keep the consistence. The basic idea of updating index is as following: when an edge is added or

deleted in the original data, first the according edge in the index would be processed and then check if the index still keeps the property, such as the k-bisimilarity. If it does not , the node sets would be split recursively until all keep the property again. At the same time, some optimization of the index would be done. How to make the participation? It is based on the notion of *stability*:

An node I is stable with respect to J if either $I \subseteq \text{Succ}(J)$ or $I \cap \text{Succ}(J) = \phi$. For a data graph G an index $\Phi(G)$ is stable w.r.t index $\Phi'(G)$ if for any node $I \in \Phi(G), I' \in \Phi'(G)$ I is stable w.r.t I'.

According to the definition, if A and B are in an index graph, A is stable with respect to B. There would be no edge from B to A or every data graph node in the extent of A has a parent in the extent of B.*Stable* is the crucial constraint for keeping the index property. In the process of updating, When finding one node set without keeping the local bisimilarity, split is needed. While the split in most cases would affect the property of its descendent node sets recursively. To make nodes stable with its ancestors is the restriction for the participation. How to make two node I and J stable? we just need to split node I into $I \cap \text{Succ}(J)$ and I-$I \cap \text{Succ}(J)$.

The fundamental steps of the updating algorithm provide in the paper includes two phases as following:

- split: Iteratively split the nodes to make the index correct form the aspect of bisimilarity based on stable.

- merge: Combine all the nearby nodes to make the index size to be minimum without violating the constraint.

Both the split and merge phases are processed in local manner but could achieve the global property. Split is triggered by the local bisimilarity, while the split nodes would cause its descendants not stable with them.So the split would be recursively done to make the index correct.After the split, some new nodes are produced. Sometimes some nodes could be merged according to the bisimilarity. If no nodes could be merged in local area, then no other nodes could be merged for the index to make node stable with each other. Therefore, this makes sure that we could always get the minimum index by the merge phase. The merge phase is the key difference between the update algorithm in [20] and the propagate algorithm in [14, 11]. For the propagate algorithm, it just has the split phase, though always keeping the correctness, usually losing some concern on the size of the index, one of the most important index properties. Because the purpose of the building index is to get the query result on data of a much small size. Size, to some degree, is the efficiency of the index. So keep the minimum size of the index should be worthy of the computation in the merge phase. In the performance comparison, there would be some more detailed analysis.

Update 1-index Based on the two phase algorithm, the update should not change the bisimilarity of the 1-index while try to maintain the minimality of the size. The following is an example for the updating 1-index with an edge insertion as shown in **Fig** 14.

In the data graph, the letter in the node is the label and the dash line is the edge that would be inserted. The second graph (b) shows the old 1-index of the data before the update in which each node set include all the extent nodes. After inserting the new edge, the split phase first checks if the bisimilarity keeps and find that C3 and C4 have different incoming labels class with the new edge. So the node set would be split into two parts: one include C4 and the other part includes the left nodes shown in (c). The split phase then triggers the stable check for the descendant of the split node set. The node set including D7 and D8 is not stable with respect to the two newly generated nodes, so it would be split into node D7 and node D8. After this step, all the nodes are stable with respect to each other and the split phase ends. Then the merge phase starts by checking the sibling nodes of newly generated node C4 if they have the same label and the same set of index parents. For C5 satisfy the conditions, it would be merged with node C4. We then iteratively consider the possible merge among the children of the newly generated nodes from the previous merge. D8 and D9 are merged again and we then reach the final result.

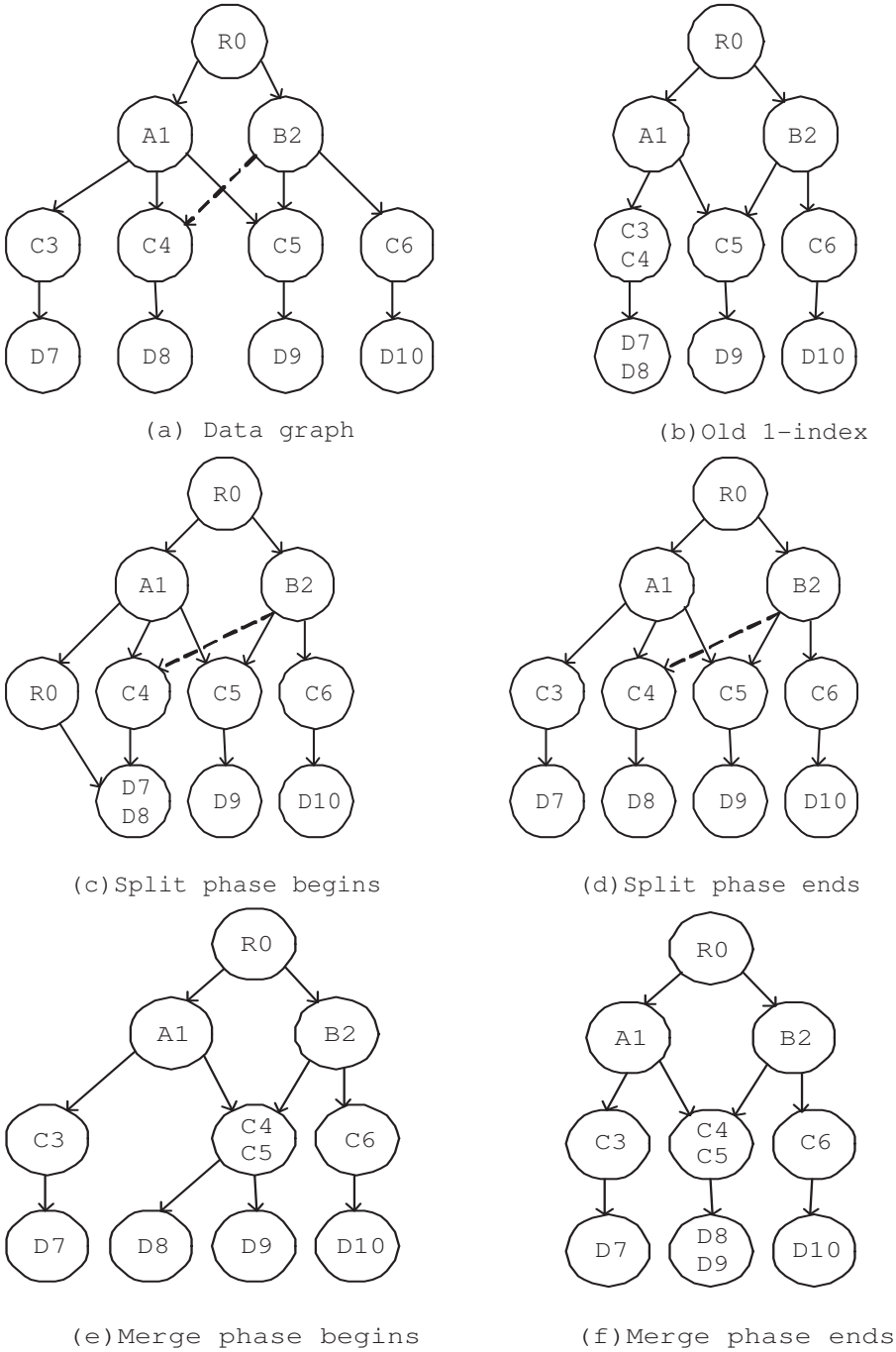


Figure 14: Update 1-index

Analysis of the update The analysis of the algorithm is based on two criterias : one is the effect of the algorithm and the other is the cost of the algorithm. As we have introduced before, efficacy is used to check whether the algorithm keeps the index in bounded size and efficiency is used to check the computational cost in the updating process. In the following, we would make some analysis from these two aspects:

- Efficacy: we can get the following theorem: For acyclic data graphs, the split/merge algorithm always maintains the minimum 1-index during edge insertion and deletions. For cyclic data graphs it always maintains a minimal 1-index.
- Efficiency: On the face of the two phases algorithm, it seems we spend some more time on the merge phase than the propagate algorithm, so definitely the cost would be a little more.But the merge phase brings us more benefit that is desirable for an index. With the little more cost, we can get a nice theoretical guarantee on quality of the resulted index. Always minimum size would reduce the cost for the subsequent query evaluation.In other words, the cost in the merge phase would be paid back in the later query, because queries would be more than update.

Update A(k)-index A(k)-index and 1-index are in the same index family and both are based on the simulation. The property of the index decides that the update algorithm for 1-index could be easily extended to the A(k)-index. The only different is that A(k)-index just needs to keep a much more approximate equivalence K-bisimilarity. While to facilitate the updating A(k)-index, the information of the A(k-1)-index is always required. Therefore, the basic idea in this algorithm is to maintain all the A(0), A(1), ... , A(k)-index together using split/merge update algorithm for the 1-index. In the algorithm, the core operation is the split/merge and the basic procedures are as following:

- Find the largest i such that A(i)-index would not be affect by the update. The A(i+1)-index is the update point.
- Make the initial split from A(k+1)-index to A(k)-index to create new nodes set with updated node. Then iteratively split would be done for each index as that for the 1-index.
- Merge phase nearly has no change. Still try to merge the newly generated node with any other node that with the same labels and the same index parents.

The main difference from 1-index is that we have to maintain all the A(i)-index for later reference. How could we efficiently maintain so many indexes without too much extra cost is a very important problem. Here, the algorithm use a refinement tree for the maintenance. The structure of the refinement tree is shown in **Fig** 15. In the refinement tree, only nodes in the A(k)-index would be stored, while for other A(i)-index just some relationship shown in the dash line is kept. From the containment relationship, all the A(i)-index could be recovered from the A(k)-index.

Analysis for updating A(k)-index The analysis is still based on the Efficacy and Efficiency:

- Efficacy: The core operation for updating the A(k)-index is still split/merge, while the natural property of the split/merge could guarantee the index to be correct while keeping minimum. As the theorem says:For any data graph G, the split/merge algorithm always maintain the minimum A(k)-index.
- Efficiency: To maintain all the A(i)-index makes the algorithm seems expensive, but with the help of the refinement tree introduced above, all the nodes would just be stored once in the A(k)-index from which all the other A(i)-index could be recovered based on the containment relationship, for every A(i+1)-index is always a refinement of the A(i)-index. So the extra storage overhead is very low.

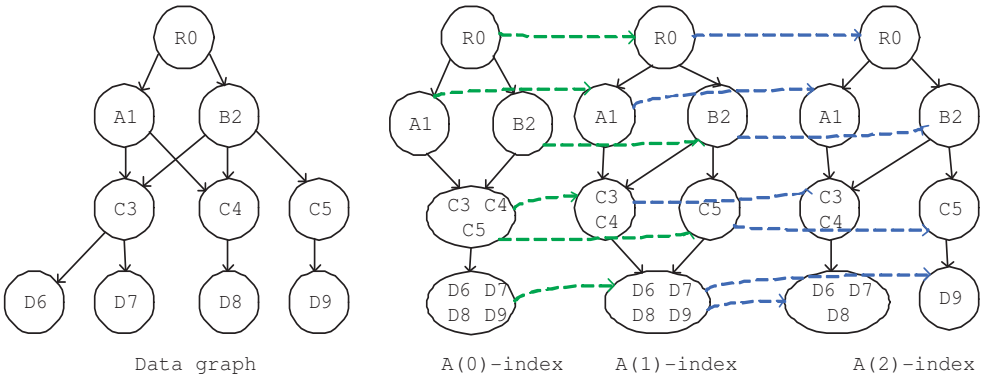


Figure 15: Refinement tree

Update D(k)-index D(k)-index is one index sensitive to the query load. Nodes with heavy queries would be given high local similarity to decrease the validation procedure. Therefore different nodes have different local bisimilarity. Unlike updating the A(k)-index, we could build the refinement tree to keep the information for all the A(i)-index for later reference. In the D(k)-index, we should first decrease the bisimilarity and then according to its index parents to enlarge the bisimilarity. With out the information of A(i)-index to update the A(i+1)-index for some nodes would be not efficient. Therefore we are considering to build a structure similar with the refinement tree but with the nodes having different local bisimilarity. But the query load may change gradually, accordingly this structure would also need update from time to time to keep the accuracy. To maintain this structure would also be expensive, another idea is find the tradeoff of the accuracy and cost. We would like to build a structure similar as the refinement tree but with a certain k that could benefit the update most according to the query load.

Subgraph update What we have talked about before is mainly about the edge processing including both adding an edge or deleting an edge. Essentially, subgraph processing could be decomposed into edge processing. For example, inserting an subgraph could be processed step by step via edge insertion. However sometimes, subgraph is the main unit for the update, for example, when you want to insert some small files into large data graph. In this kind of cases, processing edge by edge would be not efficient enough and some algorithm considering the subgraph as an entity is in demand. The basic idea of the algorithm is to process the subgraph in a batched manner as following: first build new index for the subgraph and then add all the edges between the new subgraph and the existing subgraph via the edge insertion algorithm mainly by the merge phase.

4 Performance Study

The observations in this section have been obtained from the experiments presented in the papers[2][18, 20]

4.1 Labeling Schemes Comparison

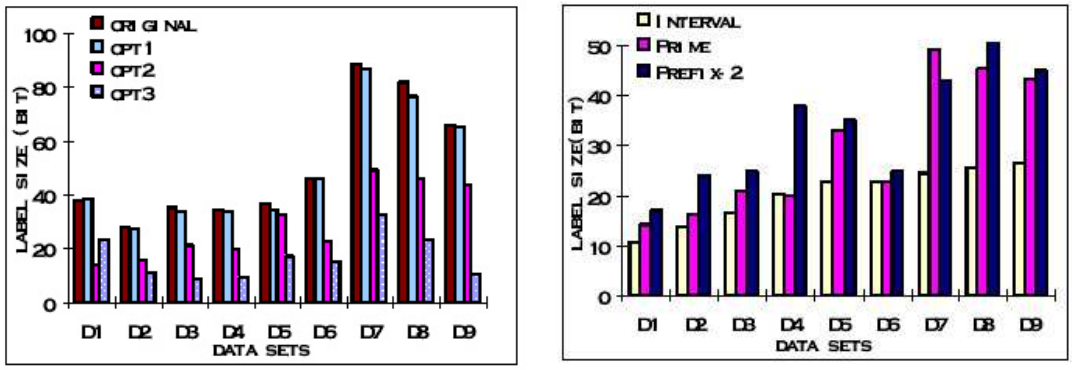
In the section on the prime number labeling scheme we briefly discussed the performance of the different labeling schemes w.r.t. the label sizes for varying depths and fan-outs of the XML document. In this section we will discuss a variety of comparison parameters such as query response times, number of nodes to re-label in case of both ordered and un-ordered queries etc.

[2]Figures illustrating performances of the various schemes have been borrowed from the original papers without the authors permission.

Let us consider a few data sets against which the performance of the prime number labeling scheme can be compared with that or the interval based and the prefix based schemes. These data-sets have different characteristics in terms of average fan-out and depth, the number of nodes etc.

Data-set	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	Shakespeare's Plays	6636
D9	Company	10052

Table 2: The datasets used in the performance analysis

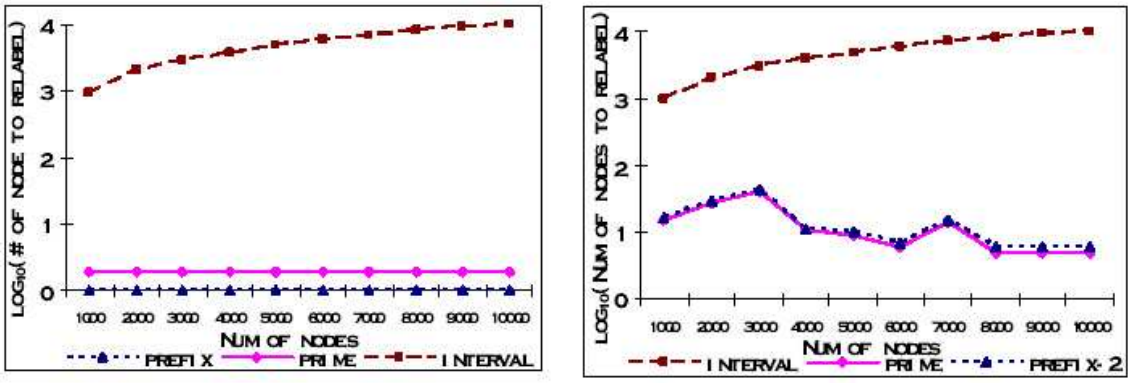


(a) **(b)**

Figure 16: Label sizes of the different datasets

As shown in **Fig**16-a we find the the different optimization steps for the prime number labeling scheme result in significant reduction in the label sizes, in **Fig**16-b we observe that the size of the labels is influenced by the data-set. Data-set D4 has low depth and high fan-out, as expected the prime labeling scheme performs very well. similarly the data-set D7 has a larger value for depth with low fan-out, in this case the prefix based scheme performs better.

Given a test case of 10 XML documents with the number of nodes ranging from 1000 to 10000, the experiments added a leaf node at the deepest level, the results observed are shown in **Fig** 17-a. **Fig** 17-b shows the results of the experiments that inserted a non-leaf node as the parent of a level 4 node. To summarize, we observe that the interval based scheme needs extensive re-labeling, where as the prefix and prime based schemes require only its descendants to be re-labeled.Let us now consider updates along with ordered queries. Using the Shakespears play dataset D8 described in table 2, experiments were carried out that updated the Hamlet XML file that contained an ordered sequence of ACT nodes. The update inserted a new ACT node and observed the number of nodes that require relabeling, the results are summarized in the **Fig** 18



(a) **(b)**

Figure 17: Updates on Leaf and non-leaf nodes

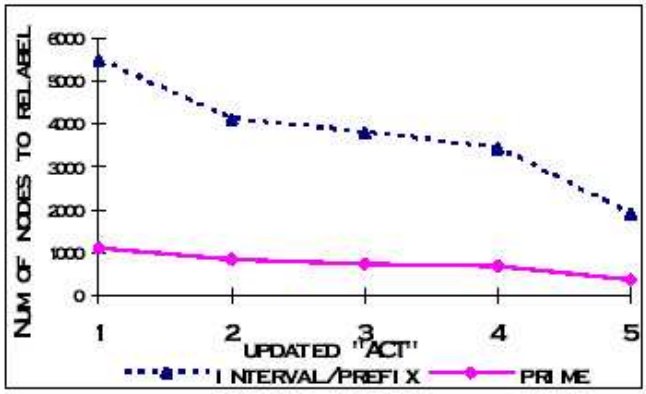


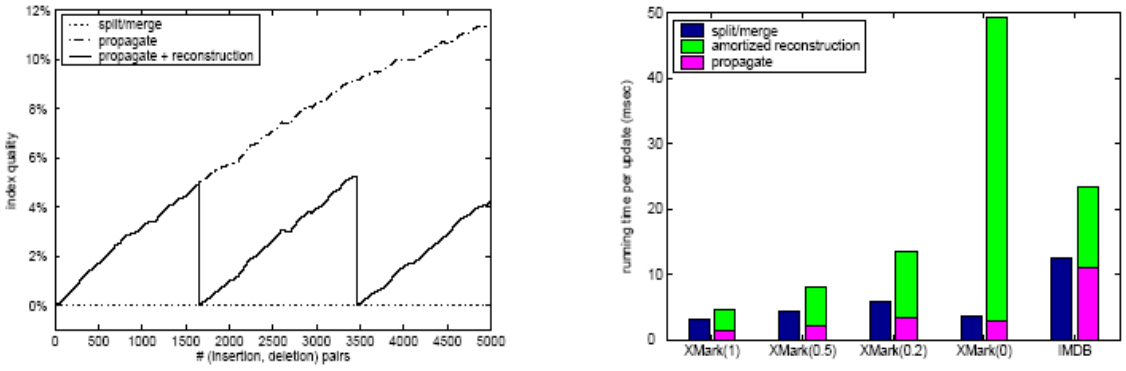
Figure 18: Ordered updates

4.2 Index Maintenance Comparison

In all, we have introduced two kinds of indexes, Strong DataGuide and the index family, and accordingly updating algorithms based on the property of the indexes. For the Strong DataGuide, because it is a label-oriented index which means that any two nodes, if having the same incoming label, they would be combined in one node set. So node merging would happen in inserting an edge while node splitting would appear in deleting an edge. This is a very intuitive algorithm but effective algorithm. For updating indexes in the index family based on simulation, there have been mainly three kind of algorithms: Propagate, propagate with reconstruction, split/merge. The later two could be considered as the improvement for the former one.

We can see the comparison from the **Fig** 19. **Fig** 19 (a) shows the size changed after the update with different algorithms. It is obvious that with the split/merge algorithm, the index size maintains stable and there is nearly no change with adding new insertions. While for the propagate algorithm , the size grows nearly linearly. This is not what we desire since we always would like the index size to be small enough. To make index size controllable , reconstruction is introduced with the propagate algorithm. The basic idea is simple: when the size of the index reaches a certain degree, the index is reconstructed. It seems great at a glance for the index has been keep in a bounded size. But the reconstruction always costs lost of computation making it not that attractive. It is clear that in **Fig** 19 (b) the reconstruction time is always much larger than that of the propagate or split/merge

part. While the time cost for the split/merge algorithm is always small enough. In one word, as an incremental algorithm, split/merge could always maintain the minimum size in reasonable time.



(a) **(b)**

Figure 19: Size comparison and Time comparison

5 Conclusion

In this report we have summarized the different approaches to address the issue of XML labeling. We have seen that for documents that are subjected to frequent updates the prime number labeling scheme is most compact and efficient. In the case of static documents however, the range based labeling scheme works best. We have also observed that not all schemes support efficient ordered querying. The prime number labeling scheme not only supports ordered querying, it also supports efficient ordered querying even in the presence of updates. One of the criticism we have of the prime number labeling scheme is that comparison of the label-size have been performed considering the self-labels alone (Refer **Fig** 7). The actual stored label includes both the self-label and the parent-label. From the paper it is not evident if the results obtained in **Fig** 16-b refer to the complete label or just the self-label. We have also observed that the computation of the SC value using the Euler's quotient function (refer equation 6) does not result in a correct solution even for the examples cited in [18]. The error can be attributed to the use of $\phi(x)$ the Euler's totient function as a substitute for "b_i" of equation 7. Lastly, we discussed the issue related to optimization 2 where the leaf nodes were labeled as 2^n, we noticed that the assumption of the chinese remainder theorem was invalidated as the set of labels were not relatively prime.

In this report, we have also made a detailed study for the incremental maintenance of XML structural indexes. First, several structural indexes are introduced including Strong DataGuide and the index family based on the similarity property . We make a comparison of these index structures and discuss the corresponding update algorithms. This report mainly talks about the update issue in edge processing aspect, at the same time, we have presented an efficient algorithm for efficiently updating sub-graph. The update mechanisms in this report is mainly based on the split/merge algorithm and we have shown that it has better performance in terms of both efficacy and efficiency as compared with propagate algorithm and propagate with reconstruction algorithms. To efficiently exploit the split/merge algorithm, a structure similar as the refinement tree is introduced with a modified k for updating the D(k)-index. We could carry out some further research in this area.

References

[1] W3C (1999) XML path language (XPath) 1.0. http://www.w3.org/tr/xpath.

[2] W3C (2000) Extensible markup language(XML)1.0,2nd edn. http://www.w3.org/TR/REC-xml/.

[3] W3C (2003) XQuery 1.0: An XML Query language. http://www.w3.org/TR/xquery/.

[4] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'02)*, 2002.

[5] T. Amagasa, M. Yoshikawa, and S. Uemura. ORS : A robust numbering scheme for xml documents. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, 2003.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of ACM SIGMOD*, 2002.

[7] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic xml trees. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS'02)*, 2002.

[8] R. Goldman and J. Widom. Dataguides : Enabling query formulation and optimization in semistructured databases. In *Proceedings of International Conference on Very Large Database (VLDB)*, 1997.

[9] H. He and J. Yang. Multiresolution indexing of xml for frequent queries. In *Proceedings of International Conference on Data Engineering (ICDE'04)*, 2004.

[10] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of ACM-SIAM Symposimu on Discrete Algorithms (SODA)*, 2002.

[11] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for structure indexs. In *Proceedings of the 28th VLDB Conference*, 2002.

[12] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'02)*, 2002.

[13] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[14] C. Qun, A. Lim, and K. W. Ong. D(k)-index : An adaptive structural summary for graph-structured data. In *Proceedings of ACM SIGMOD*, 2003.

[15] N. Santoro and R. Khatib. Labeling and implict routing in networks. In *The Computer J.*, 28:5-8, 1985.

[16] A. Silberstein, H. He, K. Yi, and J. Yang. Boxes : Efficient maintenance of order-based labeling for dynamic xml data. In *Proceedings of International Conference on Data Engineering (ICDE'05)*, 2005.

[17] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *Proceedings of ACM SIGMOD*, 2002.

[18] X. D. Wu, M. L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, 2004.

[19] G. M. Xing and B. Tseng. Extendible range-based numbering scheme for xml document. In *Proceedings of the International Conference on Information Technology : Coding and Computing (ITCC'04)*, 2004.

[20] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental maintenance of xml structural indexes. In *Proceedings of ACM SIGMOD*, 2004.

[21] 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.