An Efficient XML Encoding and Labeling Method for Query Processing and Updating on Dynamic XML data ⋆

Jun-Ki Min [1] , Jihyun Lee [2] , Chin-Wan Chung [*]

[1] *School of Internet-Media Engineering*
Korea University of Technology and Education
Byeongcheon-myeon, Cheonan, Chungnam, Republic of Korea, 330-708

[2,*] *Division of Computer Science*
Department of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
Daejon, Republic of Korea, 305-701

Abstract

In this paper, we propose an efficient encoding and labeling scheme for XML, called EXEL, which is a variant of the region labeling scheme using ordinal and insert-friendly bit strings. We devise a binary encoding method to generate the ordinal bit strings, and an algorithm to make a new bit string inserted between bit strings without any influences on the order of preexisting bit strings. These binary encoding method and bit string insertion algorithm are the bases of the efficient query processing and the complete avoidance of re-labeling for updates. We present query processing and update processing methods based on EXEL. In addition, the Stack-Tree-Desc algorithm is used for an efficient structural join, and the String B-tree indexing is utilized to improve the join performance. Finally, the experimental results show that EXEL enables complete avoidance of re-labeling for updates while providing fairly reasonable query processing performance.

Key words: XML; Query processing; Update;

⋆ The preliminary version of this paper was published in DASFAA 2007.

* Corresponding Author, Email: chungcw@islab.kaist.ac.kr, +82-42-869-3537, Fax: +82-42-869-3577

[1] Email: jkmin@kut.ac.kr,

[2] Email: hyunlee@islab.kaist.ac.kr

1 Introduction

For the purpose of sharing and integrating data collected over diverse application areas, the requirement of inter-operability has increased. Thus, W3C has proposed the eXtensible Markup Language(XML) [3]. XML data comprises hierarchically nested collections of elements, where each element is bounded by a start tag and an end tag that describe the semantics of the element. In addition, an element in XML data can contain either an atomic raw data(i.e., data value) or a sequence of nested subelements and can have a number of attributes composed of name-value pairs. Generally, an XML document is represented as a tree such as DOM [19]. The tree of XML data is implicitly ordered according to the visiting sequence of the depth first traversal of the element nodes. This order is called the *document order*.

To retrieve XML data, many query languages such as XPath [6] and XQuery [21] have been proposed. In order to search the irregularly structured XML data, path expressions are commonly used in these languages. Given a tree of XML data, the path information and the structural relationships of nodes should be efficiently evaluated. To determine the path information, diverse approaches such as path index approaches [5,10] and the reverse arithmetic encoding [17] have been proposed. In general, these techniques help to obtain the list of nodes which are reached by a certain path. Thus, the computation of the structural relationships is additionally required.

In order to facilitate the determination of relationships of nodes, the nodes in a tree of XML data are typically labeled in such a way that the structural relationship (e.g., the ancestor-descendant relationship) between two arbitrary nodes can be computed efficiently. Various labeling methods such as the region numbering scheme [15,24] and the prefix labeling scheme [22] have been proposed. In addition, structural modifications to the XML data can occur. For example, insertions of nodes change the structure of a tree of XML data, and the assigned labels may need to be changed. Thus, many researches [2,7,14,18,23] have been conducted in order to provide an efficient way to handle labels for updating XML data.

Our Contribution. In this paper, we consider the XML labeling scheme. We devise a novel XML encoding and labeling scheme, called EXEL (Efficient XML Encoding and Labeling). Our labeling scheme is simple but effective to compute the structural relationships as well as to support the incremental update.

The main contributions of this paper are summarized below:

- **An efficient XML encoding and labeling method (EXEL) is proposed:**. A novel binary encoding method to generate ordinal bit strings is

presented. The traditional region numbering scheme is mutated by using the bit strings instead of decimal values.

- **The query processing performance of EXEL is improved by the optimization techniques:** The query processing and optimization techniques for the region numbering scheme can be applied to EXEL. We use the Stack-Tree-Dec algorithm to efficiently conduct the structural join. Also, we utilize the String B-tree index to improve the join performance.

- **EXEL completely removes re-labeling for updates:** we devise an algorithm to generate a new bit string inserted between two bit strings without any influences on the order of preexisting bit strings. The generation algorithm is used to make a label for a newly inserted node in XML data. Thus, the re-labeling of preexisting labels can be completely avoided for inserting nodes in XML data.

In [16], we introduced the preliminary version of EXEL. There are several major extensions and modifications in this paper. First, in our previous work, we didn't discuss any query processing technique, while, in this paper, we applied query processing techniques such as the structural join algorithm and indexing to our labeling approach. As a result, we could obtain the improvement in the query processing performance. Second, we used an RDBMS for storing and querying XML data in [16]. But, the query processing performance could be decided by the performance of the query processor and the indexing strategy of an RDBMS. Therefore, in this paper, we stored XML data on files and implemented the structural join, indexing, and the updating programs. Third, in this paper, we additionally devised algorithms inserting a sibling and a parent while there is only the algorithm inserting a child in [16]. Fourth, in this paper, we reinforced experiments. We performed the query processing and the update on more diverse and larger data sets including real data sets. Also, we added the experiment to measure the indexing effect. Finally, we provided detailed explanation throughout the paper. Also, a large part of the section on experiments in this paper was newly written.

The remainder of the paper is organized as follows. In Section 2, we review various XML labeling schemes. We describe the details of EXEL in Section 3. Then, we present the query processing mechanism of EXEL in Section 4 and the update method of EXEL in Section 5. Section 6 contains the results of our experiments. Finally, in Section 7, we summarize our work.

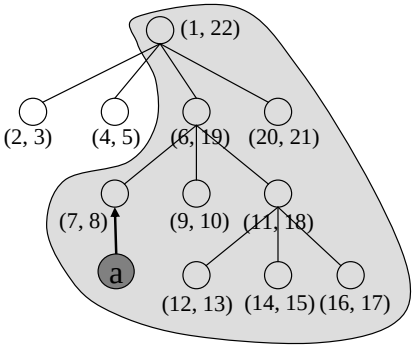


Fig. 1. The region numbering scheme

2 Related Work

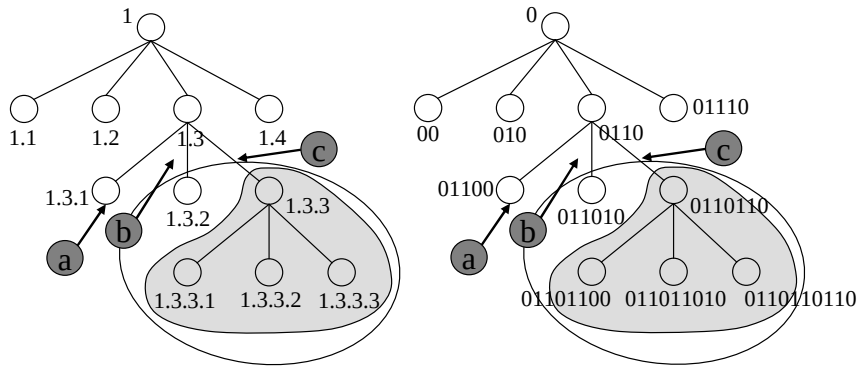
In this section, we introduce several representative labeling schemes and an encoding method for XML data. Also, we briefly mention the query processing techniques.

2.1 Region numbering scheme

In the region numbering scheme [15,24], each node in a tree of XML data is assigned a region consisting of a pair of start and end values which are determined by the positions of the start tag and the end tag of the node, respectively. Figure 1 shows a tree of XML data labeled by the region numbering scheme. Basically, ancestor-descendant relationships among nodes are determined by containment relationships of their regions. Additionally, in order to determine the parent-child relationship efficiently, the level of a node is used. Even though all structural relationships represented in XPath can be determined efficiently using <start, end, level>, an insertion of a node incurs re-labeling of its following and ancestor nodes. For example, in Figure 1, the insertion of node a incurs the re-labeling of the grey part. [15] and [2] have tried to solve the re-labeling problem by extending a region and using float-point values. However, the re-labeling problem can not be avoided for frequent insertions after all.

2.2 Prefix labeling scheme

In the prefix label scheme [7,18,22], each node in a tree of the XML data has a string label which is the concatenation of the parent's label and its own identifier (i.e., self-label). If there are two nodes x and y where x is an ancestor of y, then $label(x)$ is a prefix of $label(y)$. Dewey labeling scheme [22]



(a) Dewey labeling scheme (b) Binary labeling scheme

Fig. 2. The prefix labeling schemes

and Binary labeling scheme [7] do not require re-labeling for appending leaf nodes. Fig 2(a) shows a tree of XML data labeled by the Dewey labeling scheme and Fig 2(b) shows a tree of XML data labeled by the binary labeling scheme.

The above mentioned prefix labeling schemes do not require re-labeling for appending leaf nodes (e.g., inserting a node a in a tree of Figure 2(a) and Figure 2(b)). However, they cannot avoid the re-labeling for insertions between two sibling nodes. For example, in Figure 2(a) and Figure 2(b), the insertion of a node b incurs the re-labeling of nodes in the circle. In addition, an insertion of a node between parent and child nodes also incurs re-labeling. In Figure 2(a) and Figure 2(b), the grey part should be re-labeled by inserting a node c.

Recently, several prefix labeling approaches [12,18], which are tolerant for insertions, have been proposed. ORDPATH [18] follows a labeling principle similar to the Dewey labeling scheme. In order to avoid re-labeling, it uses only odd numbers for initial labels. When an insertion occurs, it uses an even number between two odd numbers and concatenates an odd number. [12] proposes Dynamic Level Numbering Scheme(DLN) based on the ORDPATH concept. DLN leaves a gap between two odd numbers during initial labeling in order to prevent the length of the label from rapid increasing for frequent insertions.

2.3 Prime number labeling scheme

The prime number labeling scheme [23] uses an inherent feature of the prime number which has only one and itself as its common divisors. Each node in a tree of XML data is assigned a unique prime number as its own self-label. The label of a node is a product of its parent node's label and its self-label.

Integer number	QED	Integer number	QED
1	112	9	23
2	12	10	232
3	122	11	3
4	13	12	312
5	132	13	32
6	2	14	322
7	212	15	33
8	22	16	332

Fig. 3. QED encoding of 16 numbers

To reflect the document order after insertions, the prime number labeling scheme uses the simultaneous congruence (SC) values based on Chinese Remainder Theorem. Even though re-labeling for nodes can be avoided for insertions, the SC values should be re-calculated, and the re-calculation consumes much time. Also, an insertion between parent and child nodes cannot be supported efficiently.

2.4 Encoding method for labeling

A binary encoding method QED [13] to avoid re-labeling for updates has been proposed. QED is a novel dynamic quaternary encoding which is orthogonal to a specific labeling scheme. QED uses only 4 digits, 0, 1, 2 and 3 and these are replaced by their binary values (i.e., 01 for 1, 10 for 2, and 11 for 3). QED partitions a list of numbers into three divisions, and assigns sequences of digits to the numbers located at the 1/3th and the 2/3th of the list. In Fig 3, the table shows the QED encoding of 16 numbers. Also, [13] presents an algorithm to insert a new QED code between two consecutive QED codes without any change of existing codes. For the encoding procedure in QED, see details in [13]. In QED, the label size is sometimes large because the label size increases by two bits for every insertion.

Recently, [14] proposed CDBS (Compact Dynamic Binary String) encoding which supports the insertion of a new CDBS between any two consecutive CDBSs with preserving the order and without any changes to the existing CDBSs. CDBS encoding can be applied to any labeling schemes. The algorithm generating a new binary string inserted between two consecutive CDBSs is the same as our approach but is independently developed. However, CDBS encoding scheme is quite different from our encoding scheme. The CDBS encoding algorithm is a recursive procedure. Given an interval $[P_L, P_R]$ of numbers, CDBS encodes a middle number P_M using the encoded value of the start number P_L and that of the last number P_R. Then the encoding algorithm is recursively applied to $[P_L, P_M]$ and $[P_M, P_R]$. For N numbers, CDBS assigns

empty string to the 0-th number and the $(N+1)$-th number. Therefore, CDBS needs to know the numbers to be encoded. In addition, since CDBS encodes the numbers randomly- not sequentially, the encoding algorithm needs a temporary array with size $O(N)$ for encoding N numbers. Also, the fixed length version of CDBS (F-CDBS) simply attaches zero or more '0' to the variable length version of CDBS (V-CDBS) which violates the insert-friendly property of V-CDBS.

According to the label size analysis, the sizes of binary strings generated by CDBS (i.e., $log_2 N$) and our enhanced binary encoding scheme (i.e., $log_2 N + 1$, see Section 3.2) are similar. The difference is too small to cause any degradation of the performance. Thus, the query processing performance and the scalability of EXEL using the enhanced binary encoding scheme will be similar to those of the containment labeling scheme using CDBS. In addition, we expect that the update performances of them are also similar since CDBS and our method generate the label for a newly inserted node by using the same algorithm.

In contrast, our encoding methods provide the following advantages: (1) since our methods generate binary strings sequentially and assign them to the stream of numbers, our encoding schemes don't need extra space to store the previously assigned binary strings. Also, in our method, the generating binary strings (i.e., encoding) and the labeling for nodes in XML data can be performed at the same time. On the other side, in CDBS, only after encoding the all numbers to be used for labels, it is possible to assign labels to nodes in XML data. Consequently, the amount of time for labeling of EXEL is shorter than those of labeling methods based on CDBS. (2) our basic binary encoding method does not need to know the number of numbers to be encoded. (3) our enhanced encoding method generating fixed length string obeys the insert-friendly property (i.e., Property 1) (See details in Section 3). (4) we provide inserting procedures for diverse insertion points and a kind of skewed insertion in Section 5.2. (5) we consider the index structure for our labeling technique to improve the query performance.

2.5 Query processing techniques

The structural join is regarded as the core operation in the XML query processing. Thus, the performance of the structural join determines the performance of the overall query processing. Various structural join algorithms [1,4,15,11] have been proposed. Stack-Tree-Dec algorithm [1] uses a stack to maintain elements that will be used later in the join. This algorithm needs only one sequential scan of each ordered element list in contrast to the sort-merge join proposed in [15] which may scan element lists many times.

In addition, the query performance can be improved by an index which efficiently removes unnecessary I/Os for scanning elements irrelevant to the join. [4,11] use the indices on labels, such as the B+tree and the R-tree.

3 Efficient XML Encoding and Labeling (EXEL)

In this section, we present an efficient XML Encoding and Labeling method (EXEL) which supports efficient query processing and update processing together.

3.1 Binary Encoding in EXEL

The label assigned to each node in a tree of an XML data should uniquely identify the corresponding node as well as be able to represent the order of the node to determine the structural relationships among nodes. Also, the label should be immutable for updates. Thus, EXEL uses bit strings which are ordinal as well as insert-friendly. The lexicographical order of bit strings is defined as follows:

Definition 1 Lexicographical order ($<$)
(i) 0 is lexicographically smaller than 1 (0 $<$ 1).
(ii) if two bit strings a and b are the same (=), a is lexicographically equal to b.
(iii) Given bit strings a, b, a' and b', $ab < a'b'$, if only if $a < a'$ or $a = a'$ and $b < b'$ or $a = a'$ and b is null (i.e., empty string), where $length(a) = length(a')$.

Consider a bit string s which ends with '0'. The largest bit string among bit strings which are lexicographically smaller than s is the s's longest prefix p (i.e., $s = p0$). However, we cannot generate any bit string which is greater than the prefix p and smaller than s. For example, there is no bit string which can be inserted between '110' and its longest prefix bit string '11'. Thus, we make a procedure that generates a bit string whose last bit is always '1'. In other words, if the last bits of any two consecutive bit strings are '1', we can insert a new one between the bit strings without any changes on them.

The bit strings for labeling in EXEL are generated by the following binary encoding method [3]:

[3] In the following rule, we use '+' as a binary addition and $\bigoplus$ as a binary concatenation

(1) The first bit string $b(1) = 1$.

(2) Given the i^{th} bit string $b(i)$, if $b(i)$ contains 0 bit then $b(i+1) = b(i) + 10$.

Otherwise, $b(i+1) = b(i) \bigoplus 0^k 1$, where k is the length of $b(i)$.

Bit strings generated by the above binary encoding method have the lexicographical orders presented in Definition 1. For example, 1<101<111<1110001. Also, according to the above generating rule, the bit string always ends with 1. Thus, our encoding scheme satisfies Property 1. Property 1 is the key of enabling to avoid re-labeling during the insertion of a node in the XML tree.

Property 1 *Given bit strings $s_1 1$ and $s_2 1$ generated by the above binary encoding method, if $s_1 1 < s_2 1$, then $s_1 < s_2$ in the lexicographical order.*

Theorem 1 presents the space requirement of our binary encoding scheme.

Theorem 1 *In order to encode N ordinal values, the maximum size of the binary encoding is $2^{\lceil log_2 log_2 N+1 \rceil} - 1$. In addition, the total size of the binary encoding is $\sum_{i=1}^{k} (2^{2^{i-1}-1} \cdot (2^i - 1))$ where $k = \lceil log_2 log_2 N + 1 \rceil$.*

Proof: (i) 1-bit string (i.e., 1) can represent only 1 value. (ii) 3-bit string (i.e., 101, 111) can represent 2 values. (iii) 7-bit string (i.e., 1110001,...,1111111) can represent 2^3 values. (iv) consequently, by the mathematical induction on k, $(2^k - 1)$-bit string can represent $2^{2^{k-1}-1}$ values.

Let $N = 2^0 + 2^1 + ... + 2^{2^{k-1}-1} = \sum_{i=1}^{k} 2^{2^{i-1}-1}$.
By the mathematical induction, $N = \sum_{i=1}^{k} 2^{2^{i-1}-1} < 2 * 2^{2^{k-1}-1} = 2^{2^{k-1}}$.
Therefore $k = \lceil log_2 log_2 N + 1 \rceil$. Consequently, $2^k - 1 = 2^{\lceil log_2 log_2 N+1 \rceil} - 1$.■

$2^x - 1$ bits can represent the $(2^0 + 2^1 + 2^3 + ... + 2^{2^{x-1}-1})$th ordinal values. 63 bits are enough to represent the 2^{31}th ordinal value. Generally, it is rare that a tree for an XML document has more than 2^{30} nodes. Therefore, 63 bits are sufficient to encode ordinal labels for general XML data.

3.2 Enhancement of Binary Encoding

The binary encoding of EXEL uses double lengthed bits of the necessary length in order to guarantee the lexicographical order among variable length bit strings. In the binary encoding method of EXEL, k-bit string has $(k-1)/2$ bits consisting of only 1. It is the superfluous part. For example, the 7-bit string begins from 1110001 to 1111111. In the binary encoding method, only $2^3 = 8$ ordinal values are represented since the first three bits are 111 and the last

bit is 1.

In order to remove the superfluous part, we devise another binary encoding method with a predefined length of a bit string. The predefined length is obtained from the total number of ordinal values which would be encoded. For labeling a tree of XML data, it is determined by the total number of nodes. The bit string with a predefined length is generated by the following rule:

Let N be the total number of values.

(1) The first bit string $b(1) = 0^{log_2 N}1$.

(2) Given i^{th} bit string $b(i)$, $b(i+1) = b(i) + 10$.

Note that, the bit strings generated by the above rule also obey the Property 1 since a bit string ends with 1 like the original binary encoding method. The enhanced encoding can save the storage space and is of advantage to the query processing since it generates shorter labels than the basic encoding. However, the basic binary encoding still can be useful when we cannot use the enhanced binary encoding in the case that we cannot find out the total number of nodes in XML data.

In order to encode N ordinal values, the binary encoding using a predefined length (i.e., enhanced binary encoding) needs $log_2 N+1$ bits for each bit string. Thus, the total size of the enhanced binary encoding is $N(log_2 N + 1)$ bits. In order to encode 22 values, the size of the longest bit string of the enhanced binary encoding is 6 bits while that of the basic binary encoding is 15 bits. In addition, the total size of the enhanced encoding is $22 * 6 = 132$ bits while that of the basic binary encoding is $2^0 * 1 + 2^1 * 3 + 2^3 * 7 + 2^7 * 15 = 1984$ bits. To conclude, the enhanced binary encoding scheme can effectively save the storage space, compared with the basic encoding scheme.

3.3 Labeling for XML Data

Like QED [13], the binary encoding presented in the previous section can be applied to both region numbering scheme and prefix labeling schemes. However, we choose the region numbering scheme in EXEL because of following two reasons. First, the query performance is affected by the kinds of operations to compute structural relationships. The prefix based scheme needs a prefix comparison operation to determine the ancestor-descendant relationship. It is more expensive than ordering operations (e.g.,$<$, $>$, $=$). Second, the prefix labeling scheme cannot intrinsically avoid the re-labeling problem for an inser-

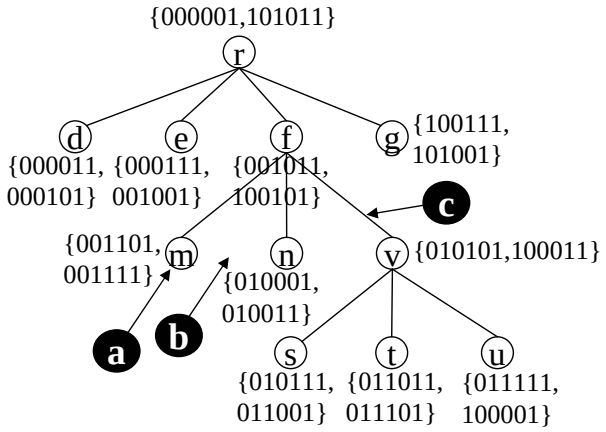


Fig. 4. EXEL with a predefined length

tion between child and parent nodes. Therefore, our labeling scheme is based on the region numbering scheme, and solves the update sensitive problem of the region numbering by using bit strings generated by the binary encoding scheme proposed in the previous section instead of decimal values.

The region numbering scheme uses the level information in order to determine the parent-child and the sibling relationship. However, the level information is sensitive to updates since the level should be changed when an ancestor node is deleted or a new node is inserted as an ancestor. Thus, EXEL uses the parent information instead of the level. Even though the parent information requires more storage space than level information, it is not changed for the insertion or the deletion of any ancestor node (except the parent). Furthermore, if we use the parent information in the query processing, the comparison of start and end values is not required in order to determine the parent-child and the sibling relationship. Thus, the query performance is improved.

Recall that both the basic binary encoding and the enhanced binary encoding can be used for labeling the XML tree. Fig 4 shows an example of a tree labeled by EXEL using the enhanced binary encoding with a predefined length. Since we should assign start and end values to total 11 nodes in the tree, we need to generate 22 ordinal bit strings using the binary encoding method. The following theorem presents the space requirement of EXEL using the binary encoding with a predefined length.

Theorem 2 *Given the total number of nodes M, the enhanced binary encoding needs $log_2 2M + 1$ bits for each bit string. Therefore, EXEL using the enhanced binary encoding requires $3(log_2 2M + 1)$ bits for each node's label since a label consists of start, end, and parent values. Consequently, the total size of labels for M nodes is $3M(log_2 2M + 1)$ bits.*

4 Query Processing

In this section, we show how to determine the structural relationships between nodes in EXEL. Also, we present the query processing techniques used in this paper such as efficient structural join algorithm and indexing.

4.1 Query Processing Operator

The computation of a structural relationship is the beginning of an XPath query processing. EXEL supports all XPath axes (i.e., ancestor, descendant, parent, child, following, preceding, following-sibling, and preceding-sibling) in the same way as the region numbering scheme because EXEL is based on the original region numbering scheme. For example, in Fig 4, f is an ancestor of t since $s_f(= 001011) < s_t(= 011011)$ and $e_t(= 011101) < e_f(= 100101)$, where ($s_x$,$e_x$) is the region of a node x.

Given x and y nodes whose regions are (s_x, e_x) and (s_y, e_y), and whose parent's start values are ps_x and ps_y, respectively, structural relationships between x and y can be determined as follows:

- **Ancestor and Descendant axes:** x is an ancestor of y, if and only if $s_x < s_y$ and $e_y < e_x$. For example, in Fig 4, f is an ancestor of t since $s_f(= 001011) < s_t(= 011011)$ and $e_t(= 011101) < e_f(= 100101)$.

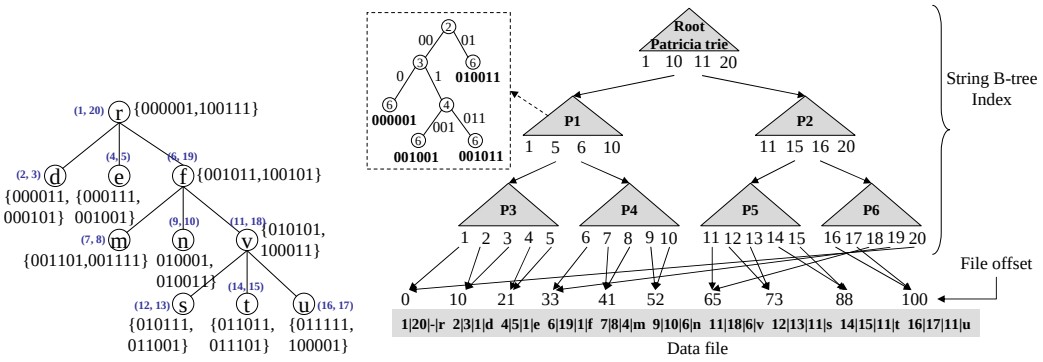
- **Parent and child axes:** x is a parent of y, if and only if $ps_y = s_x$. For example, in Fig 4, f is a parent of n since $ps_n(= 001011) = s_f(= 001011)$.

- **Following axis:** x is following y, if and only if $s_x > e_y$. For example, in Fig 4, g is a following node of m because $s_g(= 100111) > e_m(= 001111)$.

- **Preceding axis:** x is preceding y, if and only if $e_x < s_y$. For example, in Fig 4, e is a preceding node of v since $e_e(= 001001) < s_v(= 010101)$.

- **Following-sibling axis:** x is a following-sibling of y, if and only if $s_x > s_y$ and $ps_x = ps_y$. For example, in Fig 4, f is a following-sibling of d since $s_f(= 001011) > s_d(= 000011)$ and $ps_f(= 000001) = ps_d(= 000001)$.

- **Preceding-sibling axis:** x is a preceding-sibling of y, if and only if $s_x < s_y$ and $ps_x = ps_y$. For example, in Fig 4, g is a preceding-sibling of e since $s_g(= 100111) < s_e(= 000111)$ and $ps_g(= 000001) = ps_e(= 000001)$.



(a) XML Data Tree (b) String B-tree Index

Fig. 5. The index structure

4.2 Query processing and optimization technique

In order to efficiently process the structural join, we use the Stack-Tree-Desc algorithm [1]. In our experiment, we applied this algorithm to all labeling schemes and compared the structural join performance of EXEL with those of other labeling schemes.

We additionally observed the query performance of EXEL with indexing. [4] proposed a stack based structural join algorithm using B+-tree, called Anc-Des-B+ algorithm. Generally, B+-tree supports numeric data efficiently, however our encoding scheme generates a bit string which obeys the lexicographical order. Thus, we bring in the String B-tree [9] to the Anc-Des-B+ algorithm instead of the B+-tree. Fig 6 shows the Anc-Des-StringB algorithm. Note that Anc-Des-StringB algorithm is identical to Anc-Des-B+ algorithm except using String B-tree.

Anc-Des-StringB algorithm performs the structural join between two sorted element lists A and D which are potential ancestors and descendants, respectively. The algorithm sequentially scans A and D from their first elements and performs the structural join until one of the lists becomes empty. Variables a and d denote the currently accessed elements in A and D. During the execution, a stack (i.e., AStack) is used to keep elements of A which may be ancestors of elements remaining in D. Elements in A which are not ancestors of d (line 8) or elements in D which are precedings or ancestors of a (line 12) should be skipped in the structural join. In order to efficiently skip such unnecessary elements of A and D, the algorithm utilizes String-B trees (line 11 and line 15).

The String B-tree is a combination of B-trees and Patricia tries for internal-node indices. The String B-tree has the same worst-case performance as the

```
Algorithm Anc-Des-StringB( List A, List D )
begin
   /* A and D are the lists of potential ancestors and descendants, respectively,
    which are sorted in start value of their labels */
1. a := the first element of A, d := the first element of D
2. while( not at the end of A or D ) do
3.    if(a.start < d.start and d.end < a.end ) then
4.     AStack.push(all elements in A that are ancestors of d)
5.     a := the last element pushed
6.     output d as a descendants of all elements in AStack;
7.     d := the next element in D
8.    else if (a.end < d.start) then
9.     AStack.pop(all elements whose end < d.start)
10.     l := the last element popped
11.     a := the element in A (locate using String B-tree)
             having the smallest start that is larger than l.end
12.   else
13.    output d as a descendant of all elements in AStack
14.    if(AStack is empty) then
15.       d := the element in D (locate using String B-tree)
             having the smallest start that is larger than a.start
16.    else d := the next element in D
17.    end if
18.   end if
19. end while
end
```

Fig. 6. Anc-Des-StringB Algorithm

B-tree but it efficiently manages unbounded-length strings.

We have built a String B-tree index on sorted label values (i.e., start and end values) in the bottom-up manner. The label values are partitioned into value groups and each value group is stored into an internal node of the index. The value group in each internal node is composed of copies of the leftmost value and the rightmost value from each child node. Each internal node is transformed into the corresponding Patricia trie. Fig 5 illustrates an example of the String B-tree index built on labels of XML data. For simplicity, in Fig 5(b), we use the numeric values generated by the original region numbering scheme to represent the corresponding bit strings.

The index lookup is performed in the top-down manner. We traverse the index from the root Patricia trie by visiting and searching Patricia tries where the desired value is contained. Ultimately, we reach the leaf Patricia trie storing the value and get the offset indicating the location where the element having the value as the label is stored. For example, in order to find the element with

```
Algorithm MakeNewBitString( leftB, rightB )
begin
1.    if length(leftB) > length(rightB) then newB := leftB ⊕ 1;
2.    else newB := (rightB with the last bit changed to 0) ⊕ 1;
3.    return newB;
end
```

Fig. 7. MakeNewBitString Algorithm

start value of label '7', we start the traversal from root. '7' is between '1' and '10' which are copies of the leftmost and the right most value of the Patricia trie P1. Thus, next P1 is visited. Finally, the Patricia trie P4 is visited in the same way, and the location where the element 'm' is stored (i.e., '41') is returned by searching the Patricia trie P4.

5 Update

In this section, we present the update behaviors of EXEL. When a leaf node or a whole subtree is deleted, the re-labeling of any nodes is not incurred. However, if a non-leaf node is deleted, its children's parent value should be replaced by its parent's start value while start and end values for any nodes do not need to be updated. Since the delete algorithm is trivial, we omit it in this paper. In this section, we present the algorithms for diverse insertions.

5.1 Generation of an inserted bit string in EXEL

The algorithm *MakeNewBitString* in Fig 7 makes a new bit string between two preexisting bit strings. This algorithm can be applied to the basic encoding method and the enhanced encoding method since both of these encoding methods obey Property 1. But, for simplicity, we explain the update method based on the enhanced encoding method.

Our algorithm is incidentally coincident with that of CDBS [14] which was developed independently. Note that, our encoding scheme presented in Section 3 is quite different from that of CDBS.

If the length of the previous bit string *leftB* is longer than that of the next bit string *rightB*, the new inserted bit string *newB* is the previous one concatenated by 1. Otherwise, the next bit string with the last bit changed to 0 and appended by 1.

For example, when we make a new bit string between 101 and 111, the new bit string is 1101 (101<1101<111) generated by line 2 in the algorithm

```
Algorithm InsertChildOf( cur )
begin
1.    s_new := MakeNewBitString(s_cur, e_cur);
2.    e_new := MakeNewBitString(s_new, e_cur);
3.    ps_new := s_cur;
4.    Insert the new node with s_new, e_new, and ps_new;
end
```

Fig. 8. InsertChildOf Algorithm

MakeNewBitString. When an additional a bit string between 1101 and 111 is required, 11011 (1101<11011<111) is generated by line 1 in the algorithm. The length of bit string is increased by one bit for each insertion in contrast with QED increasing 2 bits per every insertion.

For bit strings generated by our binary encoding method, the lexicographical order has a property as follows.

Property 2 *Given bit strings $s_1 1$ and $s_2 1$ generated by the binary encoding method of EXEL, if $s_1 1 < s_2 1$, then $s_1 1 < s_2 01$ and $s_1 11 < s_2 1$, according to Definition 1 and Property 1.*

Thus, following the Theorem 3, a new bit string generated by the algorithm *MakeNewBitString* preserves the lexicographical order.

Theorem 3 *The bit string generated by the algorithm MakeNewBitString preserves the lexicographical order.*

Proof: If length($leftB$) > length($rightB$), then $leftB < newB (= leftB \bigoplus 1)$ (by Definition 1) and $newB < rightB$ (by Property2). Otherwise, given $leftB = s_1 1$ and $rightB = s_2 1$, $newB (= s_2 0 \bigoplus 1) < rightB (= s_2 1)$ (by Definition 1), and $leftB (= s_1 1) < newB (= s_2 01)$ (by Property 2)■

5.2 Update Processing

There are three kinds of insertions in XML data according to the positions in which nodes are inserted; inserting a child of a leaf node, inserting a sibling and inserting a parent.

The algorithm *InsertChildOf* in Fig 8 inserts a node as a child of a leaf node *cur*. In EXEL, a region of a child node is contained in that of its parent node. Thus, a region of a inserted node *new*, (s_{new}, e_{new}) should satisfy $s_{cur} < s_{new} < e_{new} < e_{cur}$. Additionally, the parent information of *new*, ps_{new} will be the start value of *cur* (i.e., s_{cur}). For example, in Fig 4, a node a is inserted into a child of a leaf node m. Thus, the region of a, (s_a, e_a) should satisfy

```
Algorithm InsertSiblingAfter( cur )
begin
1.    next := the start value of the nearest following-sibling of cur;
2.    if next = null
3.    then next := the end value of cur's parent;
4.    s_new := MakeNewBitString(e_cur, next);
5.    e_new := MakeNewBitString(s_new, next);
6.    ps_new := ps_cur
7.    Insert the new node with s_new, e_new, and ps_new;
end
```

Fig. 9. InsertSiblingAfter Algorithm

$s_m < s_a < e_a < e_m$. s_a and e_a will be computed as follows:

$$\begin{aligned} s_a &= \text{MakeNewBitString}(s_m, e_m) \\ &= \text{MakeNewBitString}(001101, 001111) = 0011101 \\ e_a &= \text{MakeNewBitString}(s_a, e_m) \\ &= \text{MakeNewBitString}(0011101, 001111) = 00111011 \end{aligned}$$

Additionally, $ps_a = s_m = 001101$.

The algorithm $InsertSiblingAfter$ in Fig 9 inserts a new node as a next sibling of a node cur. For example, in Fig 4, a node b is inserted after a node m. The next node n is the following-sibling of m. Thus, the region of b, (s_b, e_b) should satisfy $e_m < s_b < e_b < s_n$. s_b and e_b will be computed as follows:

$$\begin{aligned} s_b &= \text{MakeNewBitString}(e_m, s_n) \\ &= \text{MakeNewBitString}(001111, 010001) = 0100001 \\ e_b &= \text{MakeNewBitString}(s_b, s_n) \\ &= \text{MakeBitString}(0100001, 010001) = 01000011 \end{aligned}$$

Additionally, $ps_{new} = ps_m = 001011$.

The behavior of inserting a new sibling node before a node is similar to that of inserting a node after.

Inserting a child and inserting a sibling have been efficiently supported by other labeling schemes. However, inserting a parent node has not been handled by any previous labeling schemes. EXEL supports an efficient insertion of a parent node without re-labeling.

The algorithm $InsertParentOf$ in Fig 10 inserts a node as a parent of a node cur. In EXEL, the new parent node will be positioned between the previous and next sibling nodes of cur. Thus, the start and end values of the new

```
Algorithm InsertParentOf( cur )
begin
1.    prev := the end value of the nearest preceding-sibling of cur;
2.    if prev = null
3.    then prev := the start value of cur's parent;
4.    next := the start value of the nearest following-sibling of cur;
5.    if next = null
6.    then next := the end value of cur's parent;
7.    s_new := MakeNewBitString(prev, s_cur);
8.    e_new := MakeNewBitString(e_cur, next);
9.    ps_new := ps_cur
10.   Update ps_cur to s_new
11.   Insert the new node with s_new, e_new, and ps_new;
end
```

Fig. 10. InsertParent Algorithm

parent node will be values between the end value of the previous sibling node of *cur* (i.e., *prev*) and the start value of the next sibling node of *cur* (i.e., *next*). If there is no preceding-sibling or following-sibling of *cur*, the start or end values of the inserted node are bounded by the start or end values of the parent node of *cur*, respectively. The region of the new parent node of *cur* (s_{new}, e_{new}) should satisfy that $prev < s_{new} < s_{cur}$ and $e_{cur} < e_{new} < next$. Also, the previous parent of *cur* become the parent of *new*, and the parent of *cur* is changed to *new*. Thus, $ps_{new} = ps_{cur}$ and $ps_{cur} = s_{new}$. For example, in Fig 4, a node c is inserted as the parent of a node v. Because a node n is the previous node of v, $prev = e_n$. Also, since there is no following-sibling node of v, $next = e_f$. Therefore, the region of c should satisfy that $e_n < s_c < s_{cur}$ and $e_{cur} < e_c < e_f$. s_c and e_c are generated as follows:

$$
\begin{aligned}
s_c &= \text{MakeNewBitString}(e_n, s_{cur}) \\
&= \text{MakeNewBitString}(010011, 010101) = 0101001 \\
e_c &= \text{MakeNewBitString}(e_{cur}, e_f) \\
&= \text{MakeBitString}(100011, 100101) = 1001001
\end{aligned}
$$

In addition, $ps_c = ps_v = 001011$. After the insertion, the parent of v should be changed from f to c. Therefore, $ps_v = s_c = 0101001$.

The insertion of the parent incurs an increase of the levels of its all descendants. The original region numbering scheme uses the level information to find parent and child relationships, so it is needed to update the level information of the descendants. However, EXEL keeps the parent information instead of the level. Even if a node is inserted as an ancestor, the parents of the descendant are still unchanged except the child of the inserted node. Consequently, EXEL completely removes re-labeling for all kinds of updates.

In the case of a subtree insertion, the labeling can be efficiently handled. We first apply our labeling method to the subtree. Second, we generate a new bit-string x according to the inserting point (p, q) using the algorithm *MakeNewBitString*, then truncate the last bit (i.e., '1') of x. Let the truncated bit-string be x'. We complete labeling for the subtree by attaching x' as a prefix into the labels of the subtree's nodes. Since the prefixes of subtree's nodes are equal, lexicographical orders among labels of subtree's nodes are preserved. Note that, in the lexicographical order, $p \leq x' < q$. Thus, by Definition 1-(iii), labels of subtree's nodes (whose prefixes are x') are greater than p and smaller than q. Therefore, the generated labels still keep the lexicographical order among the pre-existing labels.

For the skewed insertions of N nodes, EXEL needs three bit strings (i.e., start, end and parent information) with the size increased by $O(N)$ as the labels of the newly inserted nodes in the worst case. The other deterministic encoding scheme designed to avoid re-labeling like CDBS [14] also has the same complexity of label size [7]. However, the algorithm to insert a subtree is a kind of a hybrid method combining dynamic and static labeling schemes. Therefore, our algorithm to insert a subtree requires an increase in the size of bit strings by $O(log_2 N)$ for the labels in the worst case since the bit string generated by our encoding is $O(log_2 N)$ and the prefix generated by the algorithm *MakeNewBitString* is smaller than or equal to the maximum size of all labels.

6 Experiments

6.1 Experimental Environment

The experiments were performed on an Intel Pentium 3GHz with 1GB memory, running Windows XP. We implemented labeling, query processing and update processing codes in Java. We stored the labeled XML data on the local disk. For each individual element name n, we created a file named $elementfile_n$, and stored the corresponding records for elements with the name. Each record is composed of the label and the name of the element. In order to find the structural relationships between two elements, we load the corresponding element files of the elements into a buffer sized 64KB.

In addition, we created a file called $datafile$ which contains records of all elements in the original XML data. The data file is used for wild card '$*$' queries which need to scan the entire data. In the element files and the data file, records were sorted by the label in ascending order.

All experiments were repeated 10 times and we used the average of the pro-

Table 1
Data Set

Data	Name	Size(MB)	# of elements	Max.Depth	Avg.Depth
XMark	X1	1	17,132	12	5
	X50	50	476,646		
	X115	115	1,666,315		
	X500	500	7,182,068		
Shakespeare	S7	7.7	179,690	7	5
Nasa	N24	24	476,646	8	6
Treebank	T84	84	2,437,666	37	8

cessing times excluding the minimum and maximum values.

Labeling methods We implemented EXEL using the binary encoding with a predefined length. As presented in Section 2, there are many labeling techniques. It is quite hard to implement all labeling techniques. Thus, among them, we choose the representative techniques: region numbering and ORDPATH (a prefix technique) to show the efficiency of EXEL.

The label organization of each label scheme is as follows: In the region numbering scheme, each label consists of $< start, end, parent >$ where the type of each constituent is a decimal value. The label organization of EXEL is the same with that of the region numbering scheme except that the type of each constituent is a bit string. The label of ORDPATH is composed of $< ordpath, parent >$ where the type of each component is a bit string. The *parent* information is the start value of the parent element in the region numbering scheme and EXEL, and the self-label of the parent element in ORDPATH.

Data Set We evaluated EXEL using XMark benchmark data [20], Shakespeare data [8], Nasa data, and Treebank data [4]. The XMark data contains the information of the internet auction. The Shakespeare data is the collection of plays of Shakespeare. In our experiment, we concatenated 37 plays of Shakespeare into a single XML document. The Nasa data is astronomical data. The Treebank data is encoded DB of English records of Wall Street Journal. The characteristics of the data sets in our experiments are summarized in Table 1. In Table 1, 'Name' column presents the abbreviation name of the corresponding data.

Query Set The queries used in our experiments are described in Table 2. The first character in a query name indicates the data set on which the query is executed: 'X' denotes XMark, 'S' is for Shakespeare, 'N' is for Nasa, and 'T' is for Treebank. In our experiments, we evaluated the query performance for four kinds of axes, descendant, child, following, and following-sibling in XPath. The number in a query name denotes the type of a query according to the axis

[4] XML Data Repository. http://www.cs.washington.edu/research/xmldatasets

Table 2
Query Set

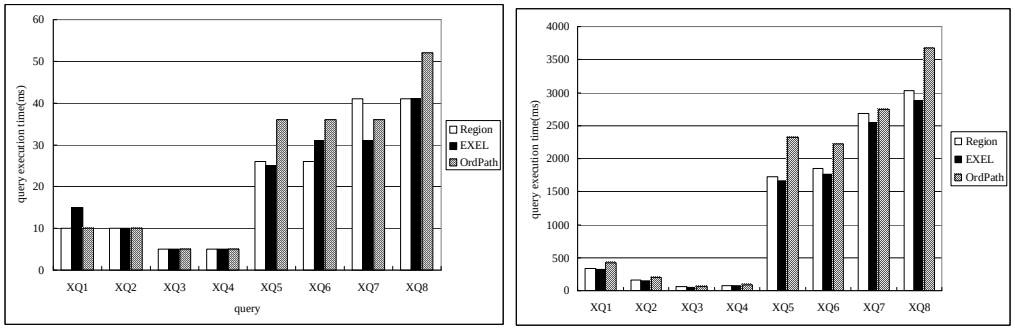
Name	Query Definition
XQ1	//item//incategory
XQ2	//item/name
XQ3	//open_auction[n]/following::bidder (where n = 60 for X1, 2580 for X50, 6000 for X115, and 258000 for X500)
XQ4	//bidder[n]/following-sibling::bidder (where n = 354 for X1, 12845 for X50, 29743 for X115, and 129101 for X500)
XQ5	//item//$*$
XQ6	//person/$*$
XQ7	//closed_auction//$*$
XQ8	//open_auction//$*$
SQ1	//ACT//TITLE
SQ2	//SPEECH/LINE
SQ3	//TITLE[10]/following::SPEECH
SQ4	//SPEECH[5]/following-sibling::SPEECH
SQ5	//SPEECH//$*$
NQ1	//field//definition
NQ2	//author/initial
NQ3	//journal[20]/following::name
NQ4	//journal[10]/following-sibling::journal
NQ5	//journal//$*$
TQ1	//NP//JJ
TQ2	//NP/NP
TQ2	//CC[20]/following:VP
TQ2	//NP[1]/VP
TQ5	//NP//$*$

contained in the query(i.e., 1 for descendant, 2 for child, 3 for following, 4 for following-sibling, and over 4 for wildcard '*' query). Other axes can be handled by the similar ways with these axes. Therefore we omitted the evaluation for them.

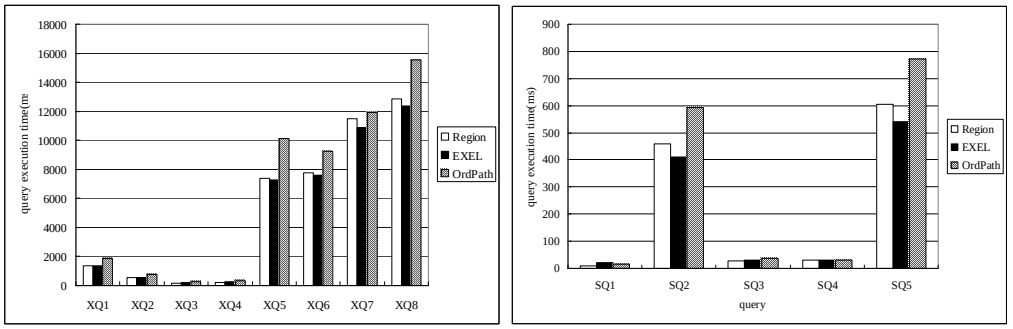
In XMark queries, variables in order predicates (e.g., n in XQ4) have the order values of the current elements positioned in the midst among its siblings with the same name. For example, in XQ3, open_auction[60] is the midst one among about 120 'open_auction' elements in X1.

6.2 Experimental Results

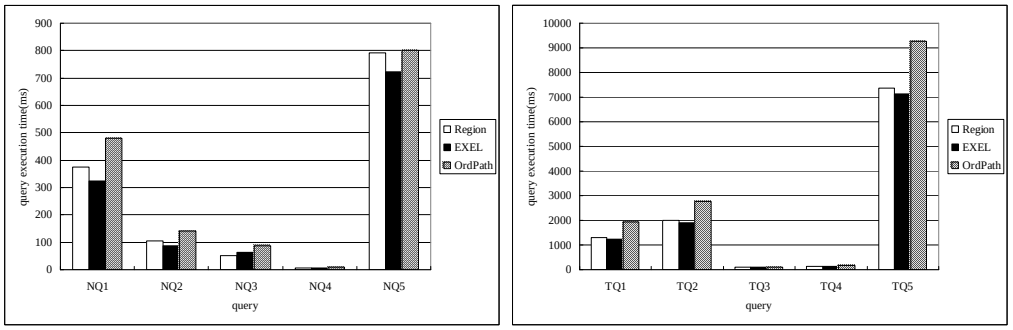
In this section, we analyze the experimental results for query processing, update processing, and storage size according to labeling schemes.



(a) X1 (b) X115



(c) X500 (d) S7



(e) N24 (f) T84

Fig. 11. Query execution time

6.2.1 Query Performance

Fig 11 shows the query execution time of each labeling schemes for various XML data sets. As shown in this figure, EXEL provides the best query processing performance among labeling schemes in almost all cases.

First, EXEL is superior to the ORDPATH since basically EXEL supports simple and efficient computations for all kinds of structural relationships like

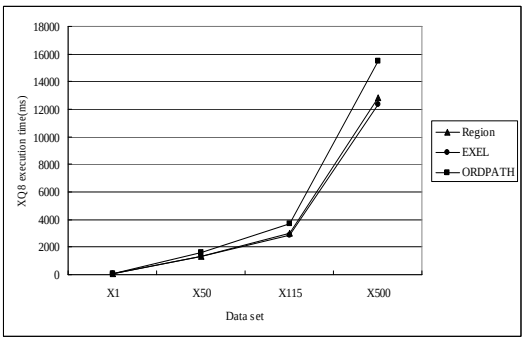


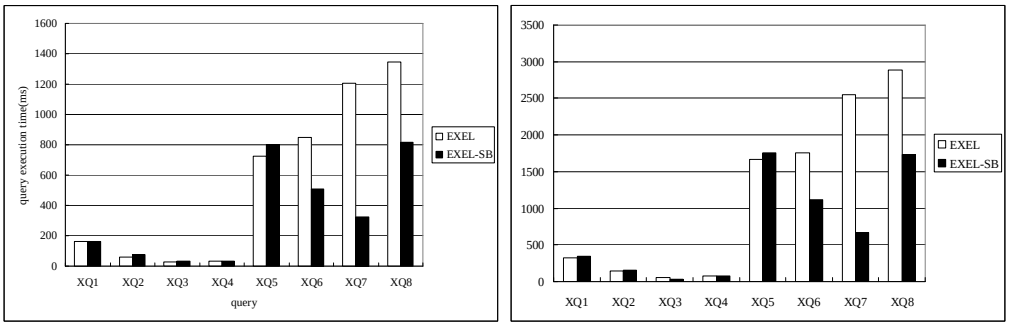
Fig. 12. Scalability for labeling schemes (query XQ8)

the region numbering scheme. This is achieved by the binary encoding scheme generating the ordinal bit strings which can be effectively adopted to the region numbering scheme. Also, in order to determine an ancestor-descendant relationship between two elements, ORDPATH needs a prefix comparison operation for two bit string labels, which requires the comparison of bit strings as the length of the shorter bit string. This operation is more time consuming than the lexicographical ordering operation (i.e., $>$, $<$, and $=$) which compares two bit strings to the first different bit. For example, when we compare two bit strings '100011101' and '10111101', the prefix comparison needs 8 bit-comparisons while the lexicographical ordering operation needs only 3 bit-comparisons. Therefore, in order to determine the ancestor-descendant relationship, EXEL outperforms ORDPATH, even if EXEL requires two lexicographical ordering comparisons for start and end values, while ORDPATH needs only one prefix comparison.

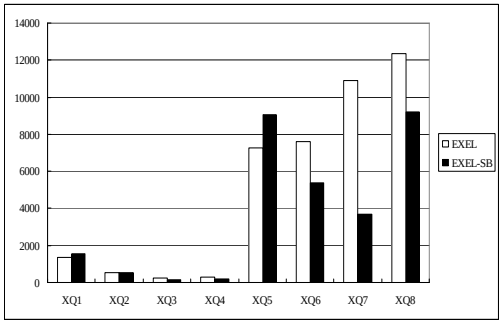
Second, the performance of EXEL is better than the original region numbering scheme even if the size of labels in the original region numbering scheme is shorter than that in EXEL. The reason is that the label of the original region numbering has the numeric order which needs comparisons of the entire byte sequences of two values, while the label of EXEL has the lexicographical order which is determined by comparing only the front parts of byte sequences of two values.

The performance gap between EXEL and other labeling schemes increases as the size of XML data gets larger. We can see the the query processing performance for various the data sizes using XMark data in Fig 12.

Fig 13 shows the effectiveness of the use of an index structure in the query processing. In this figure, EXEL-SB denotes the performance of EXEL with the String B-tree Index. The index helps to avoid unnecessary comparisons by skipping the scanning of irrelevant elements. Especially, the query performance of some queries (i.e., XQ6, XQ7, and XQ8) with a wildcard '*' is largely improved since a large part of the data can be skipped by indexing. For example, in Fig 13(c), the query execution time of the query XQ7 decreased from



(a) X50 (b) X115



(c) X500

Fig. 13. Query execution time (EXEL vs. EXEL with String B-tree Index)

Table 3
Update Query

Name	Insert location	Location details
UQ1	//person[m]	m = 5482 for X50, and 12250 for X115
UQ2	//closed_auction[c]	c = 2096 for X50, and 4875 for X115
UQ3	//namerica[1]	

10885 ms to 3682 ms, since the number of buffer I/Os was reduced. However, occasionally, the index can be unhelpful. In case that data size or the scope skipped by the index is small, the index lookup overhead outweighs the improvements obtained by the index. For example, in Fig 13(c), in case of the query XQ5, the performance of EXEL-SB is a little worse than EXEL because there is no part skipped by indexing.

6.2.2 Update Performance

We evaluated the performance of three kinds of insertions; inserting a child node of a leaf node, inserting a next sibling node of a node, and inserting a parent node. The effect of inserting a subtree on preexisting labels is the same

Table 4
The performance of inserting a child

Query	Labeling Scheme	Data	Time(ms)	#.Relabeling	Data	Time(ms)	#.Relabeling
UQ1	EXEL	X50	5	0	X115	15	0
	ORDPATH		5	0		15	0
	Region numbering		3,570	397,914		8,546	931,108
UQ2	EXEL	X50	5	0	X115	5	0
	ORDPATH		5	0		5	0
	Region numbering		974	40,999		2,120	95,572
UQ3	EXEL	X50	5	0	X115	5	0
	ORDPATH		5	0		5	0
	Region numbering		4,411	484,739		10,177	1,126,901

Table 5
The performance of inserting a sibling

Query	Labeling Scheme	Data	Time(ms)	#.Relabeling	Data	Time(ms)	#.Relabeling
UQ1	EXEL	X50	229	0	X115	489	0
	ORDPATH		239	0		510	0
	Region numbering		3,927	397,913		9,306	931,107
UQ2	EXEL	X50	448	0	X115	989	0
	ORDPATH		453	0		989	0
	Region numbering		1,312	40,998		2,974	95,571
UQ3	EXEL	X50	180	0	X115	359	0
	ORDPATH		193	0		365	0
	Region numbering		4,500	484,738		10,599	1,126,900

Table 6
The performance of inserting a parent

Query	Labeling Scheme	Data	Time(ms)	#.Relabeling	Data	Time(ms)	#.Relabeling
UQ1	EXEL	X50	229	1	X115	500	1
	ORDPATH		224	21		453	11
	Region numbering		5,359	397,934		8,859	93,118
UQ2	EXEL	X50	453	1	X115	1,015	1
	ORDPATH		416	29		937	21
	Region numbering		2,260	41,027		2,120	95,592
UQ3	EXEL	X50	182	1	X115	396	1
	ORDPATH		1,109	112,078		2,427	259,689
	Region numbering		5,369	596,816		12,198	1,386,589

as that of inserting a node. Therefore, we omitted the experiment of inserting a subtree.

Table 3 shows the description of updates conducted in our experiments. 'Insert location' denotes a current node on which updates occur. We executed thee kinds of insertions on each location. For instance, the update query UQ1 inserts a new node as a child, a sibling, or a parent of the current node indicated by the path '//person[5482]'.

Table 4, Table 5, and Table 6 show the performances of inserting a node for three kinds of insertions. In the region numbering scheme, the re-labeling was inevitable for all kinds of insertions, and the scope requiring the re-labeling is wide. Thus, the update performance of the region numbering scheme is the worst among labeling schemes for all kinds of insertions.

Table 4 shows the update performance of inserting a child node. While EXEL and ORDPATH did not incur re-labeling, the region numbering scheme cannot avoid re-labeling over a wide scope. Therefore, the update performances of the region numbering scheme is the worst.

Table 5 shows the update performance of inserting a next sibling node. In EXEL and ORDPATH, re-labeling of nodes is not incurred. Thus, there is no difference of update performance by the re-labeling. However, as shown in algorithm of Fig 9, in order to generate a label for a newly inserted node, EXEL needs to know the label of the next sibling. ORDPATH also need the information. The performance of EXEL to find the sibling node is slightly better than ORDPATH, so the performance of EXEL is a little better than that of ORDPATH.

Table 6 shows the update performance of inserting a node between parent and child nodes. ORDPATH should re-assign labels for the child and its all descendants while EXEL needs only one re-labeling which is the change of the parent value of the child node. Thus, if there are many descendants of the child node, EXEL outperforms ORDPATH. For example, in Table 6, see the results of the update query UQ3. Since the difference of the number of re-labelings between EXEL and ORDPATH is big, the performance of EXEL is much better than that of ORDPATH.

In addition, as presented in algorithm of Fig 10, EXEL needs the labels of the previous and next sibling nodes in order to generate a new label for an inserted node. But ORDPATH doesn't require the labels of the relevant nodes. Thus, EXEL requires the additional time to find the sibling information compared with ORDPATH. In case that the child node has a small number of descendants, the overhead to find the sibling information can overwhelm the advantage of avoiding re-labeling. For instance, in Table 6, UQ1 needs only a small number of re-labelings, so the performance of ORDPATH is a little better than that of EXEL.

6.2.3 Storage Space

Table 7 shows the size of the labeled data for each labeling scheme. For all data sets, the region numbering scheme requires the smallest storage space among labeling schemes. EXEL needs larger space than the region numbering scheme due to the use of the insert-friendly bit string. However, the significant

Table 7
Labeled Data Size

Labeling scheme	Labeled Data Size (MB)						
	X1	**X50**	**X115**	**X500**	**S7**	**N24**	**T84**
Region numbering	302	13,605	33,446	154,600	3,140	8,960	41,554
EXEL	326	15,732	36,551	157,706	3,165	8,989	44,792
ORDPATH	320	14,830	35,002	156,163	3,483	9,743	46,005

improvement of the update performance according to the use of the bit strings compensates for the space overhead.

Although EXEL uses three binary coding values (i.e., start, end, and parent information as a bit string type), the efficiency of the storage space for EXEL compared with ORDPATH depends on the data. In a prefix based scheme, an element in a deep level is assigned a long label. Thus, in case that many elements are located in deep levels of the tree of an XML data, ORDPATH is more disadvantageous than EXEL. For example, ORDPATH requires more space than EXEL for Treebank and Nasa data. On the other hand, EXEL can need more space than ORDPATH for data where the average depth of XML data tree is shallow (i.e., XMark data).

7 Conclusion

In this paper, we proposed EXEL, an efficient XML encoding and labeling method which supports efficient query processing and updates.

A novel binary encoding method used in EXEL generates ordinal and insert-friendly bit strings. In order to reduce the label size, we enhanced the binary encoding method using a predefined length which is determined by the total number of nodes in a tree of XML data. EXEL is a variant of the region numbering scheme using bit strings generated by the novel binary encoding method. Therefore, all efficient query processing techniques for the original region numbering scheme can be applied to EXEL. We took advantage of the Stack-Tree-Desc algorithm for efficient computing of structural joins. Also, we improved the join performance by using the String B-tree index.

The experimental results show that EXEL provides fairly reasonable query performance. In addition, we can observe that EXEL can save much time in updates through the complete avoidance of re-labeling.

Acknowledgement This research was supported by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Advancement. (grant number IITA-2008-C1090-0801-0031)

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of ICDE 2002*, pages 141–152, 2002.

[2] T. Amagasa and M. Yoshikawa. QRS: A Robust Numbering Scheme for XML documents. In *Proc. of ICDE 2003*, pages 705–707, 2003.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, http://www.w3.org/TR/REC-xml, 2004.

[4] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Join on Indexed XML Document. In *Proc. of VLDB 2002*, pages 263–274, 2002.

[5] C.-W. Chung, J.-K. Min, and K.-S. Shim. APEX: An Adaptive Path Index for XML Data. In *Proc. of ACM SIGMOD 2002*, pages 121–132, 2002.

[6] J. Clark and S. DeRose. XML Path Language(XPath) Version 1.0. W3C Recommendation, http://www.w3.org/TR/xpath, 1999.

[7] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proc. of PODS 2002*, pages 271–281, 2002.

[8] R. Cover. The XML Cover Pages. http://www.oasis-open.org/cover/xml.html, 2001.

[9] P. Ferragina and R. Grossi. The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications. *Journal of the ACM*, 46(2), 1999.

[10] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of VLDB 1997*, pages 436–445, 1997.

[11] T. Grust. Accelerating XPath Location Steps. In *Proc. of SIGMOD 2002*, pages 109–120, 2002.

[12] T. Harder, M. Haustein, C. Mathis, and M. Wagner. Node labeling schemes for dynamic XML documents reconsidered. *Data and Knowledge Engineering*, 60(1), 2007.

[13] C. Li and T. W. Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *Proc. of ACM CIKM 2005*, pages 501–508, 2005.

[14] C. Li, T. W. Ling, and M. Hu. Efficient processing of updates in dynamic xml data. In *Proc. of ICDE 2006*, page 13, 2006.

[15] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Expressions. In *Proc. of VLDB 2001*, pages 367–370, 2001.

[16] J.-K. Min, J. Lee, and C.-W. Chung. An Efficient Encoding and Labeling for Dynamic XML Data. In *To appear in Proc. of DASFAA*, 2007.

[17] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS: A Queriable Compression for XML Data. In *Proc. of ACM SIGMOD 2003*, pages 122–133, 2003.

[18] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of ACM SIGMOD 2004*, pages 903–908, 2004.

[19] R. W. Philippe Le Hegaret and L. Wood. XML Path Language(XPath) Version 1.0. http://www.w3.org/DOM, 2005.

[20] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of VLDB 2002*, pages 974–985, 2002.

[21] D. C. Scott Boag, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. W3C Recommendation, http://www.w3.org/TR/xquery/, 2005.

[22] 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 *Proc. of ACM SIGMOD 2002*, pages 204–215, 2002.

[23] X. Wu, M. L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proc. of ICDE 2004*, pages 66–78, 2004.

[24] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of ACM SIGMOD 2001*, pages 425–436, 2001.