RESEARCH

Open Access

A new effective method for labeling dynamic XML data

Eynollah Khanjari and Leila Gaeini*

*Correspondence:
leila.gaeini@gmail.com
Iran University of Science
and Technology, Tehran, Iran

Abstract

Query processing based on labeling dynamic XML documents has gained more attention in the past several years. An efficient labeling scheme should provide small size labels keeping the simplicity of the exploited algorithm in order to avoid complex computations as well as retaining the readability of structural relationships between nodes. Moreover, for dynamic XML data, relabeling the nodes in XML updates should be avoided. However, the existing schemes lack the capability of supporting all of these requirements. In this paper, we propose a new labeling scheme which assigns variable-length labels to nodes in dynamic XML documents. Our method employs the FibLSS encoding scheme that exploits the properties of the Fibonacci sequence to provide variable-length node labels of appropriate size. In XML updating process, we add a new section only in the new node's label without relabeling the existing nodes while keeping the order of nodes as well as preserving the structural relationships. Our labeling method is scalable as it is not subject to overflow, and as the number of nodes to be labeled increases exponentially, the size of labels grows linearly, which makes it suitable for big datasets. It also has the best performance in computational processing costs compared to existing approaches. The results of the experiments confirm the advantages of our proposed method in comparison to state-of-the-art techniques.

Keywords: XML labeling scheme, Dynamic labeling, XML query processing, XML updates

Introduction

XML is a semi-structural and standard document format to exchange data. Elements in XML documents are regular and there are structural relationships between them [1]. In fact, processing queries in XML should recognize these structural relationships. They also should determine the order of elements in a document. Node labeling in XML data is one way to increase the efficiency of query processing. Labeling means allocating a unique identifier to each node in XML documents [2]. A labeling scheme encompasses traversal or browsing the document, analyzing the elements, and assessing available relationships between elements. So, it should generate small enough labels in order to be processed efficiently both in initial label assigning as well as when queries are issued.

A challenging problem of the existing labeling schemes is the need for relabeling nearly all the existing nodes after inserting new nodes in XML documents. While update in XML data is a usual operation in many real-world applications, e.g. stream data, relabeling will influence the query performance, especially when large-size labels are assigned

to the nodes. In this paper, we introduce a new method to label XML documents. In this method, we pay attention to the evaluation criteria of labeling schemes, so besides the efficiency of the method, it can optimize queries on dynamic XML data without relabeling the existing nodes.

The labeling method supports the structural relationships, AD (Ancestor–Descendant), PC (Parent–Child), DO (Document Order)¹ and Sibling relations, between nodes. The method uses a simple algorithm to produce small-size labels. Experimental results show that the proposed method is efficient in terms of the label size, labeling time, querying time and update/insertion time. The results are compared against state-of-the-art labeling methods.

In the following, we provide a summary of the related works in “[Related works](#)” section. Our proposed method is introduced in “[Proposed method](#)” section. In “[Results and discussion](#)” section, the results of the experiments are reported and analyzed in terms of several performance criteria. The paper is concluded in “[Summary and conclusions](#)” section, followed by “[Future work](#)” section which gives some perspectives on future works.

Related works

Several methods have been proposed for labeling XML documents. Existing methods can be summarized and can be grouped into four categories:

- Range-based labeling schemes: These schemes are characterized by incorporating <START, END> arguments to the labels [3]. The START and END components of a label determine the start and the end of the corresponding position of nodes in the XML tree. The schemes under this category [1, 3–10] can determine AD relationships. Labels’ size in these schemes are compact. The depth of the tree does not influence the size of the labels. The main challenge of these schemes is they do not support dynamic XML documents. That is, after inserting new nodes, relabeling is inevitable. Inability to detect all the structural relationships is another weakness of the range based labeling schemes.
- Prefix-based labeling schemes: In the schemes in this category [11–25], each node label involves the label of its ancestor besides its own. The structural relationships are determined by looking at the labels. However, the size of the labels increases especially for nodes occurring in deeper levels of the XML tree, yielding more storage overhead. Moreover, inserting a new node in the rightmost places at a level does not need to re-label other nodes but inserting in elsewhere will affect the label of other nodes. In [26], the deleted labels are reused for encoding newly inserted nodes, which could effectively lower the label size.
- Multiplication based labeling schemes: These schemes [5, 27–35] use prime numbers with multiplication and division operations to determine the relationships among the nodes. Prime numbers are exploited in order to support the uniqueness of the labels. However, this leads to an exploded space overhead as well as computation overhead for determining the order of nodes in the document.

¹ Following sibling and preceding sibling.

- Vector based labeling schemes: These schemes use a vector order [36–41]. The methods in this category are defined based on vector ordering and commonly are orthogonal to the other schemes, i.e. they can be applied to other labeling schemes. The basic problem in these schemes is their computation time overhead for general cases of a new node insertion as well as for querying nodes in different levels.

Our proposed scheme exploits the advantages of prefix-based labeling schemes and provides solutions to overcome their disadvantages.

Proposed method

In this section, we provide a detailed presentation of how the whole process of node labeling is taken place. Firstly, we illustrate how to label the nodes using an example. Then, we describe how a newly inserted node is labeled and how the order of nodes is preserved, and how the structural relationships among nodes are determined. Also, we analyze the label size requirements of our labeling method.

The proposed labeling method uses binary bit values (0 and 1) for specifying the identity of a node. For a node in an XML tree, we keep the level at which the node is in the tree, the identifier of its parent, and the node's identifier, which we denote respectively by $\langle \text{Level}, \text{Parentid}, \text{Selfid} \rangle$, where Selfid is the identifier of the node itself. By the "label", we mean the whole collection composed of the three parts.

Binary values are used for each of the three parts with appropriate minimum length. For example, for a length of one bit, the first node takes Selfid 0 and the second one takes 1 in a given level. To label the third node in that level, two bits are required and the third node takes 00 for its Selfid, the fourth takes 01, and so on. When new nodes are inserted, the label of the existing nodes remains unchanged. To preserve the uniqueness of the labels, some bits are added to the end of Selfid. We call the augmented bits 'update identifier', which is denoted by UpID. Note that our labeling method keeps the ordering of nodes and supports the structural relationships among them. Figure 1 depicts an example XML tree with nodes labeled by the proposed method.

As shown in Fig. 1, the length of the labels is variable; so, we should take this into account in the labeling scheme. To do this, a fixed label length could be assigned for all nodes. In this way, there is no need to store the size of labels. However, it is clear that a fixed length label

storage schemes are subject to overflow by the update process which requires the relabeling of all existing nodes [30]. Alternatively, it is possible to use a variable length labeling scheme which requires the size (length) of the label to be stored in addition to the label itself. It is not the case to use a fixed length of bits for storing the variable size of the labels because the original fixed-length of bits will eventually be too small, requiring all existing nodes to be relabeled, or this leads to significant wastage of storage space if a very large fixed-length of bits is used [42]. In other words, not only labels should have variable sizes but also the length field of the labels should be stored, and be identified, using a variable length scheme. Thus, we store each label's length using the encoding method called FibLSS [43] along with the label itself, just before it.

FibLSS encoding

The FibLSS method uses the Fibonacci numbers sequence. It is defined such that each term in the sequence is the sum of the previous two terms:

The Fibonacci sequence is given by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$, where $F_0 = 0$ and $F_1 = 1$.

Note that, every positive integer n has a unique representation as the sum of one or more distinct non-consecutive Fibonacci numbers called Zeckendorf representation. For each positive integer n , there is a positive integer N such that:

$$n = \sum_{k=0}^N \epsilon_k F_k \quad \text{where } \epsilon_k \text{ is 0 or 1, and } \epsilon_k * \epsilon_{k+1} = 0$$

The Zeckendorf representation of a positive integer is unique because no two consecutive Fibonacci terms occur in the Zeckendorf representation to build the Fibonacci label encoding scheme.

Given a binary encoded bit-string label $N_{\text{new}} = 110101$, the length of N_{new} is firstly determined, i.e. 6 bits. The Zeckendorf representation of 6 is $5 + 1 = 1 \times 1 + 0 \times 2 + 0 \times 3 + 1 \times 5$. So, the binary string is "1001". Since the last bit in the Fibonacci coded binary string of a Zeckendorf representation is always "1", an extra "1" bit is appended to the end of the bit string to act as a delimiter, separating the length of a node label from the label itself. Thus, the binary string for the length is now "10011". The length of the label is encoded and stored before the label itself. Hence, the label N_{new} (110101) is encoded and stored using the Fibonacci label storage scheme as 10011110101.

Label size analysis

To label the nodes at each level of the XML tree, the number of bits used starts from 1, i.e. the length field. We can identify two nodes with one bit. Afterward, the number of bits increases one bit and we can identify four nodes with two bits. Likewise, the number of nodes we can encode with n bits is 2^n . So, the number of total bits used to encode nodes with 1 up to the n bits, denoted by B , is computed using the following equation:

$$B = \sum_{i=1}^n i \times 2^i = (n-1) \times 2^{n+1} + 2 \quad (1)$$

Equation 1 can be proved by induction. Let N be the maximum number of nodes to be encoded. It can be obtained from the following relation:

$$N = 2^1 + 2^2 + 2^3 + \dots + 2^n = 2^{n+1} - 2 \rightarrow n = \log_2^{N+2} - 1 \quad (2)$$

By substituting Eq. 2 in Eq. 1 we will have:

$$\begin{aligned} B &= (n - 1) \times 2^{n+1} + 2 = \left[\left(\log_2^{N+2} - 2 \right) \times (N + 2) \right] + 2 \\ &= N \log_2^{N+2} + 2 \log_2^{N+2} - 2N - 2 \end{aligned} \quad (3)$$

Consequently, to assign the unique identifiers to N nodes each with an appropriate length, i.e. at least 1 bit and at most n bits, the total storage requirement of the labeling scheme is $N \log_2^{N+2} + 2 \log_2^{N+2} - 2N - 2$ bits. Note that, this is the total storage which is needed in the worst case. However, for the average case, as the number nodes to be encoded using the FibLSS encoding increases exponentially, the growth rate in the number of bits required to encode nodes is linear [43].

Algorithm 1 provides a summarized illustration of how labels are assigned to the nodes based on the FibLSS encoding scheme.

Algorithm 1. Create SelfID

Input: an XML tree

Output: SelfID of the nodes

```

begin
    foreach level
        begin
            n=1 //n is the number of bits needed to create new SelfID
            nnode=0 //nnode denotes the number of nodes that labeled
            foreach node
                begin
                    nnode++
                    SelfID=create_SelfID (n, nnode)
                    Save SelfID
                    if nnode=2 pow n then n++
                end
            end
        end
    end

    create_SelfID (n, nnode)
    begin
        id=convert nnode to n-bit binary code // i.e. n=2 and nnode=1 then id=01
        id_length=create_Zeckendorf(n) // i.e. if n=2 its zeckendorf is 011
        return (id_length concatenate id)
    end
end

```

Updating the document

An important issue in labeling XML documents is the “updating process”. Particularly, the insertion of new nodes to the XML tree should not influence the label of the other nodes. Our proposed method avoids relabeling due to the technique used for labeling nodes when new nodes are inserted.

Inserting new nodes occurs in three general cases: inserting a node before the leftmost node, inserting a node after the rightmost node, and inserting a node between any two nodes at any position. We review each of these cases and explain how the proposed method overcomes such cases.

Case 1: Insert a node before the leftmost node

For this case, the label of the inserted node becomes the label of the leftmost node concatenated with a more “0” bit as its UpID. For example, in Fig. 2 the leftmost node in the second level has the label “2,1,0”. Thus, the label of the inserted node A would be “2,1,0.0”. Then, node B is inserted and its label will be “2,1,0.00”.

Case 2: Insert a node after the rightmost node

The label of the inserted node is the label of the rightmost node concatenated with a more “1” bit as its UpID. For example, in Fig. 3 the rightmost node has the label “3,10,1”.

Therefore, the label of the inserted node C will be “3,10,1.1”. Then, node D is inserted and its label will be “3,10,1.11”.

If E and F (Fig. 4) are inserted in the XML tree as the children of node C, their labels will become “4,1.1,0” and “4,1.1,1”, respectively.

Case 3: Insert a node between any two nodes at any position

In such cases, the size of labels of the two neighbor sibling nodes is compared. If the size of the left sibling node’s label is less than or equal to the size of the right sibling, we add a more ‘0’ bit to UpID of the right node. This means that the new node has been inserted between these nodes as its left and right siblings. Otherwise, the label of the inserted node is the label of left sibling node concatenated with a more “1” bit to its UpID.

Due to the method of assigning labels to the nodes the size of identifiers increases from left to right. So, the length of each node’s ID is smaller than the one of its right neighbors. Hence, the label size of the left node is greater than its right neighbor if UpID of left node’s label is not null.

For example, in Fig. 5 node G is inserted between nodes with labels “2,00,00” and “2,00,01”. The left sibling node has the SelfID “00” and the right sibling node has

the SelfID “01”. The length of the two labels is equal and the right node has not UpID. Hence, the label of G is the SelfID of the right node concatenated by a “0” bit as its UpID, i.e. “2,00,01.0”. Then, node H is inserted into the tree. The Labels of the left and the right neighbors of node H are “2,00,00” and “2,00,01.0”, respectively. The label of node H is formed by the right node’s UpID concatenated with a “0” bit (“2,00,01.00”) because the size of the left node’s label is less than that of the right node. Also, node I is insert between the nodes with respective labels (“2,00,01.0” and “2,00,01”). The SelfIDs of these nodes are “01.0” and “01”. Since the length of the left node’s label is greater than that of the right node, we concatenate the label of the left node with a more “1” bit to obtain the label of the inserted node, I, which is “2,00,01.01”.

Algorithm 2 summarizes the node insertion process discussed above.

Algorithm 2. Insert Node

Input: Left_SelfID and Right_SelfID

Output: New_SelfID

Begin

if (Left_SelfID is empty and Right_SelfID is not empty)

//Insert a node before the leftmost node

 New_SelfID = Right_SelfID concatenate 0

else if (Left_SelfID is not empty and Right_SelfID is empty)

//Insert a node before the rightmost node

 New_SelfID = Left_SelfID concatenate 1

else if (Left_SelfID is not empty and Right_SelfID is not empty)

 if (Length (Left_SelfID) <= Length (Right_SelfID))

 New_SelfID = Right_SelfID concatenate 0

 else

 New_SelfID = Left_SelfID concatenate 1

end

Order of labels

Primarily, the ordering between nodes is determined according to the size and the binary value of the labels. Here we only consider the SelfID for simplicity and omit the Level and the ParentID which are clear in a given context node. We label the nodes of a given level from left to right using appropriate label sizes starting from 1 bit, up to whatever is needed.

After inserting a node and updating the XML tree, we add the UpID part to the node’s label, just after its SelfID. Therefore, for insert operation, only the UpID part of the inserted nodes are changed. The following rules are used to determine the order of two nodes A and B in a given context:

- (a) Without considering UpIDs, if SelfID of A is less than B’s SelfID then A is in the left of B, e.g. $1 < 00.001$ and $1 < 1010$ and $10 < 11$.

- (b) Without considering UpIDs, if SelfID of A is equal to B's SelfID then we should compare their UpIDs. If UpID of A is less than B's UpID then A is in the left of B. Two UpIDs are compared according to the following rules:
- (1) If the UpID of a label is not null and start with 0 then it is smaller than the label that its UpID is null and if the UpID of a label starts with 1, then it is greater than the label that has no UpID.
 - (2) The lexicographical comparison of UpIDs is used if the UpIDs have the same size.
 - (3) If the length of A's UpID is less than the length of B's UpID and A's UpID is the prefix of B's, then if the first bit after the prefix string of A is 1 then B's UpID is greater than A's UpID otherwise A's UpID is the greater one.
 - (4) If the length of A's UpID is greater than B's and B's UpID is the prefix of A's UpID then, if the first bit after the prefix string of B is 1 then A's UpID is less than B's UpID otherwise A's UpID is the greater one.

For example: $0.000 < 0.00 < 0.001 < 0.0 < 0.01 < 0 < 1.00 < 1.0 < 1.01 < 1 < 00.0 < 00.01 < 00 < 01$.

Determining structural relationships

To have an efficient XML query processing, the labeling scheme should determine the structural relationships. The proposed labeling scheme can determine P–C, A–D and sibling relationships among arbitrary nodes.

P–C relationship

In an XML tree to specify P–C relationship between node A with label $\langle \text{Level}_A, \text{ParentID}_A, \text{SelfID}_A \rangle$ and node B with label $\langle \text{Level}_B, \text{ParentID}_B, \text{SelfID}_B \rangle$ it suffices to compare ParentID of a node with SelfID of the other node. It means, if $\text{ParentID}_A = \text{SelfID}_B$ and $\text{Level}_A = \text{Level}_B + 1$ then A is the child of B. For example, node A with label “2,00,01.001” is the child of node B with label “1,0,00” because $\text{ParentID}_A = \text{SelfID}_B = 00$ and $\text{Level}_A = 2 = \text{Level}_B + 1 = 1 + 1$.

A–D relationship

We can determine the A–D relationship like to the P–C relationship as a recursive function. It repeats this recursive function $|\text{Level}_A - \text{Level}_B|$ times.

Sibling relationship

In an XML tree, node A with label $\langle \text{Level}_A, \text{ParentID}_A, \text{SelfID}_A \rangle$ and node B with label $\langle \text{Level}_B, \text{ParentID}_B, \text{SelfID}_B \rangle$ should be in the same level to have the sibling relationship. That is, $\text{ParentID}_A = \text{ParentID}_B$ and $\text{Level}_A = \text{Level}_B$. For example, node A with label “2,00,01.011” is the sibling of node B with label “2,00,01.001” since $\text{ParentID}_A = \text{ParentID}_B = 00$ and $\text{Level}_A = \text{Level}_B = 2$.

Results and discussion

To evaluate the proposed method for labeling XML documents, we compare it with two leading methods, namely IBSL and P–Containment [5]. The P–Containment method has the difficulty with relabeling when inserting nodes in the XML tree. We solve this problem by applying our method to it. In this way, we implement the mapping function

Table 1 Characteristics of the datasets

Dataset	Title	Size (MB)	No. of nodes	Max. depth	No. of attributes/# usage	Max. fan-out
DS0	Xmark (0.02)	2.27	33,140	12	14/7384	4858
DS1	Orders	5	150,001	3	1/1	15,000
DS2	Xmark (0.3)	34	501,498	12	14/114,731	73,761
DS3	Lineitem	30	1,022,976	3	1/1	60,175
DS4	Treebank_e	82	2,437,666	36	1/1	56,384

Table 2 Label format of schemes

Scheme	Label format
New	Level.ParentID.SelfID
IBSL	ParentLabel.SelfLabel
NP—Containment	Start.End.ParentStart

to convert integers allocated to parameters Start, End, and Parent_Start to the binary bit string, and we call it NP—Containment.

There are four sets of tests in this performance evaluation: the first set compares the storage requirement of three schemes. The second set analyzes labeling time. The third set examines the query performance and the last set investigates update performance.

Experimental setting

We conduct the performance evaluations on a Pentium (R) Dual-Core CPU E5300 @2.60 GHz 2.60 GHz and 4.00 GB of RAM Windows 7 Professional computer. We implement all schemes using Visual C#. net 2008. To avoid the discrepancy, we run each performance test 6 times. We ignore the first run and take the average value.

Characteristics of the datasets

The datasets used in the performance evaluations are “Lineitem”, “Treebank_e”, and “Orders” that are accessible online on the internet [44]. The XMark datasets were generated using xmlgen of the XMark: Benchmark Standard for XML Database Management [45] with factors 0.02 and 0.3. Table 1 presents the characteristics of the datasets.

Storage requirement

In this section, we evaluate the storage space required for storing the labels generated by each of the labeling schemes. We save labels in a file in the formats shown in Table 2.

Figure 6 shows the total amount of storage required for labels to be generated by each of the three methods when executed on the datasets.

In the experiments, no compression is used. The storage requirement of IBSL is more than that of the other methods. This is expectable, as IBSL is a prefix-based scheme and its label size depends on the number of children of every node, i.e. the fan-out of XML documents. Regarding Table 1, the maximum fan-out of dataset Orders is 15,000.

Fig. 6 Storage requirement

Fig. 7 Largest size of labels

Therefore, it uses 15,001 bits (1 bit zero and 15,000 bits one) to signify the last child of a given node. Moreover, this scheme is based on a fix-length approach, i.e. the children of an intended node have the same label size. However, new needs maximum 17 bits for this case, where all the children of a node do not have the same label size.

Figure 7 presents the maximum length of labels generated by the three competing methods.

Fig. 8 Labeling time

The label's length is the number of bits that assigned to SelfID for the IBSL as well as New schemes, and the sum of the Start length and End length for the NP–Containment scheme. The New scheme stores the length field of SelfID and SelfID itself. Nevertheless, the size of the labels generated by New is still much smaller than that of IBSL, and outperforms NP–Containment.

Overall, regarding all the datasets, the New method has the least storage requirement, and its average label size is 19% and 63.53% of IBSL's and NP–Containment's label sizes, respectively. The maximum label size of the New scheme is 0.04% of IBSL and 33.88% of NP–Containment, on average for all the datasets in our experiments.

Labeling time

We study the time required to label a given XML document. Figure 8 shows the time required for labeling. It is the average labeling time taken from five tests we performed on each dataset. The labels are generated by applying depth-first strategy by all the three labeling schemes.

Figure 8 shows that for all the five datasets. The New scheme is faster than IBSL, especially for bigger data sets. Also, the labeling time for the NP–Containment scheme (that is, P–Containment combined with the New labeling scheme in order of contextualization) is as good as New's time. However, notice that the P–Containment method has to relabel all nodes when new nodes are inserted. IBSL manipulates longer node labels as it stores the label of the parent (hence the label of its ancestors) of an intended node in addition to SelfID. Therefore, this method requires much more time for the labeling process. Considering all the datasets, the New method generates required labels almost 12.74 times faster than IBSL.

Table 3 Evaluated queries

	Number of retrieved nodes	Query
Q1	6	//site/*
Q2	510	//person/name
Q3	510	//person[child::name]/@id
Q4	249	//person/*/city
Q5	1140	//bidder/parent::open-auction
Q6	33,139	//site/*

Query response time

For performance evaluation in updates, we investigate the time needed to process the queries. It stores labels in the tables of a relational database. To optimize queries and increase the efficiency of the update operations, we consider two table storing information about datasets. The first table includes the name of each node, its text, and label. The second table stores node attributes name and their values with the foreign key that is the primary key of the first table.

We store the labels of New in three fields including NodeLevel, ParentID, and SelfID. The primary key of the first table is SelfID along with NodeLevel. For NP-Containment, the relational table contains the name of every node, its text, Start, End, and Parent_Start values. At the update time, We use New to generate inserted nodes' labels. We implemented the IBSL scheme for which the relational table includes the name of each node, its text and the values of SelfID and ParentLabel.

We have implemented all schemes by the depth-first traversal method. We applied this evaluation on Xmark (0.02) database. The table for the elements includes 33,140 records and 7384 records for the attributes.

Table 3 shows the studied queries. The “number of retrieved nodes” is the count of records retrieved as the query response.

Figure 9 presents the results. For the New method, the time needed for queries, Q1 to Q5, is 80% of the NP-Containment time on average, but for query Q6, NP-Containment has been 1.3 times faster.

To answer query Q6, the New method should check one more condition than NP-Containment; i.e. the comparison of the levels to determine P-C relation. This additional operation justifies why the NP-Containment scheme operates faster than the New method.

In query Q6, where it retrieves all descendants for the “site” node, IBSL acts better than the other two methods because it is a full prefix-based scheme and can quickly determine the A-D relationships. Therefore, IBSL is 40.9 times faster than New. New and NP-containment use a recursive algorithm to determine A-D relationship, hence, they need more time to process such queries. For the queries Q1 to Q5, New is 15.9 and 1.3 times faster than IBSL and NP-Containment respectively, on average.

Updating time

In this part of the performance evaluation, we test the time needed to insert new nodes in various positions of the XML tree. Figure 10 illustrates the results.

Fig. 9 Queries response times

For this test, we insert 5 nodes in the leftmost place of the first level. Figure 10 shows the average time needed for the insertions as “Left_ins”. Then we insert 5 nodes in the rightmost place with an average time expressed as “Right_ins”. After that, we add 5 nodes in between the sixth and the seventh nodes in this level, with an average time denoted by “Middle_ins”.

According to the results, our method needs less time than the others. As we apply New for labeling in NP-Containment, it runs faster than IBSL for insertion. New is 2.8 and 1.4 times faster than IBSL and NP-Containment respectively, on average.

Summary and conclusions

In this paper, we provided an overview of the existing state-of-the-art methods for labeling dynamic XML data which have been investigated during the past years. Considering dynamic XML, the existing methods do not fulfill all the performance requirements at the same time. An XML labeling method should support the structural relationships among nodes as well as avoid relabeling any existing node and keep the order of nodes when new nodes are inserted into the XML tree. Moreover, scalability and efficiency are the two essential requirements to be fulfilled by a labeling method. We proposed a novel labeling method which exploits the FibLSS encoding for labeling XML documents. We discussed the efficiency of the proposed scheme and tested it considering the label size, labeling time, querying time and node insertion time. We compared this evaluation with IBSL and P-Containment which are among the leading labeling schemes in labeling the literature. Our method is scalable as it avoids relabeling and it has a linear growth for label size. Moreover, it supports

structural relationships while keeping the order of nodes. Our experimental evaluation demonstrated that the proposed method outperforms existing methods in terms of several essential requirements, especially the computational processing costs, the storage cost while keeping the order and relationships among nodes without any relabeling.

Future work

The proposed scheme is suitable and usable for both static and dynamic documents. We will apply this scheme to different labeling approaches, especially “range-based” methods that have a small storage size and high efficiency. We will also conduct more comparative studies on the efficiency of our method for a wider range of XML queries. Besides, some investigations are required for supplying more compact representations of the labels.

Authors' contributions

EK and LG conceived the research in the present study and conducted data analysis in this study. LG performed the analysis and experiments and developed the algorithm which was verified by EK. LG drafted the manuscript and EK prepared the revised manuscript and participated in the complexity analysis and the interpretation of the results. The whole work was supervised by EK. Both authors read and approved the final manuscript.

Acknowledgements

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

X. d. repository, "[online] <http://www.cs.washington.edu/research/xmldatasets>", 2015.

Funding

Not applicable.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 4 September 2018 Accepted: 7 December 2018

Published online: 19 December 2018

References

1. Fu L, Meng X. Triple code: an efficient labeling scheme for query answering in XML data. In: Web information system and application conference (WISA). 2013. p. 42–7.
2. Haw S-C, Lee C-S. Data storage practices and query processing in XML databases: a survey. *Knowl Based Syst*. 2011;24(8):1317–40.
3. Amagasa T, Yoshikawa M, Uemura S. QRS: a robust numbering scheme for XML documents. In: Proceedings 19th international conference on data engineering. 2003. p. 705–7.
4. Dietz PF. Maintaining order in a linked list. In: Proceedings of the fourteenth annual ACM symposium on theory of computing. New York: ACM; 1982. p. 122–7.
5. Li C, Ling TW, Lu J, Yu T. On reducing redundancy and improving efficiency of XML labeling schemes. In: Proceedings of the 14th ACM international conference on information and knowledge management. New York: ACM. 2005. p. 225–6.
6. Li Q, Moon B. Indexing and querying XML data for regular path expressions. *Vldb*. 2001;1:361–70.
7. Min J-K, Lee J, Chung C-W. An efficient encoding and labeling for dynamic xml data. In: International conference on database systems for advanced applications. Berlin: Springer; 2007. p. 715–26.
8. Thonangi R. A concise labeling scheme for XML data. In: COMAD. 2006. p. 4–14.
9. Yun J-H, Chung C-W. Dynamic interval-based labeling scheme for efficient XML query and update processing. *J Syst Softw*. 2008;81(1):56–70.
10. Zhang C, Naughton J, DeWitt D, Luo Q, Lohman G. On supporting containment queries in relational database management systems. In: ACM SIGMOD Record, vol 30. New York: ACM; 2001. p. 425–36.
11. Al-Jamimi HA, Barradah A, Mohammed S. Siblings labeling scheme for updating XML trees dynamically. In: International conference on computer engineering and technology. 2012. p. 21–5.
12. Almelibari A. Labelling dynamic XML documents: a group based approach. In: Doctoral dissertation, University of Sheffield; 2015.
13. Assefa BG, Ergenc B. Order based labeling scheme for dynamic XML query processing. In: International conference on availability, reliability, and security. Berlin: Springer; 2012. p. 287–301.
14. Cohen E, Kaplan H, Milo T. Labeling dynamic XML tree. *SIAM J Comput*. 2010;39(5):2048–74.
15. Duong M, Zhang Y. LSDX: a new labelling scheme for dynamically updating XML data. In: Proceedings of the 16th Australasian database conference, vol 39. New York: Australian Computer Society, Inc.; 2005. p. 185–93.
16. Duong M, Zhang Y. Dynamic labelling scheme for xml data processing. In: On the move to meaningful internet systems: OTM 2008. Berlin: Springer; 2008. p. 1183–99.
17. Ghaleb TA, Mohammed S. Novel scheme for labeling XML trees based on bits-masking and logical matching. In: 2013 world congress on computer and information technology (WCCIT), IEEE. 2013. p. 1–5.
18. Ghaleb TA, Mohammed S. A dynamic labeling scheme based on logical operators: a support for order-sensitive XML updates. *Procedia Computer Sci*. 2015;57:1211–8.
19. Li C, Ling TW. An improved prefix labeling scheme: a binary string approach for dynamic ordered XML. In: Database systems for advanced applications. Berlin: Springer; 2005. p. 125–37.
20. Liu J, Ma Z, Qv Q. Dynamically querying possibilistic XML data. *Inf Sci*. 2014;261:70–88.
21. Lu J, Ling TW, Chan C-Y, Chen T. From region encoding to extended dewey: on efficient processing of XML twig pattern matching. In: Proceedings of the 31st international conference on very large data bases. *Vldb Endowment*; 2005. p. 193–204.
22. Lu J, Meng X, Ling TW. Indexing and querying XML using extended Dewey labeling scheme. *Data Knowl Eng*. 2011;70(1):35–59.
23. O'Neil P, O'Neil E, Pal S, Cseri I, Schaller G, Westbury N. ORDPATHs: insert-friendly XML node labels. In: Proceedings of the 2004 ACM SIGMOD international conference on management of data. New York: ACM; 2004. p. 903–8.
24. Soltan S, Zarnani A, Ali Mohammadzadeh R, Rahgozar M. IFDewey: a new insert-friendly labeling schema for XML data. In: World academy of science, engineering and technology, international journal of computer, electrical, automation, control and information engineering 2. 2008. p. 203–5.
25. Tatarinov I, Viglas SD, Beyer K, Shanmugasundaram J, Shekita E, Zhang C. Storing and querying ordered XML using a relational database system. In: Proceedings of the 2002 ACM SIGMOD international conference on management of data. New York: ACM; 2002. p. 204–15.
26. Liu J, Zhang X. Dynamic labeling scheme for XML updates. *Knowl Based Syst*. 2016;106:135–49.
27. Kha DD, Yoshikawa M, Uemura S. A structural numbering scheme for XML data. In: International conference on extending database technology. Berlin: Springer; 2002. p. 91–108.
28. Ko H-K, Lee S. A binary string approach for updates in dynamic ordered xml data. *IEEE Trans Knowl Data Eng*. 2010;22(4):602–7.

29. Kobayashi K, Liang W, Kobayashi D, Watanabe A, Yokota H. VLEI code: an efficient labeling method for handling XML documents in an RDB. In: Proceedings 21st international conference on data engineering, 2005. ICDE 2005. p. 386–7.
30. Li C, Ling TW. QED: a novel quaternary encoding to completely avoid re-labeling in XML updates. In: Proceedings of the 14th ACM international conference on information and knowledge management. New York: ACM; 2005. p. 501–8.
31. Li C, Ling TW, Hu M. Efficient updates in dynamic XML data: from binary string to quaternary string. In: The VLDB journal—the international journal on very large data bases, vol. 17, no. 3. 2008. p. 573–601.
32. O'Connor M, Roantree M. SCOOTER: a compact and scalable dynamic labeling scheme for XML updates. In: Database and expert systems applications. Berlin: Springer; 2012. p. 26–40.
33. Weigel F, Schulz KU, Meuss H. The BIRD numbering scheme for XML and tree databases—deciding and reconstructing tree relations using efficient arithmetic operations. In: Database and XML technologies. Berlin: Springer; 2005. p. 49–67.
34. Wu X, Lee ML, Hsu W. A prime number labeling scheme for dynamic ordered XML trees. In: 2004 proceedings 20th international conference on data engineering, 2004. p. 66–78.
35. Qin Z, Tang Y, Tang F, Xiao J, Huang C, Xu H. Efficient XML query and update processing using a novel prime-based middle fraction labeling scheme. *China Commun*. 2017;14(3):145–57.
36. Jiang Y, He X, Lin F, Jia W. An encoding and labeling scheme based on continued fraction for dynamic XML. *J Softw*. 2011;6(10):2043–9.
37. Mirabi M, Ibrahim H, Udzir NI, Mamat A. An encoding scheme based on fractional number for querying and updating XML data. *J Syst Softw*. 2012;85(8):1831–51.
38. Ni Y-F, Fan Y-C, Tan X-C, Cui J, Wang X-L. Numeric-based XML labeling schema by generalized dynamic method. *J Shanghai Jiaotong University (Science)*. 2012;17:203–8.
39. Noor Ea Thahasin S, Jayanthi P. Vector based labeling method for dynamic XML documents. In: 2013 international conference on information communication and embedded systems (ICICES). 2013. p. 217–21.
40. Xu L, Bao Z, Ling TW. A dynamic labeling scheme using vectors. In: Database and expert systems applications. Berlin: Springer; 2007. p. 130–40.
41. Xu L, Ling TW, Wu H, Bao Z. DDE: from dewey to a fully dynamic XML labeling scheme. In: Proceedings of the 2009 ACM SIGMOD international conference on management of data. 2009. p. 719–30.
42. Härder T, Haustein M, Mathis C, Wagner M. Node labeling schemes for dynamic XML documents reconsidered. *Data Knowl Eng*. 2007;60(1):126–49.
43. O'Connor MF, Roantree M. FibLSS: a scalable label storage scheme for dynamic XML updates. In: East European conference on advances in databases and information systems. Berlin: Springer; 2013. p. 218–31.
44. X. d. repository. <http://www.cs.washington.edu/research/xmldatasets>. 2015.
45. Schmidt A, Waas F, Kersten M, Carey MJ, Manolescu I, Busse R. XMark: a benchmark for XML data management. In: Proceedings of the 28th international conference on very large data bases: VLDB Endowment. 2002. p. 974–85.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com