Algorithms for Indexing Dynamic XML Data

Jeang-Kuo Chen* Jen-Peng Hsu

Department of Information Management
Chaoyang University of Technology

jkchen@cyut.edu.tw s9314630@cyut.edu.tw

Abstract

Some approaches such as path indexing, labeling, and numbering scheme have been proposed in order to facilitate query of XML data. The indexes derived by these approaches must be rebuilt if XML data is updated. The method LSDX (Labeling Scheme for Dynamic XML data) can dynamically add new labels without updating existing old labels. However, there are two defects in LSDX. The first defect is that the nodes within the same level can be labeled no more than twenty five when LSDX builds the XML index tree. The second defect occurs when updating the index tree. If LSDX inserts a new node C between two nodes A, B and then inserts a new node D between the two nodes A, C, the label of the node D cannot be coded because of the label coding strategy of LSDX. In this paper, we propose algorithms with new label coding rules to solve the two defects of LSDX. With our method, unlimited nodes within the same level can be labeled when building the index. When updating the index tree, we use a new label coding method for new-inserted nodes. If the number of new-inserted nodes within the same level is not more than five, the label length of these nodes will not increase immediately.

Keywords: XML, Algorithm, Indexing, Labeling

1. Introduction

In recent years, the eXtensible Mark-up Language (XML) has become a popular data medium for data exchange on the web. In order to speed up query for large data, one of the most efficient method is to build a good index. A well-constructed index will allow a query to bypass the need of scanning the entire table for results [6]. A unique label is assigned to each node in the index tree in such a way that it can clearly show the ancestor-descendant or sibling relationship between any two given nodes [4]. To establish a fast search index, several methods have been proposed such as path indexing, labeling, and numbering scheme. The path indexing [1,3,5] represents an XML document data by a tree structure. Each node in the index tree is labeled with the path from the root to the node. The labeling scheme [2,7,8,10] generates labels by using numbering scheme which can be classified into region-based [8] and prefix-based [2,7,10] numbering schemes. Although these techniques can build indices fast, but they have a common defect. Namely, when the original XML data is updated, the indices created by these techniques need to be rebuild. It takes time and reduces system throughput. To solve this problem, a new labeling scheme, Labeling Scheme for Dynamic XML data [4] (LSDX), is proposed. LSDX is a persistent labeling scheme to label nodes in the index tree for XML data. If XML data is updated frequently, LSDX can generate labels for the new-inserted nodes without changing the old labels of the existent nodes. The label generated by LSDX can present the depth of the tree because the label of each node specifies the level of each node in the index tree.

However, we found that LSDX has two significant defects because of the limitation of the labeling rules. The first one occurs when building the index tree. The nodes within the same level can be labeled no more than twenty five. This defect makes LSDX impractical. The second defect occurs when updating the index tree. If LSDX continuously inserts two new nodes towards left between two old nodes. The label of the second new-inserted node cannot be coded. This defect makes LSDX fail to work continuously. In this paper, we propose algorithms with new label coding rules to solve the significant defects of LSDX. To solve the first defect, we extend the label building rule of the index tree by labeling the node with a formula. Theoretically, infinite number of labels can be coded when building the index tree with our rule. To solve the second defect, we use a new label coding rule for the new-inserted nodes. If the number of new-inserted nodes within the same level is not more than five, the label length of these nodes will not increase immediately. Even though the label length of new-inserted node increases, our method still can continue to label new-inserted nodes without the problem of unable to labeling. When a new-inserted

node n is inserted between two nodes, the label of node n is based on the alphabet string abstracted from the partial code of n's previous sibling plusing some characters. Each character is the middle alphabet of a certain alphabet interval. Anyway, the label of the new-inserted node is greater than the label of its previous sibling and smaller than the label of its next sibling in the alphabetical order.

As shown in Fig. 1, data in an XML document can be commonly modeled into a tree structure, where nodes represent elements, attributes, or text data. The tree is called XML data tree. An index tree for an XML document is similar to the XML data tree but containing some extra information in each node such as label string, index tag name, etc.

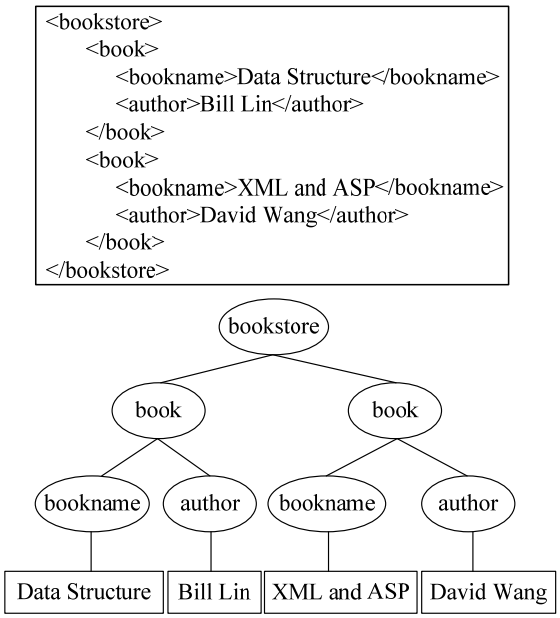

Fig. 1: An example of an XML document and its data tree.

2. The algorithms for building and updating index tree

2.1. The node structure of an XML index tree

The node structure in an XML index tree is shown in Fig. 2. Each node contains four fields. The *label* field, string type, is used to represent the relationship among nodes. The *tag_name* field, a pointer, is used to point to the position of an XML element tag. The *ele_content* field, a pointer, is used to point to the position of an XML element content. If element content is not practical data, then this field is null. The $child_i(i=1,...,n)$ fields are pointers to individual child nodes.

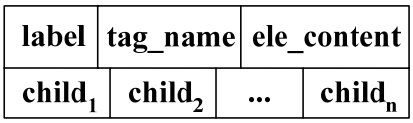

Fig. 2: The node structure of an XML index tree.

2.2. The coding rule of node label for building an index tree

When LSDX builds an index tree, the coding rule of node label is as follows. Given a node v with n child nodes:u_1, u_2, u_3, …, u_n, the label for u_1 is a combination of its level + the code of its parent node + '.' + 'b'. The label for u_2 is a combination of its level + the code of its parent node + '.' + 'c'. The labels of the rest child nodes can be coded with the same concept. We find that there are two problems in this rule. First, the label of the root is not defined. Second, only 25 nodes at most can be labeled within the same level. Therefore, we propose a new rule for building an index tree as follows.

- The label of the root node is defined as "0a". Level number starts from '0' which represents the level of the root node. The 'a' alphabet represents the code of the root node.
- When a non-root node N is the α^{th} node from left to right at its level and α smaller than or equal to 25. The label for N is a combination of its level + the code of its parent node + '.' + @, where @ is the α^{th} alphabet of the interval [b, c, …, z].
- When a non-root node N is the β^{th} node from left to right at its level and β greater than or equal to 26. The label for N is a combination of its level + the code of its parent node + '.' + @, where @= $\sum_{1}^{i=\beta-24} z^i$, i is the number of z, i is greater than or equal to 2.

The algorithm of the label coding of node for building an index tree is as follows.

Algorithm Build_Node_Labeling(T)
Input: node_pointer T;
Begin
  char alphabet[25] = {b,c,d,…,z};

  T.label ← "0a";
  for each child i of T, do
    T.child[i].label ← the level of i + T's parent code + '.';
    if i <= the 25th member of nodes in the level of i, then
      T.child[i].label ← T.child[i].label + alphabet[i];

```
    else
      T.child[i].label ← T.child[i].label + Σ_1^{i-24} z ;
    end-if;
  end-for;
  for each child i of T, do
      Build_Node_Labeling(T.child[i]);
  end for;
End.
```

2.3. The coding rule of new-inserted node label for updating an index tree

When updating the XML index tree, if LSDX inserts a new node C between two old nodes A, B and then inserts again a new node D between the two nodes A, C, LSDX cannot assign an appropriate label for node D due to the coding rule of LSDX. The label of node D must be greater than the label of its left sibling node A and smaller than the label of its right sibling node C in alphabetical order. We can solve this problem by a new coding rule that does not increase the label length of a new-inserted node as possible. If the number of new-inserted nodes between any two old nodes is not more than five, the label length of any new-inserted node will not increase. Our coding rule for the label of the new-inserted node is as follows.

- Case1: If there is no node standing before the place where the new-inserted node M is inserted, then the label of node M is obtained by abstracting the label of the node standing after M and appending an 'a' after '.'.
- Case2: If a new-inserted node M is inserted between two nodes, the label of node M is created as follows. (a)Abstract all characters which contains '.' and before '.' from the label of M's left sibling node and append one character x_1 after '.'. This character x_1 is computed as follows. The first characters, y_1 and z_1, after '.' are abstracted from the labels of M's left sibling node and M's right sibling node, respectively. According to the sequence number of the two characters in alphabetical order, x_1 is obtained by computing the middle character between y_1 and z_1. Then check the label of M whether it is greater than the label of its left sibling node and smaller than the label of its right sibling node. If the label of M conforms to this condition, then the coding of M's label is finished. (b)If the label of M does not conform to the condition, then we append anther character x_2 to the end of M's label. This character x_2 is computed as follows. The second characters, y_2 and z_2, after '.' are abstracted from the labels of M's left sibling node and M's right sibling node, respectively. According to the sequence number of the two characters in alphabetical order, x_2 is obtained by computing the middle character between y_2 and z_2. Then check the label of M whether it is greater than the label of its left sibling node and smaller than the label of its right sibling node. If the label of M conforms to this condition, the code of M's label is finished. If there is no second character after '.' in the label of M's left sibling node or M's right sibling node, then we use the next label of the label for M's left sibling node or the previous label of the label for M's right sibling node. (c)If the label of M still cannot conform to the condition, then we append again another character to the end of M's label repeatedly until M's label conforms to the condition.
- Case3: If there is one node standing before the place where the new-inserted node M is inserted and no node standing after M, then the label of node M abstracts the label of M's left sibling node. If the last character of the obtained label is not 'z', then replace it by the 'z' character. If the last character is 'z', then a 'z' character is appended to the end of M's label.

The algorithm of the label coding of node when inserting nodes into the XML index tree is as follows.

Algorithm Adding_Node_Labeling(T, M)

Input: node_pointer T;
node_pointer M;

Begin

// Let new-inserted node as M, M_l is the left sibling node of M, M_r is the right sibling node of M. //

// Let the label of a node is consisted of three parts: (1)C_{dl}: It's a coding string of before '.'.(2)'.'(3)C_{dr}: It's a coding string of after '.'. //

C_{dl}[1,n] // It's used to store C_{dl} of the label with a array. //

C_{dr}[1,n] // It's used to store C_{dr} of the label with a array. //

// atoi( ): It's used to transform an alphabet into the number in the alphabetical order. //

// itoa( ): It's used to transform the number in the alphabetical order into an alphabet. //

// Find an appropriate place to put M into T. //

case1: if there's no M_l standing before M, then
$M.label \leftarrow M_r.C_{dl} + '.' + 'a' + M_r.C_{dr}$;
end-if;

case2: if M is standing between M_l and M_r, then
$M.label \leftarrow M_l.C_{dl} + '.' + itoa\left(\left\lfloor \frac{atoi(M_l.C_{dr}[1]) + atoi(M_r.C_{dr}[1])}{2} \right\rfloor\right)$;
end-if;

if M_l.label<M.label<M_r.label, then return M.label;
else
for i←2 to n
begin
M.label←M.label+ itoa($\lfloor \frac{atoi(M_l.C_{dr}[i]) + atoi(M_r.C_{dr}[i])}{2} \rfloor$);
if M_l.label<M.label<M_r.label, then
return M.label;
end-if;
end-for;
end-if;

case3: if there's M_l before M and no M_r after M, then
if the last character of M_l's label <>'z', then M.label←$M_l.C_{dl}$+'.'+$M_l.C_{dr}$ (replaced the last character with 'z');
else
M.label←M_l.label+'z';
end-if;
end-if;
End.

3. Conclusion

In this paper, we propose two algorithms with two new rules to build and update an index tree for XML data. Our method overcomes the restriction of labeling limited number of nodes. The algorithms can label infinite number of nodes in the same level. Apply our method, any new-inserted node can be labeled appropriately in any situation when updating the index tree. If the number of new-inserted nodes in the same level is not more than five, the label length will not increase. Even the label length of new-inserted node increases, our method still can work correctly without any error happenness. Finally, the advantages of our method are described as follows. It is suitable for the static or dynamic index of XML data. The length of the label keeps as short as possible to facilitate label coding of nodes.

4. References

[1] G. Amato, F. Debole, F. Rabitti, and P. Zezula, "Yet Another Path Index for XML Searching," *Proc. Of the 7th European Conference on Research and Advanced Technology for Digital Libraries*, pp. 176-187, 2003.

[2] E. Cohen, H. Kaplan, and T. Milo, "Labeling dynamic XML Trees," *Proc. Of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 271-281, 2002.

[3] F. B. Cooper, N. Sample, J. M. Franklin, R. G. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," *Proc. Of the 27th International Conference on VLDB*, pp. 341-350, 2001.

[4] M. Duong and Y. Zhang, "LSDX: A New Labelling Schema Dynamically Updating XML Data," *Proc. Of the 16th Australasian Conference on Database technologies*, pp. 185-193, 2005.

[5] T. Grust, "Accelerating XPath Location Steps," *Proc. Of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 109-120, 2002.

[6] http://techrepublic.com.com/5100-10878_11-5146588.html?tag=search.

[7] http://www.math.tau.ac.il/~haimk/papers/comparison.ps.

[8] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," *Proc. Of the 27th International Conference on VLDB*, pp. 361-370, 2001.

[9] J. Lu and W. T. Ling, "Labeling and Querying Dynamic XML Trees," *Proc. Of the 6th Asia Pacific Web Conference*, pp. 180-189, 2004.

[10] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and Querying Ordered XML Using a Relational Database System," *Proc. Of the ACM SIGMOD international conference on Management of data*, pp. 204-215, 2002.

[11] W. Wang, H. Jiang, H. Lu, and X. J. Yu, "PBiTree Coding and Efficient Processing of containment Joins," *Proc. Of the 19th International Conference on Data Engineering*, pp. 391-402, 2003.

[12] X. J. Yu, D. Luo, X. Meng, and H. Lu, "Dynamically Updating XML Data: Numbering Scheme Revisited," *World Wide Web: Internet and Web Information System*, Vol. 8, Issue: 1, pp. 5-26, 2005.