IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 22, 2020, accepted September 13, 2020, date of publication September 21, 2020, date of current version October 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3025566

UCIS-X: An Updatable Compact Indexing Scheme for Efficient Extensible Markup Language Document Updating and Query Evaluation

WEN-CHIAO HSU[1] **AND I-EN LIAO**[2]

[1]Department of Information Management, National Taichung University of Science and Technology, Taichung 404, Taiwan
[2]Taiwan Information Security Center at NCHU, Department of Computer Science and Engineering, National Chung Hsing University, Taichung 402, Taiwan

Corresponding author: Wen-Chiao Hsu (chiaohsu@nutc.edu.tw)

This work was supported in part by the Taiwan Information Security Center at National Chung Hsing University (TWISC@NCHU), and in part by the Ministry of Science and Technology, Taiwan, under Grant MOST 108-2218-E-005-016 and Grant MOST 109-2218-E-005-005.

ABSTRACT One of the difficulties faced when using XML as the data storage structure is query inefficiency. Therefore, various indexing methods have been proposed. When designing indexing methods, the first step is to choose the labeling method. Some labeling methods can work well; however, if they cannot effectively support update operations, their use is subject to considerable limitations. Most of the update-friendly labeling methods proposed in the literature assign a unique label to each node in XML and provide an expandable mechanism for future insertion. However, they encounter some difficulties, such as increasing the index space, more difficulty in evaluating the relationships between nodes, and increasing the complexity of labels. In this paper, we introduce a novel update-friendly labeling scheme called branch map, which records the correspondence between parent and child nodes instead of assigning a label to each node. The space required for the index is reduced considerably. More importantly, the branch map can maintain the profile as if it was encoded initially, even after being frequently updated. This paper also proposes a compact indexing scheme called UCIS-X. Experimental results indicate that UCIS-X performs well in terms of index size, query, and update efficiency.

INDEX TERMS Branch map, update-friendly labeling scheme, XML indexing, XML update operations.

I. INTRODUCTION

Extensible markup language (XML) is increasingly used for data exchange and transfer [1], [2], such as in electronic publishing, web services, e-business, and search engines. In addition to the standard format for exchanging data, XML is one of the NoSQL database models [3], [4] and a new language for specifying video games, called the XML-based video game description language (XVGDL) [5]. Because of its high availability and scalability, XML has rapidly gained attention [6]. XML documents are normally stored as plain text files. Therefore, it is important to identify an efficient and easy means of managing data in XML format. There have been many research studies regarding the issues of efficiently storing and querying XML data, such as indexing [7]–[14] or effective query evaluation [15]–[20]. Compared to indexing and query evaluation, the topic of updating both XML documents and indexes [21]–[26] has received much less attention from the research community [27]. The natural and most convenient means to update XML documents is to simply edit the text files. However, efficient query evaluation algorithms require that XML documents be indexed. Typically, every element and attribute of an XML document is given a unique identifier that is also recorded in the index and is used as a key during query evaluation. Most proposed indexing schemes assume a static environment in which there is no update of XML documents. Therefore, these indexing methods involve static labeling schemes. This restriction obviously limits the application scenarios of the XML indexing mechanism, as most real-world data change over time. Because the index must be consistent with the document, expensive index rebuilding or identifier reassigning is necessary when the original document is updated. Thus, an efficient indexing technique is required to provide a mechanism for the synchronized updating of XML documents.

The associate editor coordinating the review of this manuscript and approving it for publication was Juan A. Lara.

The key to an updatable index depends on how the elements and attributes are labeled. The literature on synchronizing data in XML and the index has mainly emphasized the labeling method. When modifying XML, the labels must be updated accordingly, e.g., the relative indexes [25]. Two common labeling schemes for XML are region labeling and prefix labeling. Both schemes have their advantages and disadvantages, as discussed later. Because no known labeling method can effectively address the following issues, designing an updatable indexing method is challenging.

(1) The label size may increase to support an updatable mechanism.
(2) The query cost may increase because the evaluation becomes more complex.
(3) The length of the labels may increase after multiple XML updates.
(4) The ability to maintain the initial labeling state may be weak.

To address these issues, we present an update-friendly labeling scheme, i.e., branch map, and a compact indexing structure, i.e., UCIS-X. Instead of assigning labels to elements, branch map records the matching map between parent and child elements. It can therefore shorten the label length and reduce storage. In addition, branch map can extract information and update operations efficiently. UCIS-X is an improved version of CIS-X [13]. It adopts a branch map that can support updating operations on XML documents. The experimental results indicate that UCIS-X outperforms many other methods in terms of the index construction cost, query evaluation performance, and label maintenance costs.

The remainder of this paper is organized as follows. Section II explores related studies on labeling schemes, indexing, and the query evaluation of XML. Section III introduces the proposed branch map labeling scheme. Section IV describes the UCIS-X index structure as well as how UCIS-X can effectively support query and update operations. Section V presents our experimental results, and Section VI concludes this paper with some directions for future work.

II. RELATED WORK

To fully evolve XML into a data representation and exchange standard, it is important to provide an efficient means to address not only query evaluation but also update management. To speed up queries, the indexing of documents is a good solution. To handle the hierarchical tree model of XML data, a suitable indexing method is required to extract and reconstruct XML document structural information, such as the tag names of elements, containment relationship, sibling order, and depth of the XML data tree. The labeling scheme used by the indexing method is the most important basis for recording the document structure. To maintain the correctness of the query results, the labels should be updated when the original XML document has been updated. The portion to be relabeled could be very large if the labeling method is not update-friendly, making the update processes quite inefficient. In this section, we briefly review some related studies, such as basic labeling schemes, updateable labeling methods, and how to carry out indexing and evaluate queries.

A. BASIC LABELING SCHEMES

An XML document can be represented by an XML data tree [28]. Before indexing, each node of the XML data tree is assigned a label to record its structural information. Two commonly used labeling schemes for trees are the region labeling scheme and prefix labeling scheme [24]. The region labeling scheme [12], [22] exploits the properties of tree traversal to maintain the node order and determine various structural relationships among nodes. Typically, each node in the tree is encoded with a triple (s_n, e_n, d_n), where s_n is the serial number of node n derived from a depth-first traversal of the data tree, e_n is the serial number after visiting all child nodes of n, and d_n is the depth of n. In region encoding, a node u is an ancestor of a node v if $s_u < s_v < e_u$. Moreover, u is the parent of v if $d_v = d_u + 1$. The region-labeling scheme is suitable for most query evaluation methods. However, when applied to an update mechanism, a significant number of nodes may need to be relabeled when a new node is inserted in an intermediate position of the tree. For example, as shown in Fig. 1(a), when a new node is inserted, the gray nodes must be relabeled.

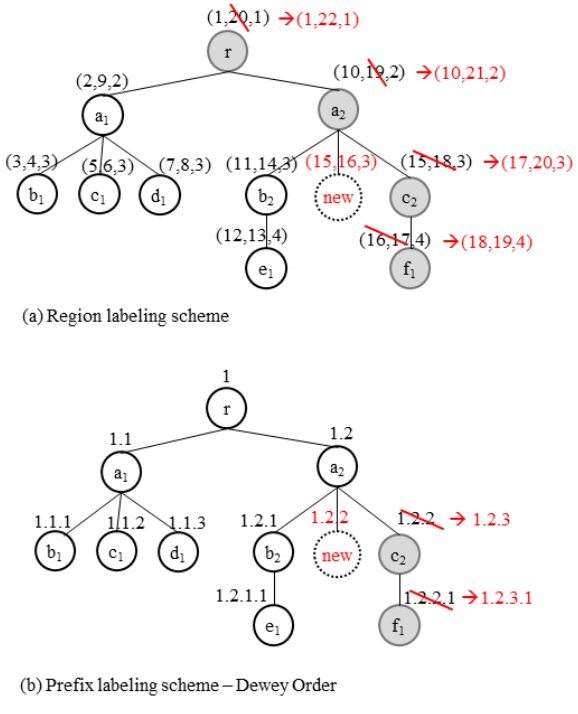


FIGURE 1. Examples of relabeling a portion of an XML document.

The Dewey order [29] is a typical example of a prefix labeling scheme. If node v is the n^{th} child of node u, then $Dewey(v)=Dewey(u)+"."+n$, where the root is always set to 1. The Dewey label of each node contains the labels of all its ancestors. Thus, the relationship between two nodes

can be deduced by comparing their labels. For example, the ancestors of node u labeled "1.3.1.5" are nodes labeled "1.3.1," "1.3," and "1." Because the prefix labeling scheme allows each node to inherit its parent's label as the prefix of its own label, the portion to be relabeled is usually smaller than when applying the region labeling scheme while inserting new nodes. An example is shown in Fig. 1(b). However, it is still not an update-friendly method. If the new node is to be inserted as the first child of the root, the entire tree except the root must be relabeled. Moreover, the results presented establish that any immutable labeling scheme requires $\Omega(N)$ bits per label, where N is the size of the document, thus incurring a high storage overhead. In addition, using a prefix labeling scheme is less efficient than using a region labeling scheme because determining the relationship between two elements using a prefix comparison is slower than using a simple integer comparison.

B. UPDATEABLE LABELING SCHEMES

The region-based labeling methods represent a global order encoding method [29], which is more unfriendly toward updating than prefix-based labeling methods. Many updatable labeling schemes have been proposed and are prefix-based [19], [23], [25], [30], [31]. The strategies used to support updates mostly preserve the space available for labeling or extend a label. In this paper, we examine three updatable labeling methods: ORDPATH [32], used in the latest versions of Microsoft®SQL Server™, and DFPD [31] and DPLS [23], two recently proposed methods.

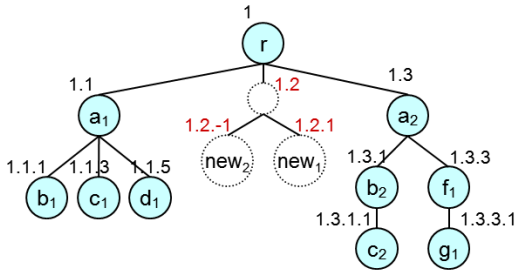


FIGURE 2. Example of insertions in ORDPATH.

ORDPATH [32], which is based on the Dewey order numbering scheme, permits the insertion of new nodes into arbitrary positions in a tree without the need to relabel existing nodes. In ORDPATH, only positive odd numbers are assigned during the initial labeling, and even numbers and negative integers are reserved for later insertions. Fig. 2 illustrates an example of ORDPATH, where two new nodes are inserted. The first new node is inserted between nodes 1.1 and 1.3. The even number 2 is used because $2 = (1 + 3)/2$. Assume that there is a virtual node labeled 1.2; then, the first new node is labeled 1.2.1 as a child of 1.2 so that the last number is an odd number. The second new node is inserted to the left of the first new node (i.e., 1.2.1) and is labeled 1.2.-1. The level or depth of each node in the tree can be determined by counting the number of odd component values in the label. For example, 3.5.6.2.1 is a child of 3.5 and a grandchild of 3.

The limitations of ORDPATH result from the variable length labeling scheme employed in conjunction with the waste of one-half the total number of nodes by virtue of labels ending in odd numbers. This can result in increased storage costs in the case of frequent updates as well as expensive comparative label evaluations between sibling nodes of varying lengths. Furthermore, the ORDPATH labeling scheme cannot completely avoid the relabeling of existing nodes because of the overflow problem.

To summarize, the characteristics of ORDPATH are as follows:

- A variable-interval labeling scheme is used for nodes of the same depth.
- It uses odd numbers for the initial labeling and saves even numbers for future expansion.
- The labels of deleted nodes can be reused.
- No relabeling is required when updating.
- A query evaluation can be more complicated because of the different lengths of the labels.
- The design of the virtual nodes requires larger memory space for labels.

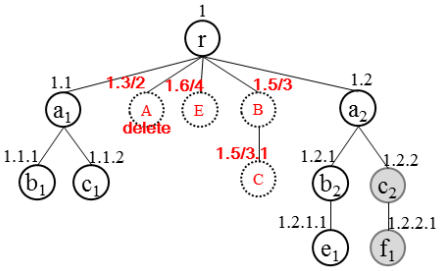


FIGURE 3. Example of updating in DFPD.

DFPD [31], a dynamic floating-point Dewey labeling scheme, is an insertion-friendly labeling method that efficiently supports dynamic queries. The initial labeling method is consistent with the Dewey order. To avoid relabeling existing nodes, the encoding of new nodes becomes somewhat complicated when inserting them. For example, Fig. 3 shows an XML data tree, initially labeled by the Dewey order, and then several changes are made in the following order: inserting A, inserting B, inserting C, deleting A, inserting E. Node A is inserted between nodes 1.1 and 1.2, and its label is 1.3/2, calculated from the following formula "1.(1 + 2) / (1 + 1)". Similarly, node B is inserted between nodes 1.3/2 and 1.2, and its label is 1.5/3, calculated from "1.(3 + 2) / (2 + 1)". Node C is a child of node B, and its label is 1.5/3.1 according to the Dewey labeling scheme. Then, node A is deleted, and node E is inserted in the same position previously occupied by node A. In this case, node E is inserted between node 1.1 and node 1.5/3, and its label is 1.6/4, calculated from "1.(1 + 5) / (1 + 3)", which differs from the label of node A. From this, it can be seen that when alternating insertions and deletions occur, DFPD is suboptimal.

The characteristics of DFPD are summarized as follows:

- A fixed-interval labeling scheme is used for nodes of the same depth.
- Integers are used for the initial labeling, and fractions are used for expansion.
- The labels of deleted nodes are not reused.
- No relabeling is required when updating.
- The query evaluation for label comparison is faster than that of ORDPATH.
- The space required for labels may grow quickly if updates are frequent.

The dynamic prefix-based labeling scheme (DPLS) [23] is also a Dewey-based labeling scheme. In fact, the solutions of the initial labeling and primitive node insertions in DPLS are consistent with those in DFPD. Unlike DFPD, which does not consider the reuse of deleted labels when new nodes are inserted, DPLS uses the reduction of a fraction operation to reuse deleted node labels. When an insertion occurs at the same position where a deletion occurred, no new label is introduced. Therefore, DPLS improves the shortcomings of DFPD by limiting the growth rate of label sizes when frequent insertions and deletions of nodes occur. Let us consider the same example as in Fig. 3 with the following update order: inserting A, inserting B, inserting C, deleting A, and inserting E. The label of node E is 1.6/4 for DFPD when the reuse of deleted labels is not considered. However, DPLS reduces the fraction of the label of node E to 1.3/2 to reuse the original label of node A. Thus, DPLS inherits the characteristics of DFPD but improves on the latter's disadvantages by reusing labels; hence, the space required for labels is reduced.

From the above examples, the space required for labels and the time for query evaluation will increase when XML documents must be updated dynamically. A noteworthy phenomenon is that the appearances of the labels are inconsistent, as would occur after initialization. Taking Fig. 3 as an example, when initialized, the label of the second child of the root was 1.2, and it changed to 1.3/2 after node E was inserted. The same situation occurs in ORDPATH, DFPD, and most updatable labeling methods. When the labels become longer and more complex, it is anticipated that complete relabeling will occur.

C. INDEXING AND QUERY EVALUATION

To accelerate query processing, various indexing and query evaluation methods have been proposed. DataGuide [11] was one of the first XML indexing models proposed. DataGuide is a structural summary scheme that merges nodes lying along the same traversal path. Subsequent literature introduced other different solutions for merging. Two common methods are path equivalence [9]–[11], [33] and bisimulation [7], [8], [34], [35]. DataGuide summarizes XML using path equivalence. On the other hand, summarizing by bisimulation involves grouping together nodes having the same set of incoming paths. In terms of node labels, earlier studies used simple encoding methods. For example,

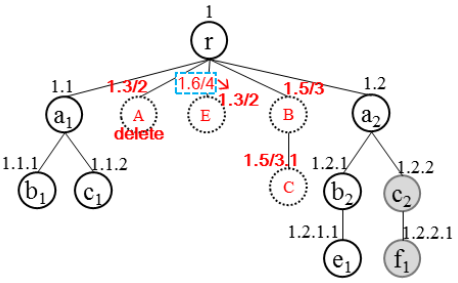
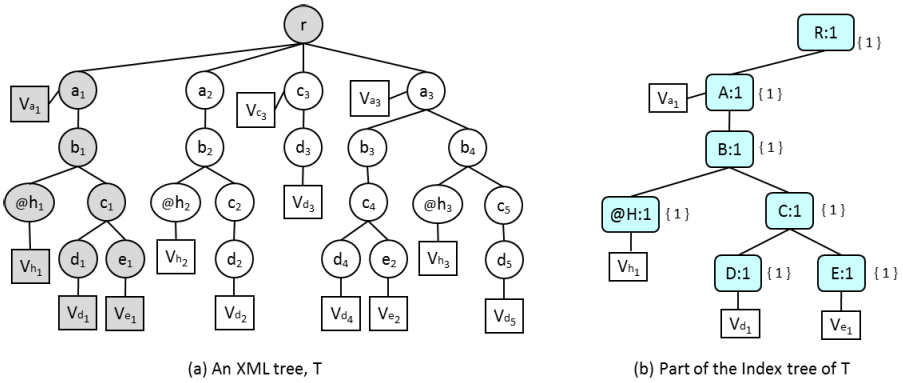


FIGURE 4. Example of updating in DPLS.

DataGuide [11], 1-index [35], and A(k)-index [34] encode an XML tree by making a breadth-first traversal and use only one number for each node. Because the structural information in the summarized structure is incomplete, it can support only a single-path query. To handle more complex queries, such as twig (or branching), different encoding methods have been proposed. As mentioned earlier, region labeling and prefix labeling schemes have been applied for this purpose. TwigX-Guide [22] labels each node of the XML using a region-labeling scheme and then constructs a DataGuide as an index. With more structural information recorded in the label, a twig query can be handled. PCIM [12] and NCIM [36] also make good use of the advantages of encoding methods combined with a structural summary scheme to support twig queries efficiently.

In addition to index models, many query evaluation algorithms have been developed in recent years to address complex queries. Basic XML queries include single-Path and Twig-Path queries [37]. Most of the existing query evaluation methods encode the nodes of the XML map using a regional labeling scheme and then simply cluster the labels using the same tag name. Then, they use their proposed query evaluation algorithms to effectively find the matched results. Both Structural Join [16] and TwigStack [17] are two-phase algorithms, which break down a twig query into several single paths that are evaluated separately in the first phase, with the final result being produced by merging the results of the single paths in the second phase. Twig^{2}Stack [18] and TwigList [20] are one-phase algorithms. To eliminate the merge costs incurred in the second phase, Twig^{2}Stack and TwigList use hierarchical stacks and a set of lists, respectively, to store the results. Most query evaluation algorithms produce some intermediate results during processing. Some of them are not part of the final result. DGReLab$^+$ [38], a path query processing technique, was proposed to avoid buffering irrelevant results before producing the final results. The evaluation results showed that DGReLab$^+$ outperformed TwigStack and QTwig [39] in query processing performance.

Some studies combine the structural summary scheme and query evaluation algorithm to provide better query processing, such as TwigX-Guide [22], CIS-X [13], and MatchQTP [40]. TwigX-Guide constructs the DataGuide and converts it into a DG index table, which is used to evaluate



(a) An XML tree, T (b) Part of the Index tree of T

FIGURE 5. Example of branch map using encoding rules 1, 2, and 5.

input queries using TwigStack. TwigX-Guide attempts to combine the advantages of these two schemes. Although the speed of query processing for a single path in the first phase is fast, the efficiency is less optimal in the second (merging) phase. The shortcomings of TwigStack, which produces useless intermediate results and requires an expensive merging phase, have not been resolved. For more details on the challenges faced when applying structural summary methods and query evaluation methods, please refer to our previous work [13]. A good index structure must reduce the costs associated with time and space and must support complex queries. CIS-X performs well regarding these aspects and has the following characteristics:

- In terms of labeling, CIS-X uses only one integer to label each node by counting the numbers in the preorder traversal of XML. The effect is that the space required for labels is reduced.
- In terms of indexing, CIS-X adopts the structural summary method to minimize the required size and stores the index in hash-based tables for rapid access.
- In terms of query processing, CIS-X uses the TwigList algorithm. Unlike TwigList, which runs its processes directly on XML, CIS-X processes the index. Therefore, the additional cost of useless intermediate comparisons associated with TwigList is very low in CIS-X.

Although CIS-X provides a good solution to speed up XML queries, it is not an update-friendly method. Therefore, based on CIS-X, this paper proposes the branch map labeling scheme to support the demands associated with updates.

MatchQTP is a twig pattern matching algorithm with an indexing technique, called RLP-Index, and an XML node labeling scheme, called RLPScheme. RLP (root-to-leaf labeled path) records the sequence of tags from root to leaf, such as $/tag_0/tag_1/\ldots/tag_n$, where tag_0 is the tag of the root and tag_n is a leaf node. A binary ID is assigned to each RLP, which can be used to compute the ancestor nodes of a node. RLP-Index also adopts the principle of structural merging of DataGuide, which minimizes the storage space utilization and query processing time of MatchQTP.

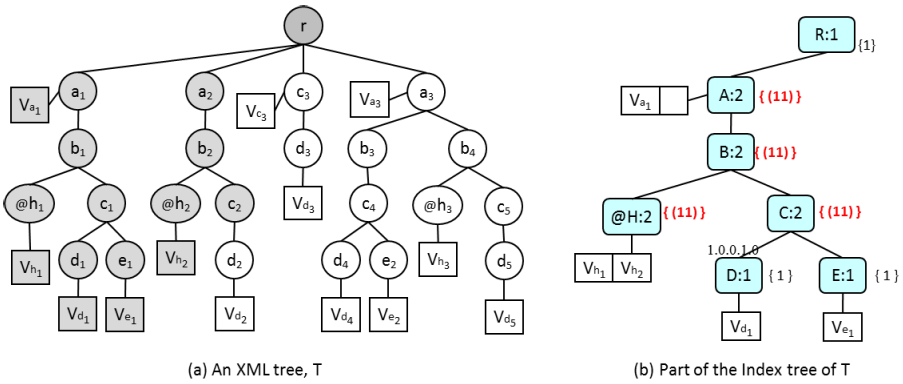
III. BRANCH MAP LABELING SCHEME

In this section, we propose a novel update-friendly labeling scheme called branch map. Before describing branch map, let us consider the purpose of encoding, which is to retain the information of the original XML structure in the index. Therefore, as long as this purpose can be achieved, it is not necessary for each node to have a unique label. The branch map labeling scheme was designed based on this idea by recording the corresponding map of the nodes. A branch map is suitable for the structural summary indexing scheme and is generated during the index construction phase.

The XML document is presented as an original "XML tree" (Fig. 5(a)). During the indexing construction phase, the XML tree is traversed in a specified order, and each node is visited twice. The nodes with "path equivalence" are summarized to generate a summarized "index tree" (Fig. 5(b)) as well as the branch map of each "index node." The encoding rules of the branch map method are as follows:

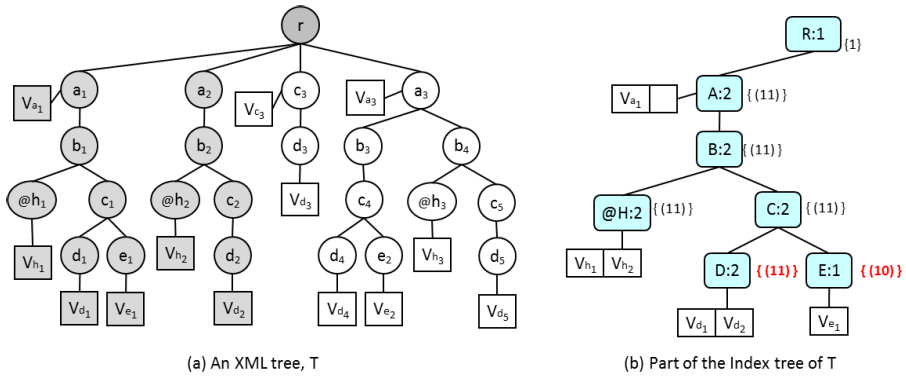
1. The branch map of the root is "1".
2. While first visiting node *n* of the XML tree, if there is no index node with the same travel path, create a new index node N. The branch map length of N is equal to the length of N's parent, where the first length-1 bits are "0" and the last bit is "1".
3. While first visiting node *n* of the XML tree, if there is an index node N with the same travel path, append a "1" after the branch map of N and group the bits with the same parent by using brackets (i.e.,"(" and ")").
4. During the second visit to node *n* of the XML tree, determine if each child of index node N has been visited in the same round. For each unvisited child N' of index node N, append a "0" after the branch map of N', and group bits with the same parent using the "(" and ")" notation.
5. For plain text nodes lying in the same travel path, group the values of attributes according to the order of travel. If there is no such path, use an empty value to maintain the position.

Fig. 5(a) shows an original XML tree, in which element nodes and attribute nodes are represented by circles and the



(a) An XML tree, T (b) Part of the Index tree of T

FIGURE 6. Example of branch map using encoding rule 3.



(a) An XML tree, T (b) Part of the Index tree of T

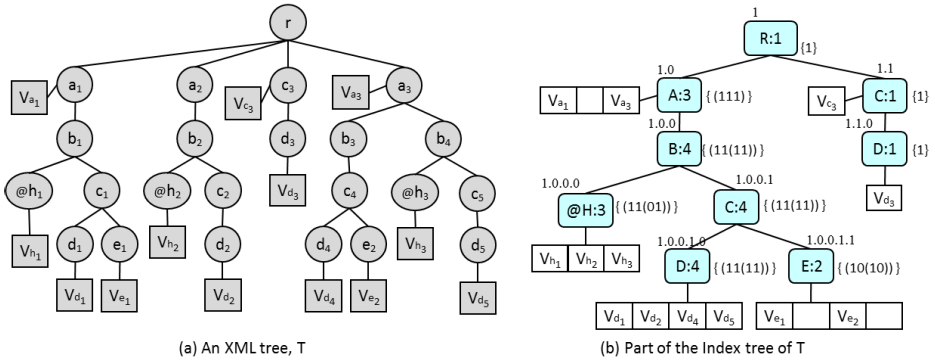
FIGURE 7. Example of branch map using encoding rule 4.

plain text nodes are represented by rectangles. The tag of an attribute starts with "@" for identification. When the gray nodes in Fig. 5(a) are visited, part of the index tree is as shown in Fig. 5(b). The branch map of each index node is enclosed by "{" and "}". The branch map of root R is therefore {1} according to Rule 1. When traveling from a_1 to e_1, there are no identical travel paths in the index; thus, the corresponding index nodes are generated, and the branch map consists only of {1} values according to Rule 2. The plain text nodes are also recorded in the index tree according to Rule 5.

When node a_2 is first visited, there is a corresponding index node A. Because a_2 has the same parent as a_1, a "1" is appended after the branch map of A, and two bits, representing a_1 and a_2, are grouped according to Rule 3. The branch of A is thus {(11)} at this point. Fig. 6 shows the index tree when node c_2 is first visited. When c_2 is visited the second time, as shown in Fig. 7, the child index node E of C is not visited in this round. According to Rule 4, a "0" is appended, and the two bits, 1 and 0, are grouped. The branch map of E is {(10)} at this point.

Fig. 8(b) shows the complete index tree. The number in the round rectangle represents the appearance frequency of the tag. For example, "A:3" indicates that nodes with the tag "a" appeared three times. In fact, the index tree does not actually exist; it is recorded as hash tables during the indexing construction phase. We describe this in the next section. To determine the relationship between index nodes in the hash tables, each index node is labeled using Dewey labels, which can be found at the top of each index node.

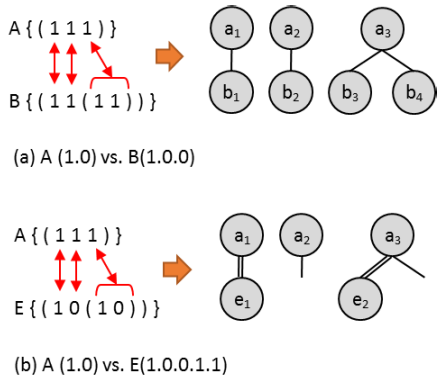
A branch map can be used to reconstruct the original XML architecture by comparing the relative positions of the bits of two index nodes. Fig. 9 shows two examples. Fig. 9(a) compares *A:3(1.0)* {(111)} and *B:4(1.0.0)*{(11(11))}. It is easy to determine that *A* is a parent of *B* based on the Dewey labels. Comparing the branch maps, the first two bits of *A* are "1", which correspond to the first two "1" bits of *B*. This means that there are two pairs of nodes A and B, i.e., (a-b, a-b), in the original XML tree. The third bit of A, which is "1", is related to the group following B, which is (11). This indicates that a branch occurred and that the third node of "a" has two b-tag children. Fig. 9(b) compares *A:3(1.0)*{(111)} and E*:2(1.0.0.1.1)*{(10(10))}. We know that A is an ancestor of E based on the Dewey labels. The first node "a" has an e-tag descendant, while the second node "a" does not, because the first two bits of the branch maps of A and E are {11} and {10}, respectively. In addition, a branch occurred in the path between the third node "a" and second node "e," and there is an e-tag node in the first fork path; however, there is no branch in the second fork path.



(a) An XML tree, T

(b) Part of the Index tree of T

FIGURE 8. Example of the complete index tree.



(a) A (1.0) vs. B(1.0.0)

(b) A (1.0) vs. E(1.0.0.1.1)

FIGURE 9. Reconstruction examples.

Compared to other labeling schemes, branch map requires less space because each node in the XML uses only one bit for labeling. Although some positions where a node does not exist are supplemented with a "0" to maintain the correspondence among nodes, the wasted space is much less than the saved space. The most novel feature of branch map, which cannot be handled well by other labeling schemes, is its ability to maintain the initial labeling state even after multiple updates.

IV. UCIS-X: AN UPDATABLE COMPACTED INDEXING SCHEME FOR XML

UCIS-X is an index structure applicable to an XML document that adopts the branch map labeling scheme. In this section, we first provide the basic conceptual structure of UCIS-X, followed by a query evaluation.

A. INDEX CONSTRUCTION

In Section III, an XML document is summarized as an index tree based on a branch map. Because the tree structure is inefficient for partial matching in most cases, the index tree is stored as a pair of hash tables, which are called UCIS-X. We use an algorithm similar to CIS-X for structural summarization (see Algorithm 1: Index-Construction in [13]). The differences between the two are the labeling schemes and the information stored in the hash tables. The index construction algorithm is shown in Algorithm 1, where the SAX parser is used to parse an XML document. The actS is a temporary stack used to record the active traversal path. While looking back at Section III, lines 7-14 handles the first visit to the root as in Fig. 5. Lines 24-18 deal with the first visit to nodes a_1 to e_1 in Fig. 5. Nodes a_2 to d_2 in Fig. 6 are processed by lines 17-22. Fig. 7 demonstrates the situation of rule 4 for an unvisited child E of C, which is addressed in lines 34-35. The output of Algorithm I is UCIS-X, a path index table and a content index table, as shown in Fig. 10. The path index stores the information regarding the path and structure with the tag name, represented by a hash key. Each hash entry points to a list that links data nodes having the same tag name. Each data node holds certain information (*label*, *branch*, *count*, *children*), where *label* is the Dewey label used to represent the positions of the nodes in the index tree, *branch* is the branch map and each "1" represents a node in the XML tree, *count* is the appearance frequency of the tag, and *children* records the tag names of the children of a node. The content index uses *label* as the hash key and stores the corresponding content. UCIS-X provides a compact format for an XML document. The original XML document can be rebuilt when the root node in UCIS-X is known.

B. QUERY EVALUATION

The query evaluation algorithm used by UCIS-X is shown in Algorithm 2. In the initial stage, we obtain the corresponding dataNode lists from T_p according to the QTP (query tree pattern) (lines 1-5). The first phase generates candidate data nodes (Function getCandidate), and the second phase filters out inconsistent bits in the branch maps (Function checkBM). For example, consider a query Q_1 ="//B[@H]/C[D]/E", where the QTP [14] is as shown in Fig. 11(a). The target node E is underlined. In the first phase, five linked lists in the path index table are examined: B, @H, C, D, and E. By getCandidate, the Dewey labels of each data node are checked, and the candidate data nodes are produced, as shown in Fig. 11(b). In the second phase, the branch maps are checked. Only bits at the relative position that are "1" are

Algorithm 1 Index Construction

Input: an XML document, D
Output: a path index table, T_p, and a content index table, T_c

```
1  Function startDocument()
2  initialize stack actS as empty and integer N as 0;
3  initialize hash tables T_p and T_c as empty;
4
5  Function startElement(Elemente)
6  if e is an attribute, e.fullname = "@" +e.fullname;
7  if actS is empty
8     create a new dataNode c
9     c.label="1", c.tagname= e.fullname;
10    c.Vid = N+1, and c.parent=N;
11    c.preBM= "(1" and c.postBM= ")";
12    c.count =1 and c.children = empty;
13    push(actS, c);
14    N + +;
15 else
16    p =top(actS);
17    if e.fullname ⊂ p.children and is the (x+1)th child of
         p;
18       mark the child of p as visited;
19       pathlist = T_p.get(e.fullname);
20       c = the dataNode with label (p.label+ ".x") in
            pathlist;
21       c.Vid = N+1;
22       updateBM(c, p)
23    else
24       create a new dataNode c
25       c.label=p.label+count(p.children), c.tagname=
            e.fullname;
26       c.Vid = N+1, and c.parent=p.Vid;
27       c.preBM= p.preBM, c.postBM = p.postBM;
28       c.count =1 and c.children = empty;
29    push(actS, c);
30    N + +;
31
32 Function endElement()
33    c =top(actS);
34    if there is any child y of c that has not been visited
35       updateBM(y, c);
36    pathlist = T_p.get(c.fullname);
37    if pathlist = null, create a new pathlist;
38    if c.label is in the pathlist
39       update corresponding dataNode in pathlist;
40    else append c into pathlist;
41    T_p.put(c.fullname, pathlist);
42    Pop(actS, c)
43
44 Function characters(Stringvalue)
45    c =top(actS);
46    contentlist = T_c.get(c.label);
47    if contentlist = null, create a new content list;
48    append (value) into contentlist;
49    T_c.put(c.label, contentlist);
50
51 Function endDocument()
52    //Finished
53
54 updateBM(c, p)
55    if c.parent == p.Vid
56       if c is the second child of p.Vid
57          c.preBM = c.preBM.substring(0, length-2)+
               "("+ "11";
58          c.postBM = ")"+c.postBM;
59       else
60          c.preBM = c.preBM+ "1";
61    else
62       if p.preBM end with "(11"
63          c.preBM = c.preBM.substring(0, length-2)+
               "("+ "11";
64          c.postBM = ")"+c.postBM;
65       else
66          c.preBM = c.preBM+ "1"
```

matched (lines 23-24). In this case, only the first bits of all branch maps are "1". By accessing the content index using the label of E, "1.0.0.1.1", the final result "<e>V_{e1} </e>" can be outputted.

C. BASIC UPDATE OPERATIONS

In recent studies, several update operations and languages for XML have been defined [22], [26]. The XQuery Update Facility language [41] is recommended by the World Wide Web Consortium (W3C) and provides the ability to modify some parts of an XML document and leave the remainder unchanged. Although the W3C approach includes multiple update operations, the basic update operations are insert, delete, replace, and rename. In this section, only the insertion and deletion operations are demonstrated because the replace and rename operations can be completed by directly changing some features of the existing node or through a series of deletions and/or insertions. All update processes are directly executed using UCIS-X without index rebuilding. How UCIS-X supports W3C XQuery update operations is introduced in the next sections.

1) INSERTION

For UCIS-X, the insertion process can be classified into three cases. The basic insertion algorithm is shown in Algorithm 3.

Case 1: when there is a reserved space denoted by "0" in the *branch* of the corresponding data node for the new node, the corresponding bit in the *branch* is changed from "0" to "1", and the value of *count* is increased by 1. For example, "<e_3>Ve_3</e_3>" is inserted as the second child of c_2, as shown in Fig. 12. The *branch* of E(1.0.0.1.1) is {(10(10))}, where the second bit, denoted as "0", is the space reserved for this insertion. The *branch* is then updated from

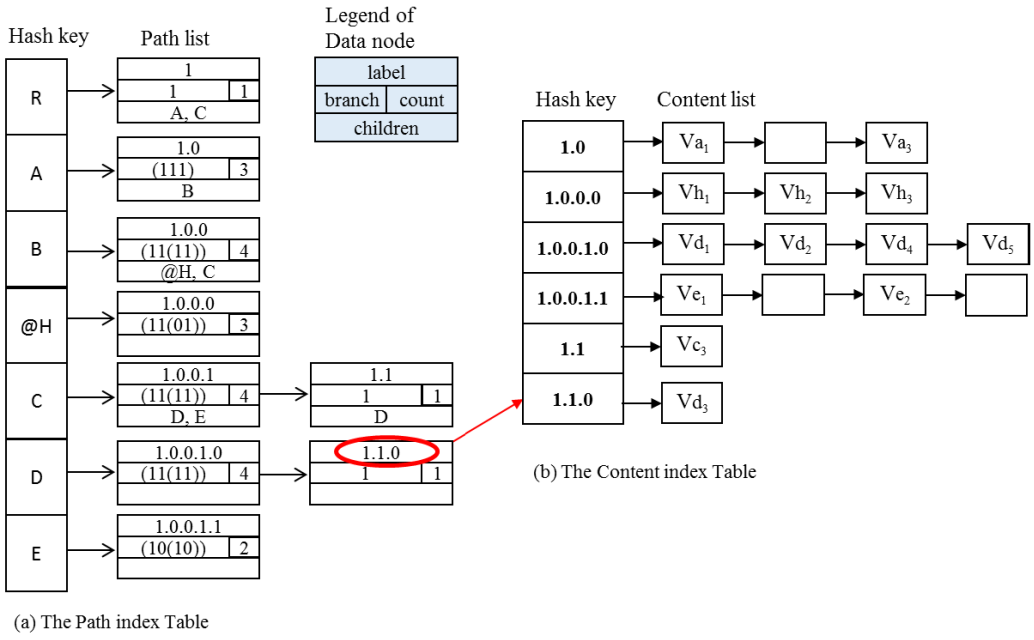


FIGURE 10. UCIS-X of an XML tree T.

{(10(10))} to {(1**1**(10))}, and the value of *count* is updated from 2 to 3. Then, the label of *E*, "1.0.0.1.1", is used to access the corresponding content list. Write "Ve_3" to the second position to complete this update operation.

Case 2: when there is no reserved space in the *branch* and the traversal path of the new node exists in the index tree, the *branches* of the target data node and its descendants (if any) need to be updated. Following the principles of Rule 3 in Section III, a "1" is added to the *branch* of the target data node, and bits with the same parent are grouped by "(" and ")." In addition, a "0" is added to the *branch* of each descendant of the target data node. An example is shown in Fig. 12, where "$<d_6>Vd_6</d_6>$" is inserted as the second child of c_5. The *branch* of D(1.0.0.1.0) is {(11(11))}, and there is no reserved space for this insertion. The new node $d_6$is therefore inserted after d_5, and both have the same parent. Therefore, the branch is updated from {(11(11))} to {(11(1**11**))}, and the value of the count is updated from 4 to 5. Then, the corresponding content list of D is accessed through the "1.0.0.1.1" label, and "Vd_6" is appended at the end of the list.

Case 3: if the traversal path of the new node does not exist in the index, a data node is created, and the tag name of the new node is added to the *children* of the parent (Fig. 12). In this case, "$<g_1></g_1>$" is inserted under a_1, and a new data node with "G" as the hash key is created in the path index. Following the principles of Rule 2 in Section III, the length of the branch map of the new data node is equal to the length of its parent. Therefore, the length of *branch* G is 3. Because g_1 is the child of a_1, the first bit of the *branch* is "1", while the others are "0". Finally, add "G" to the *children* of A to complete this update operation.

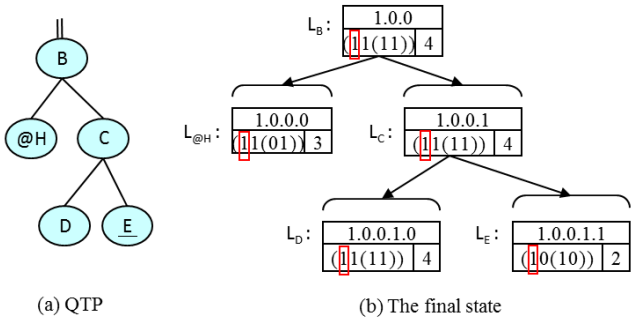


FIGURE 11. Query Evaluation of Q1 = //B[@H]/C[D]/E using UCIS-X.

The costs to insert a node in Cases 1 or 3 are low because only a few data nodes must be updated. The worst case occurs in Case 2, when cascading updates in the descendants occur. A related example is when the maximum number of updated nodes is equal to the depth of the index tree (Example I_4). However, this case is similar to reserving space for future insertions, which is quite similar to reserving columns of attributes for an inserted tuple in a relational database.

2) DELETION

For deletion, the easiest method is to find the corresponding digit and change it from "1" to "0". After deletion, one of the following three situations may be encountered. We consider several examples to show that this method will not break the rules of the branch map.

Case 1: when the node to be deleted does not have a sibling node with the same tag, change the corresponding bit from "1" to "0". An example is shown in Fig. 13. When

Algorithm 2 Query Evaluation

Input: a QTP, Q, with n nodes $\{N_1, N_2, \ldots, N_n\}$, and the UCIS-X, T_p and T_c
Output: a set of branch maps with marked matching digits

```
1  Function queryEvaluation(Q, Tp, Tc)
2  initialize a tree T as empty;
3  Travel Q in preorder, for each node Ni in Q
4     obtain the sequence of Ni-type dataNodes from Tp
   by the tagname Ni;
5     add the Ni-list to T in the same position of Q;
6  getCandidate(T);
7  checkBM(T);
8
9  Function getCandidate(T)
10 initialize n pointer to the first dataNodes of each Ni-list
11 for each dataNode x of Nroot-list
12    check each other pointed dataNode y of Ni-list
13    while length(y.label) <= length(x.label)
14       y->next dataNode
15    if one of the pointed dataNode y is not the child of x
16       remove x from Nroot-list
17    else
18       x->next dataNode
19
20 Function checkBM(T)
21 initialize n pointer to the first dataNodes of each Ni-list
22 while not end of list
23    for each branchMap of dataNodes
24       if all the corresponding bits are "1"
25       mark as matched.
26    each n->next dataNodes
```

Algorithm 3 Basic Insertion

Input: the UCIS-X, T_p and T_c, a target node p, and an inserted node c
Output: the updated UCIS-X

```
1  Function insertion(Tp, Tc, p, c)
2  find the dataNode x of the p-type list and mark the digit
   of p;
3  if a c-type dataNode y is found, which is the child of x,
4     check the corresponding digits of the branch maps of
      x and y
5     if the position of c to be inserted is denoted by "0"
6        change "0" to "1" //Case 1
7        update Tc if corresponding data exist
8     else
9        change the previous digit "d" to "(d1)" //Case 2
10       update Tc if corresponding data exist
11    y.count ++;
12 else //Case 3
13    create a new c-type dataNode y
14    y.label = x label+count(x.children),
         y.tagname=c.fullname;
15    y.count = 1;
16    y.branchmap = x.branchmap, where all values of
         "1" are set to "0" and the position of c is set to
         "1"
17    update Tc if corresponding data exist
```

"$<d_1>Vd_1</d_1>$" is deleted and because d_1 is the only d-tag child of c_1, the branch of D(1.0.0.1.0) is updated from {(11(1(11)))} to {(**0**1(1(11)))}, and the count is reduced by 1. Then, the corresponding content list is accessed, and "Vd_1" is deleted.

Case 2: when the node to be deleted has a sibling node with the same tag, there are two choices available for the deletion. The first step is the same as in Case 1. For example, after deleting "$<d_5>Vd_5</d_5>$" (see Fig. 13), the branch and count of D(1.0.0.1.0) are updated from {(01(1(11)))} to {(01(1(**0**1)))} and from 4 to 3, respectively. The deletion can end here but will leave some unused space. The branch of D(1.0.0.1.0) will be {(01(11)) }, and the space for Vd_5 will be deleted from the content list. Another choice is to reserve them for later insertions. In practice, garbage collection can be triggered at an appropriate time if necessary.

Case 3: when the count of a data node is equal to 1 and only one node is to be deleted, then similar to Case 2, the data node is either reserved or deleted. The example in Fig. 13 reserves G in the path index even if no such path exists. It is also alternatively acceptable to delete data node G.

We demonstrate that the initial encoding rules of the branch map are not broken after multiple insertions and deletions. This feature is difficult to achieve using the labeling methods proposed in the current literature. Another feature is that all the queries and updates are performed under the structure of UCIS-X instead of the original XML document. We know that when performing an update operation, the first step is to find the target node and then perform the update. Almost all other methods involve finding the locations of the target nodes through the index and then traveling across an XML document to reach the target node for updating. Because UCIS-X is a compact form of XML, any searching and updating operations can be completed in succession without having to visit the original document. UCIS-X is one of the few methods that can fully integrate the functions of labeling, indexing, querying, and updating.

D. SUPPORTED W3C XQUERY UPDATE OPERATIONS

The update operations defined by UCIS-X are based on the W3C XQuery Update Facility 3.0. Twelve update operations are defined by the W3C. For update operations, *$target, $content, or $replacement* represent XPath expressions [2], [42] that specify a set of XML node(s). UCIS-X can support almost all of the update operations except the *put* operation, which stores an XDM (XQuery and XPath Data Model) node tree to a location specified by a valid absolute user resource identifier (URI). Furthermore, because UCIS-X

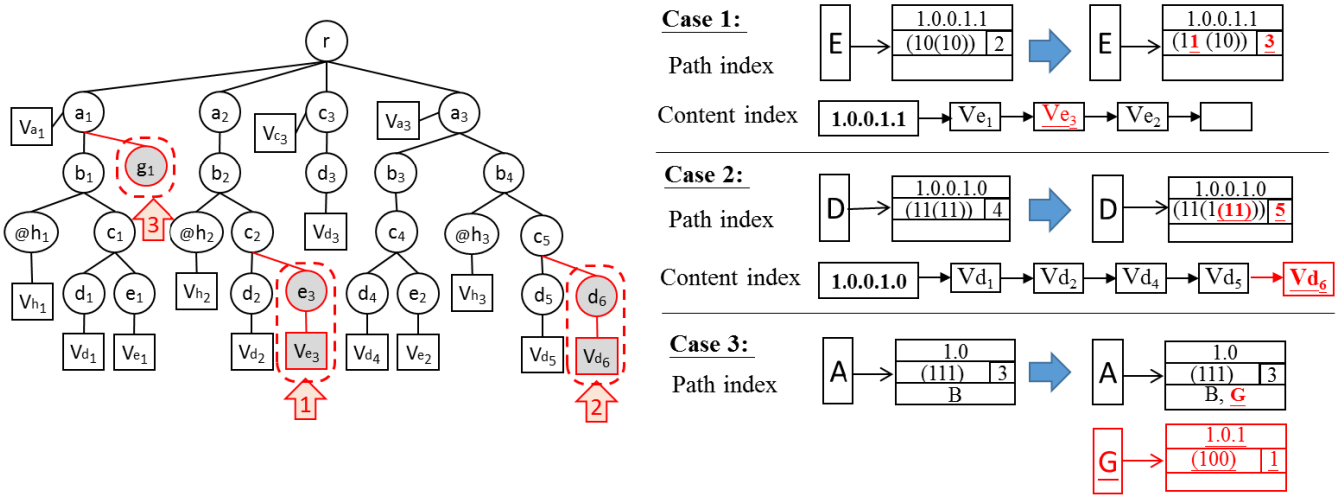


FIGURE 12. Insertion examples.

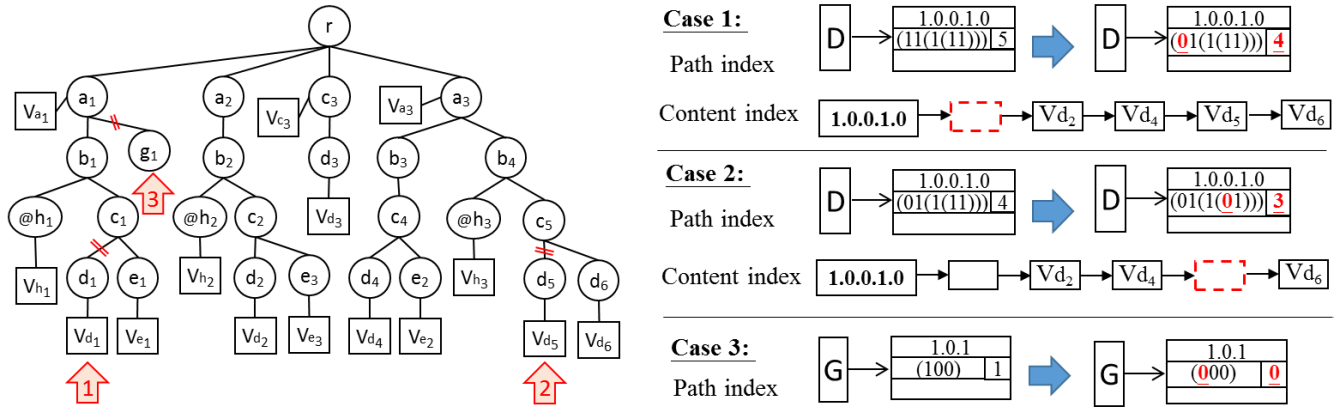


FIGURE 13. Deletion examples.

supports unordered matching of XML [42], a few insert operations defined by UCIS-X are different from those recommended by the W3C. Fig. 14 shows the initial state of an XML tree and the summarized index tree obtained using UCIS-X. The description of each update operation is followed by an example with graphical explanations. The update operations defined by UCIS-X are as follows.

1) INSERTINTO($TARGET, $CONTENT)

Insert *$content* as the child of *$target*. If the root of *$content* does not exist in UCIS-X, then *$content* will be inserted as the last child of *$target*. Otherwise, *$content* will be inserted after the children, with the same tag name *$target*.

Example I_1 (insertInto (/R/A[2], <X><Y></Y></X>)): Because the path "R/A[2]/X/Y" does not exist in UCIS-X, node x (denoted by "x_1" in Fig. 15) is supposed to be inserted as the last child of "R/A[2]" in the XML tree, followed by the insertion of node y (denoted by "y_1") as the child of x. In UCIS-X, two new data nodes X and Y are created at "1.0.1" and "1.0.1.0" instead. Fig. 15 shows the results after the insertion operation.

2) INSERTATTRIBUTES($TARGET, $CONTENT)

Insert *$content* as attributes of *$target*. Similar to ***insertInto***, *$content* will be inserted as the last child of *$target* or after the children with the same tag name *$target*.

Example I_2 (insertAttributes (/R/A[2], @id= "#01")): Because the path "R/A[2]/@id" does not exist in UCIS-X, attribute "id" (denoted by "$@id_1$" in Fig. 15) is supposed to be inserted as the last child of "R/A[2]". However, in UCIS-X, a new data node @ID is created at "1.0.2". Fig. 15 shows the results after insertion.

3) INSERTINTOASFIRST ($TARGET, $CONTENT)/ INSERTINTOASLAST ($TARGET, $CONTENT)

Insert *$content* as a child of *$target* so that *$content* becomes the first/last sibling of all nodes with the same tag name.

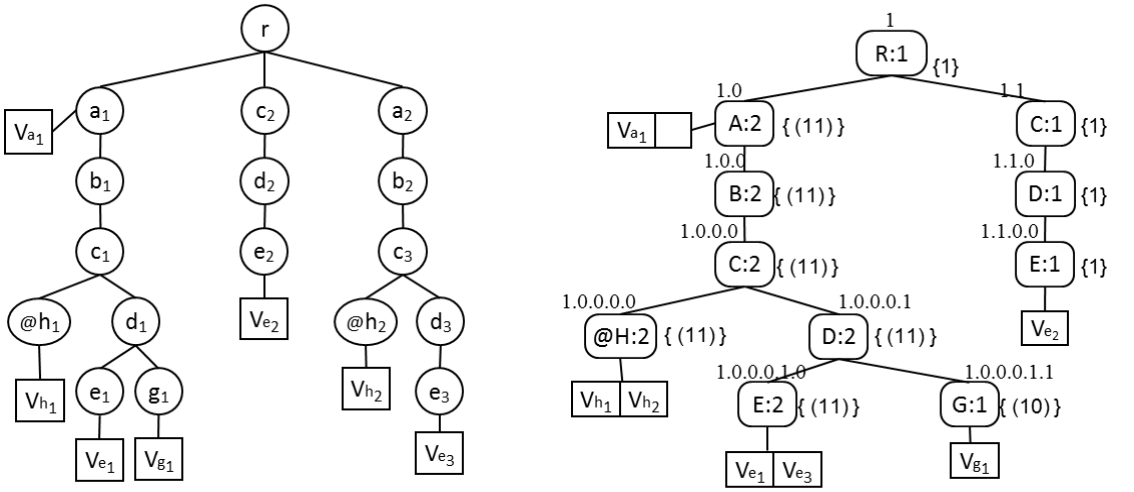


FIGURE 14. Initial state of the XML and the summarized index tree obtained using UCIS-X.

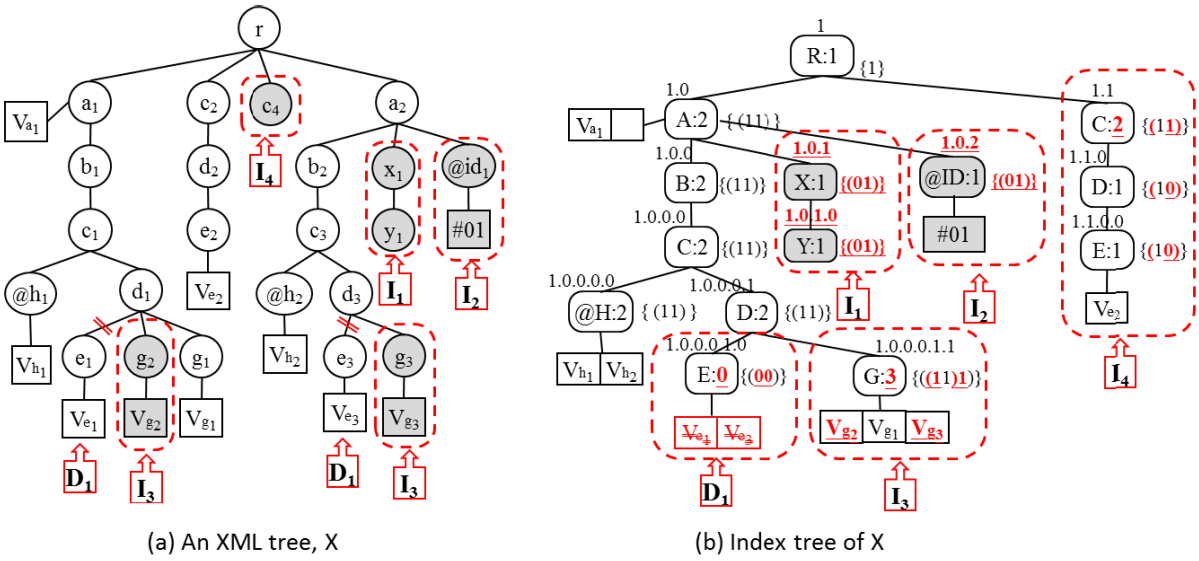


FIGURE 15. Final state after Examples I1, I2, I3, and D1.

Example I_3 (insertIntoAsFirst (/R/A/B/C/D, <G> Vg</G>)): There are two nodes that meet "*/R/A/B/C/D*" in Fig. 15. Node d_1 is the first matched node. Because it has a g-tag child, g_1, a new g-tag node is inserted before g_1, denoted by "g_2". The second matched node d_3 does not have a g-tag child; thus, a new g-tag node is inserted as the last child of d_3. UCIS-X is therefore updated directly. In this example, multiple nodes are inserted simultaneously. If each node in the XML tree has a unique label, many labels must be created. In contrast, only one data node needs to be updated using UCIS-X.

4) INSERTBEFORE ($TARGET, $CONTENT)/ INSERTAFTER ($TARGET, $CONTENT)

Insert *$content* immediately before/after *$target* if the leaf node of *$target* and the root of *$content* have the same tag name. Otherwise, insert *$content* as the last child of *$target*. Note that this condition is applicable to the unordered matching of XML, which is slightly different from the W3C definition.

Example I_4 (insertAfter (/R/C[1], <C></C>)): Node c_2 in Fig. 15 matches *$target*, and the tag name is the same as the root of the path "<C></C>". A new node c_4 is supposed

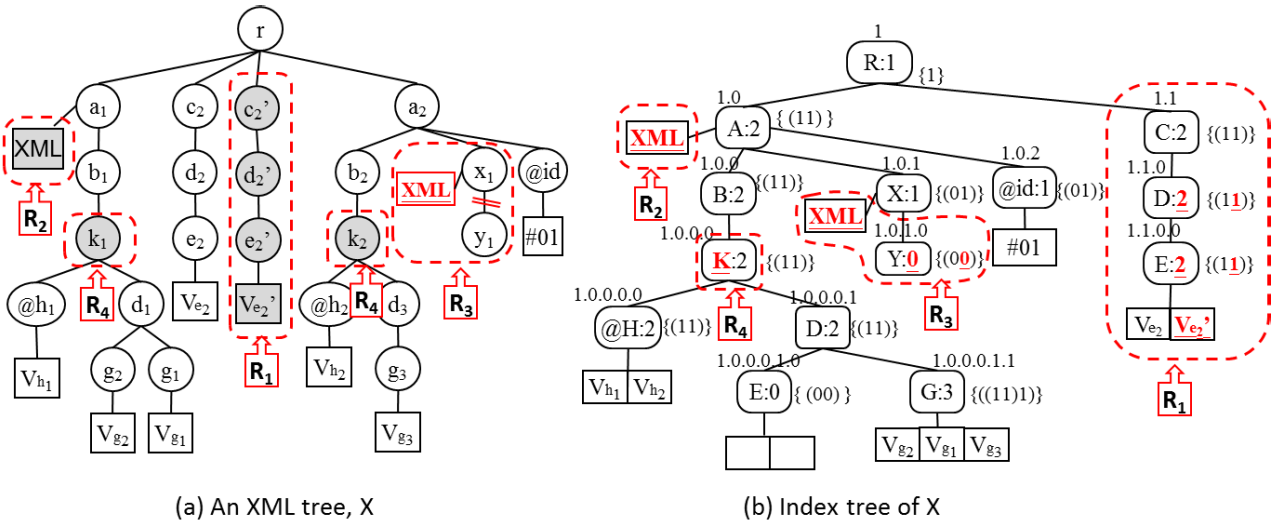


FIGURE 16. Final state after examples R1, R2, R3, and R4.

to be inserted immediately after c_2. In UCIS-X, according to this insertion, data node C(1.1) is updated instead, and the branch and count are changed from "{1}" to "{(11)}" and from 1 to 2, respectively. Note that cascading updates occur in this case. The branches of all the descendants of C must be updated from "{1}" to "{(10)}" to maintain the complete mapping relationship.

5) DELETE($TARGET)

Delete the subtree rooted at *$target*. The descendants of *$target* are automatically deleted.

Example D_1 (delete(/R/A/B/C/D/E)): There are two nodes that meet "*/R/A/B/C/D/E*" in Fig. 15: e_1 and e_3. After deleting them as well as their content values, the count of data node D in the index tree is set to 0. As mentioned previously, data node D can be either reserved or deleted.

6) REPLACENODE ($TARGET, $REPLACEMENT)

Replace *$target* with *$replacement*.

Example R_1 (replaceNode (/R/C[2], /R/C[1])): The second node of "/R/C" will be replaced with the content of the first node of "/R/C". In Fig. 16, there are three nodes and a content value rooted at the first node, denoted by c_2, which are copied and pasted to the position of the second node, denoted by c_4, before updating. In UCIS-X, only the subtree rooted at C(1.1) is updated. Because the spaces of the descendants of C were created in advance in Example I_4, the cost of this updating was reduced.

7) REPLACEVALUE ($TARGET, $STRING-VALUE)

Replaces the string value of *$target* with *$string-value*.

Example R_2 (replaceValue (/R/A[1], "XML")): The first a-tag node of "/R/A" is denoted by a_1 in Fig. 16. The string value of a_1, "V_{a1}", is replaced by *$string-value,* "XML." Please refer to Fig. 16 for the results obtained after performing the replacing operation.

8) REPLACEELEMENTCONTENT ($TARGET, $TEXT)

Replace the existing children of the element node *$target* with the optional text node *$text*. The attributes of *$target* are not affected.

Example R_3 (replaceElementContent (/R/A[2]/X, "XML")): The second a-tag node of "/R/A" is denoted by a_2, and x_1 matches "/R/A[2]/X" in Fig. 16. Each child of x_1 is set to empty, and therefore, y_1 is deleted in this case. *$text*, "XML," is placed in x_1. The updated results of UCIS-X based on this rule are shown in Fig. 16.

9) RENAME ($TARGET, $NEWNAME)

Change the tag name of *$target* to *$newName*.

Example R_4 (rename (/R/A/B/C, K)): There are two nodes that meet "*/R/A/B/C*" in Fig. 16: c_1 and c_3. The results of changing the tag name "c" to "k" are shown in Fig. 16. This is also an example of updating multiple nodes simultaneously. However, using UCIS-X, only one data node is updated.

V. EXPERIMENTAL DESIGN AND RESULTS

In this section, the proposed labeling scheme based on UCIS-X was compared with several other labeling schemes: ORDPATH, DFPD, DPLS, and CIS-X. ORDPATH is a reference indicator used in Microsoft®SQL Server™. DFPD and DPLS are relatively recent methods. Because DPLS has been compared with many methods and performs well, we omit a comparison with other comparable methods. A comparison of the five labeling methods is shown in Table 1. Because the studies on DFPD and DPLS did not specify the query method used, to obtain a fair comparison, a data structure (hash tables) and query method (TwigList) similar to those used by UCIS-X were applied to the greatest extent possible.

TABLE 1. Comparison of labeling methods.

Method	Target of encoding	Original encoding	Future expansion for update	Indexing structure	Query method
ORDPATH	nodes of XML	Dewey-based odd numbers	even numbers	table-based NODE table and secondary indexes	query plan
DFPD	nodes of XML	Dewey-based integers	fractions without reduction	not specified	not specified
DPLS	nodes of XML	Dewey-based integers	fractions with reduction	not specified	not specified
CIS-X	nodes of XML	preorder integers	not supported	hash-table-based CIS-X	TwigList-based algorithm
UCIS-X	nodes of index	branch map	nested branch	hash-table-based UCIS-X	TwigList-based algorithm

TABLE 2. Characteristics of XML datasets.

Dataset	Data size (MB)	Elements	Attributes	Total	Max./avg. depth	Max. fan-out	Number of distinct tags
DBLP	127	3,332,130	404,276	3,736,406	6/2.9	324,540	40
XMark	111	1,666,315	381878	2,048,193	12/5	2,550	83
Nasa	24	476,646	56,317	532,963	8/5.6	2,435	68

The methods were evaluated with respect to the index construction cost, query evaluation performance, and update performance.

The experiments were performed using a Windows 7 system with an Intel Core i7-2600 3.4 GHz central processing unit and 16 GB of RAM. Three widely used datasets for benchmarking XML indexing methods were chosen: DBLP, XMark, and Nasa. The statistical data for the three datasets are shown in Table 2. DBLP represents the class of low-depth and high-fan-out documents. XMark embodies a document of high depth and low fan-out. Nasa is of a small size with a low fan-out. A noticeable characteristic of DBLP is the high average number of node repetitions. The total number of nodes of elements and attributes is 3,736,406 in DBLP, but only 40 distinct tags are used. In contrast, there are only 532,963 nodes in Nasa, the number of distinct tags of which is 68, which is greater than that of DBLP. The follow-up will also explain whether these characteristics affect the results of the experiments.

A. INDEX CONSTRUCTION COSTS

The index construction costs include the construction time and the required size. Because the index mainly records structural information, there is no special processing for the plain text nodes; only the index sizes of the structure are compared here. The results of the index construction time for the different methods are shown in Fig. 17. ORDPATH, DFPD, and DPLS use the Dewey order for the initial encoding. DFPD and DPLS follow the same method in the initial stage; thus, the time spent is the same. The construction time for ORDPATH is slightly longer than for the other two methods, probably because ORDPATH assigns only positive odd integers, which requires a larger number of calculations. Typically, the structural summary indexing methods require more time to set up because building the summarizing structure is time-consuming [13]. CIS-X and UCIS-X improve the construction performance by using a temporary stack to record an active traversal path and thus avoid actually building a summarized index tree. The results indicate that the time spent by CIS-X or UCIS-X is less than that of the other methods. Comparing CIS-X and UCIS-X, CIS-X needs only cluster nodes, while UCIS-X must additionally compare the mapping relationships between parent and child nodes; thus, UCIS-X requires a longer build time than does CIS-X.

To achieve more precise discriminability, the indexes in the memory were written to text files, and their sizes were compared. Fig. 18 shows the required index sizes for the different methods. ORDPATH had a larger index size than that of DFPD and DPLS, as expected. Both CIS-X and UCIS-X use streamlined labeling methods, which efficiently save space. UCIS-X performed better than CIS-X by using one bit instead of an integer to record the position of each node in the XML data tree. Based on the above experimental results, CIS-X and UCIS-X demonstrate excellent performance in terms of the index construction costs.

The nodes of the XML trees were visited in a preordered manner, where ORDPATH, DFPD, and DPLS, using TwigList, clustered the labels in linked lists with the same tag name and CIS-X and UCIS-X merged those labels with the same tag name. We observed that depth and fan-out had not significant effects on the cost of index construction. However, CIS-X and UCIS-X were more affected by the degree of node repeatability. The higher the repeatability of a node,

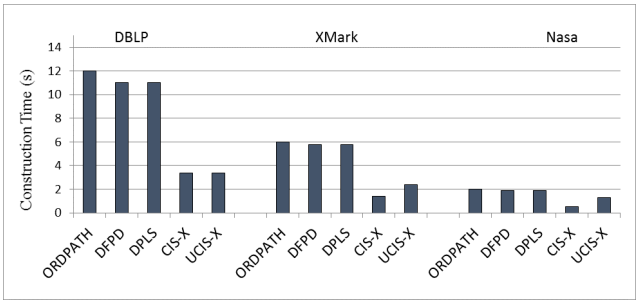


FIGURE 17. Comparison of index construction times.

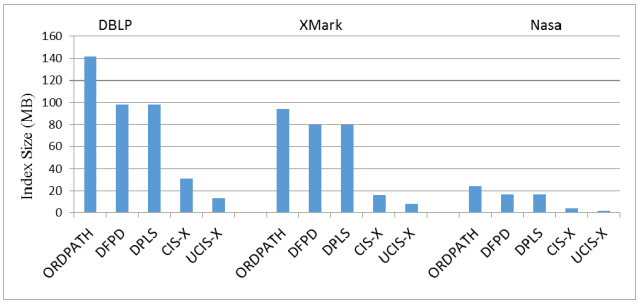


FIGURE 18. Index sizes required by different methods.

the better for CIS-X and UCIS-X. For example, DBLP is a dataset with highly repetitive nodes, while Nasa is relatively less repetitive. The costs of index construction in DBLP (Fig. 17 and Fig. 18) for CIS-X and UCIS-X were obviously much less than those of the other three methods. However, only slight differences exist in Nasa.

B. PERFORMANCE ON QUERY EVALUATIONS AND UPDATE OPERATIONS

In this section, the update performances are compared for the above methods, except CIS-X, which does not provide an update-friendly labeling mechanism. Because insertions can trigger the creation of new nodes, modification of existing nodes, and/or deletion of related nodes, most research has focused on insertions. In addition, other operations can be created by a series of deletions and/or insertions. Therefore, the following experiments focused on insertions and deletions.

The index structures of the four methods clustered nodes with the same tag name and stored them in hash tables. The position of the node to be inserted or deleted was not the main factor affecting the process of query evaluation, insertion and deletion. However, due to the different theories of the labeling methods, the worst cases happen in different situations. For ORDPATH, DFPD, and DPLS, the worst case of insertion occurs when all new nodes are inserted concentratedly in a certain location (such as in Figs. 2, 3, and 4). For UCIS-X, the worst case of insertion occurs in Case 2, such as Example I_4. To avoid an unfair experimental design, YFilter [43] was used to generate test target expressions and evaluate the overall performance.

Each update operation first searches the target nodes and then performs insertion or deletion. To examine the query and update performances, a large number of update expressions were fed into each method. Each update expression included a target expression for searching the target node(s) and an active expression, which actually updated the node(s). The target expressions were single-path queries generated by YFilter. Two types of target expressions were examined. The first type was XPath expression with a position predicate. For example, the *update expression* of "***insertAfter(*** *//<author>/books/book[1]/author,* *<author>Thomas</author>)*" contains a position predicate "*book[1]*" and inserts another *author* element after the *author* of the first *book*. Therefore, only one *author* element is inserted. The second type was XPath expression without a position predicate. For example, "***insertAttributes(***//books/book), @code="B"***)*** inserts a *code* attribute into each *book*, which triggers numerous insertions.

1) INSERTION PERFORMANCE

For insertion, ORDPATH, DFPD, and DPLS always create new nodes in the original XML document. The insertion process using UCIS-X is classified into three cases. Cases 1 and 2 involve modification of the related branch maps, and Case 3 creates new nodes, as discussed in Section IV. All modifications are made within the summarized index structure. One hundred insertion expressions were sequentially processed by each method, the results of which are shown in Figs. 19 and 20. Because the processing time range was large, a logarithmic scale was used for the vertical axis.

All three methods, i.e., ORDPATH, DFPD, and DPLS, are update-friendly labeling methods, but they are inefficient in label evaluation, as this process requires considerable time to search for the target nodes. Each insertion must first check the labels of the neighboring nodes to calculate the value of the target label and then assigns a new space for the new node. Moreover, each insertion will cause further expensive manipulation and storage costs. In contrast, the proposed UCIS-X must create only new index nodes under certain conditions or modify an existing branch map of index nodes to avoid creating a large number of nodes at once. Therefore, the execution times of UCIS-X for insertion were faster than those of the other three methods. We also noticed that by comparing the insertion times with and without the position predicate, the gap between UCIS-X and the other methods increased for the latter, reflecting the advantages of UCIS-X with the summarized index method.

2) DELETION PERFORMANCE

Figs. 21 and 22 illustrate the number of executions for 100 deletions when using each method. For deletion, ORDPATH, DFPD, and DPLS deleted nodes and released the space without any relabeling issues. However, if the deleted node was an intermediate node, a cascading deletion was triggered. Using these three methods, searching for the descendants of the

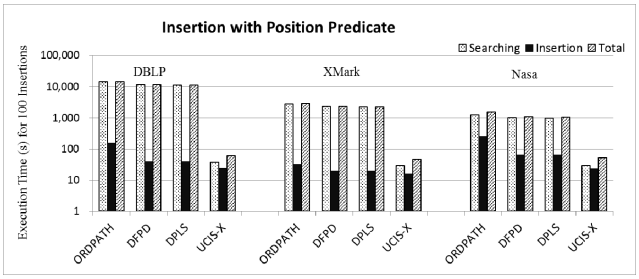


FIGURE 19. Execution time for insertion with position predicate.

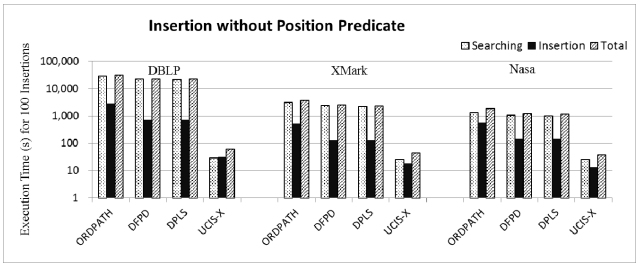


FIGURE 20. Execution time for insertion without position predicate.

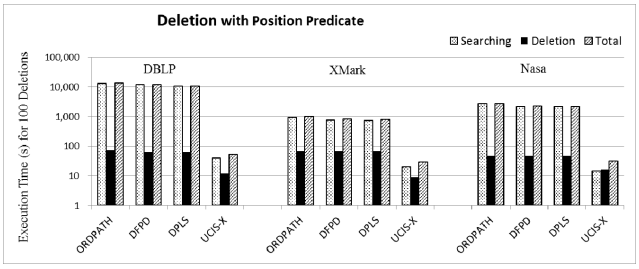


FIGURE 21. Execution time for deletion with position predicate.

deleted node became expensive, which caused the methods to spend more time than expected to process a deletion. In contrast, it is not necessary to delete nodes using UCIS-X while processing a deletion by reserving the deleted space for later insertions. Cascading deletion may also occur in the UCIS-X method; however, because it is relatively easy to find descendants in the index structure, the overall speed of UCIS-X is still faster than that of the three other methods. Similarly, comparing Figs. 21 and 22, when deleting multiple nodes at once, the efficiency of UCIS-X is more obvious.

A noteworthy result is that when processing deletion with position predicates (Fig. 21), the execution times of ORDPATH, DFPD, and DPLS were almost the same despite the use of different databases. This result indicated that the characteristics of the dataset have little effect on deletion. The same is true for UCIS-X. While processing deletion without position predicates (Fig. 22), ORDPATH, DFPD, and DPLS spent almost the same amount of time for the same dataset. The deletion time differed for different datasets mainly due to the number of deleted nodes. However, UCIS-X still had no significant difference among different datasets.

3) DIFFERENCES IN THE INDEX SPACE

To compare the increases in the index size after multiple updates, 200 insertion and 200 deletion expressions were

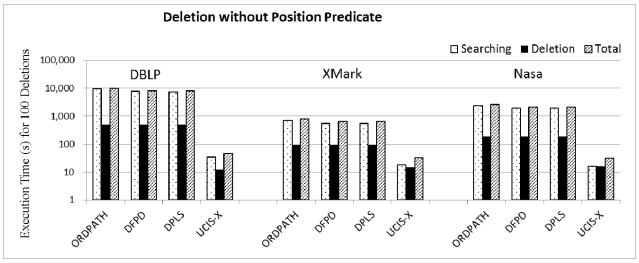


FIGURE 22. Execution time for deletion without position predicate.

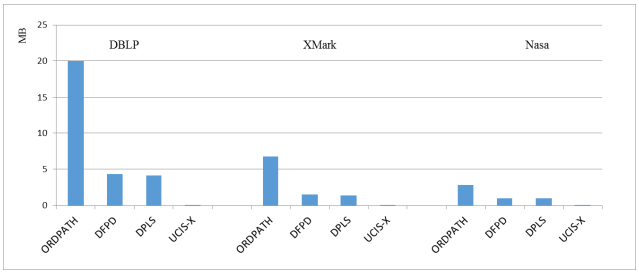


FIGURE 23. Increase in index size after 200 insertions and 200 deletions.

mixed and fed into each method. Fig. 23 shows the differences in the index size after execution. The increased index size is the index size after execution minus the index size before execution. Among all methods, the size increase achieved using ORDPATH was the largest because it only uses odd numbers for normal encoding and even numbers for extending. Because the even numbers are virtual labels, they increase the sections of a Dewey label and require additional space. The encoding principles of DFPD and DPLS are relatively similar; the difference is that DPLS considers the mechanism of label reuse. Thus, DPLS performed slightly better than DFPD in terms of the required index space. UCIS-X maintains the original labeling state after multiple insertions and deletions; thus, the index space normally does not change significantly. In some cases, the space may be reserved for future insertions, as previously discussed.

VI. CONCLUSION

In this paper, we reviewed some well-known indexing methods and query evaluation algorithms. In addition, the problems associated with existing labeling schemes for supporting dynamic XML updates were discussed. To overcome these problems, UCIS-X with a branch map was proposed. The advantages of UCIS-X include compression of the index structure, the support of efficient query processing, and the dynamic update of XML documents. Moreover, UCIS-X can support most update operations defined by the W3C without disturbing the original labels.

Several methods were compared with UCIS-X: ORDPATH, DFPD, DPLS, and CIS-X. The results demonstrate that CIS-X and UCIS-X can be constructed quickly and require less space than the other three methods. The experimental results also indicate that the execution time for the update processing of a branch map is more efficient when using UCIS-X than that when using ORDPATH, DFPD, or DPLS.

UCIS-X benefits from the branch map scheme and achieves both efficient structural information extraction and low storage consumption.

There are several issues worthy of further discussion. First, the space required for indexes was simply compared with the text files in the experiments. To keep the indexes in the database in the future, a theoretical analysis method is necessary. Second, although UCIS-X is excellent regarding its index space, query speed, and update friendliness, it is suitable only for XML partially ordered matching. The UCIS-X design is based on the fact that the children of a node are represented in two types: unordered collection for nodes with different tag names and ordered collection for nodes with the same tag names. It is important to know the different tag names of the children of each node, but their order can be ignored. Only the order of children with the same tag name was preserved, because the information would be meaningful in some cases. For example, the first and second authors of a book are meaningful, and the branch map preserves this information. Although the proposed method can only support partially ordered updates, the experimental results show that UCIS-X has significant improvements in terms of the index construction time and index size, as shown in Fig. 17 and Fig. 18. However, an ordered updatable XML is necessary in some applications. Therefore, in the future, we will attempt to design a user-friendly indexing and labeling method that supports the ordered matching of XML documents.

REFERENCES

[1] E. Menahem, A. Schclar, L. Rokach, and Y. Elovici, "XML-AD: Detecting anomalous patterns in XML documents," *Inf. Sci.*, vol. 326, pp. 71–88, Jan. 2016, doi: 10.1016/j.ins.2015.07.007.

[2] G. Z. Qadah, "Indexing techniques for processing generalized XML documents," *Comput. Standards Interfaces*, vol. 49, pp. 34–43, Jan. 2017, doi: 10.1016/j.csi.2016.07.002.

[3] Z. Brahmia, H. Hamrouni, and R. Bouaziz, "XML data manipulation in conventional and temporal XML databases: A survey," *Comput. Sci. Rev.*, vol. 36, May 2020, Art. no. 100231, doi: 10.1016/J.COSREV.2020.100231.

[4] Y. Khan, A. Zimmermann, A. Jha, V. Gadepally, M. D'Aquin, and R. Sahay, "One size does not fit all: Querying Web polystores," *IEEE Access*, vol. 7, pp. 9598–9617, 2019, doi: 10.1109/ACCESS.2018.2888601.

[5] J. R. Quinones and A. J. Fernandez-Leiva, "XML-based video game description language," *IEEE Access*, vol. 8, pp. 4679–4692, 2020, doi: 10.1109/ACCESS.2019.2962969.

[6] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, Jul. 2015, doi: 10.1109/TKDE.2015.2427795.

[7] Q. Chen, A. Lim, and K. W. Ong, "D(k)-index: An adaptive structural summary for graph-structured data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, San Diego, CA, USA, 2003, pp. 134–144, doi: 10.1145/872757.872776.

[8] Q. Chen, A. Lim, and K. W. Ong, "Enabling structural summaries for efficient update and workload adaptation," *Data Knowl. Eng.*, vol. 64, no. 3, pp. 558–579, Mar. 2008, doi: 10.1016/j.datak.2007.09.012.

[9] C.-W. Chung, J.-K. Min, and K. Shim, "APEX: An adaptive path index for XML data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2002, pp. 121–132, doi: 10.1145/564704.564706.

[10] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A fast index for semistructured data," in *Proc. 27th Int. Conf. Very Large Databases (VLDB)*, 2001, pp. 341–350.

[11] R. Goldman and J. Widom, "DataGuides: Enabling query formulation and optimization in semistructured databases," in *Proc. 23rd Int. Conf. Very Large Databases (VLDB)*, 1997, pp. 436–445.

[12] W.-C. Hsu, I.-E. Liao, S.-Y. Wu, and K.-F. Kao, "An efficient XML indexing method based on path clustering," in *Proc. 20th IASTED Int. Conf. Modeling Simulation*, 2009, pp. 339–344.

[13] W.-C. Hsu and I.-E. Liao, "CIS-X: A compacted indexing scheme for efficient query evaluation of XML documents," *Inf. Sci.*, vol. 241, pp. 195–211, Aug. 2013, doi: 10.1016/j.ins.2013.03.055.

[14] S. K. Izadi, T. Härder, and M. S. Haghjoo, "S3: Evaluation of tree-pattern XML queries supported by structural summaries," *Data Knowl. Eng.*, vol. 68, no. 1, pp. 126–145, Jan. 2009, doi: 10.1016/j.datak.2008.09.001.

[15] S. Agreste, P. De Meo, E. Ferrara, and D. Ursino, "XML matchers: Approaches and challenges," *Knowl.-Based Syst.*, vol. 66, pp. 190–209, Aug. 2014, doi: 10.1016/j.knosys.2014.04.044.

[16] 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. 18th Int. Conf. Data Eng.*, San Jose, CA, USA, Feb./Mar. 2002, pp. 141–152, doi: 10.1109/ICDE.2002.994704.

[17] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: Optimal XML pattern matching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: Association Computing Machinery, 2002, pp. 310–321, doi: 10.1145/564691.564727.

[18] S. Chen, H. G. Li, J. Tatemura, W. P. Hsiung, D. Agrawal, and K. S. Candan, "Twig2Stack: Bottom-up processing of generalized tree-pattern queries over XML documents," in *Proc. 32nd Int. Conf. Very Large Databases (VLDB)*, 2006, pp. 283–294.

[19] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, and D. Srivastava, "Index structures for matching XML twigs using relational query processors," *Data Knowl. Eng.*, vol. 60, no. 2, pp. 283–302, Feb. 2007, doi: 10.1016/j.datak.2006.03.003.

[20] L. Qin, J. X. Yu, and B. Ding, "TwigList: Make twig pattern matching fast," in *Proc. 12th Int. Conf. Database Syst. Adv. Appl.* Jeju, South Korea: Springer, 2007, pp. 850–862.

[21] H. A. Al-Jamimi, A. F. Barradah, and S. Mohammed, "Siblings labeling scheme for updating XML trees dynamically," in *Proc. Int. Conf. Comput. Eng. Technol.*, 2012, pp. 21–25.

[22] S.-C. Haw and C.-S. Lee, "Extending path summary and region encoding for efficient structural query processing in native XML databases," *J. Syst. Softw.*, vol. 82, no. 6, pp. 1025–1035, Jun. 2009, doi: 10.1016/j.jss.2009.01.007.

[23] J. Liu and X. X. Zhang, "Dynamic labeling scheme for XML updates," *Knowl.-Based Syst.*, vol. 106, pp. 135–149, Aug. 2016, doi: 10.1016/j.knosys.2016.05.039.

[24] J.-K. Min, J. Lee, and C.-W. Chung, "An efficient XML encoding and labeling method for query processing and updating on dynamic XML data," *J. Syst. Softw.*, vol. 82, no. 3, pp. 503–515, Mar. 2009.

[25] X.-T. Nguyen, S.-C. Haw, S. Subramaniam, and C.-K. Pham, "Dynamic node labeling schemes for XML updates," in *Proc. 6th Int. Conf. Comput. Inform.*, 2017, pp. 505–510.

[26] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Updating XML," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2001, pp. 413–424, doi: 10.1145/376284.375720.

[27] Z. Brahmia, H. Hamrouni, and R. Bouaziz, "TempoX: A disciplined approach for data management in multi-temporal and multi-schema-version XML databases," *J. King Saud Univ.-Comput. Inf. Sci.*, to be published, doi: 10.1016/j.jksuci.2019.08.009.

[28] S. Maneth and F. Peternek, "Grammar-based graph compression," *Inf. Syst.*, vol. 76, pp. 19–45, Jul. 2018, doi: 10.1016/j.is.2018.03.002.

[29] 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. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2002, pp. 204–215, doi: 10.1145/564691.564715.

[30] S.-C. Haw, S. Subramaniam, W.-S. Lim, and F.-F. Chua, "Hybridation of labeling schemes for efficient dynamic updates," *Indonesian J. Electr. Eng. Comput. Sci.*, vol. 4, no. 1, pp. 184–194, 2016, doi: 10.11591/ijeecs.v4.i1.

[31] J. Liu, Z. M. Ma, and L. Yan, "Efficient labeling scheme for dynamic XML trees," *Inf. Sci.*, vol. 221, pp. 338–354, Feb. 2013, doi: 10.1016/j.ins.2012.09.036.

[32] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, "ORDPATHs: Insert-friendly XML node labels," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2004, pp. 903–908, doi: 10.1145/1007568.1007686.

[33] B. Zhang, Z. Geng, and A. Zhou, "SIMP: Efficient XML structural index for multiple query processing," in *Proc. 9th Int. Conf. Web-Age Inf. Manage.* Washington, DC, USA: IEEE Computer Society, Jul. 2008, pp. 113–118.
[34] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting local similarity for indexing paths in graph-structured data," in *Proc. 18th Int. Conf. Data Eng.*, San Jose, CA, USA, Feb./Mar. 2002, pp. 129–140, doi: 10.1109/ICDE.2002.994703.
[35] T. Milo and D. Suciu, "Index structures for path expressions," in *Database Theory—ICDT'99* (Lecture Notes in Computer Science), vol. 1540. Berlin, Germany: Springer-Verlag, 1999, pp. 277–295.
[36] I.-E. Liao, W.-C. Hsu, and Y.-L. Chen, "An efficient indexing and compressing scheme for XML query processing," in *Networked Digital Technologies* (Communications in Computer and Information Science), vol. 87, F. Zavoral, J. Yaghob, P. Pichappan, and E. El-Qawasmeh, Eds. Berlin, Germany: Springer, 2010, doi: 10.1007/978-3-642-14292-5_8.
[37] H. Fan, Z. Ma, D. Wang, and J. Liu, "Handling distributed XML queries over large XML data based on MapReduce framework," *Inf. Sci.*, vol. 453, pp. 1–20, Jul. 2018, doi: 10.1016/j.ins.2018.04.028.
[38] S. Subramaniam, S.-C. Haw, and L.-K. Soon, "DGReLab+: Improving XML path query processing by avoiding buffering irrelevant results," *Procedia Comput. Sci.*, vol. 115, pp. 804–811, Dec. 2017, doi: 10.1016/j.procs.2017.09.157.
[39] S. Subramaniam, S.-C. Haw, L.-K. Soon, and K.-L. Koong, "QTwig: A structural join algorithm for efficient query retrieval based on region-based labeling," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 27, no. 2, pp. 321–342, Mar. 2017, doi: 10.1142/S0218194017500115.
[40] F. Azzedin, S. Mohammed, M. Ghaleb, J. Yazdani, and A. Ahmed, "Systematic partitioning and labeling XML subtrees for efficient processing of XML queries in IoT environments," *IEEE Access*, vol. 8, pp. 61817–61833, 2020, doi: 10.1109/ACCESS.2020.2984600.
[41] (Jan. 24, 2017). *XQuery Update Facility 3.0, W3C Working Group Note*. [Online]. Available: https://www.w3.org/TR/xquery-update-30/
[42] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient filtering of XML documents with XPath expressions," *VLDB J. Int. J. Very Large Data Bases*, vol. 11, no. 4, pp. 354–379, Dec. 2002, doi: 10.1007/s00778-002-0077-6.
[43] Y. Diao, P. Fischer, M. J. Franklin, and R. To, "YFilter: Efficient and scalable filtering of XML documents," in *Proc. 18th Int. Conf. Data Eng.*, San Jose, CA, USA, Feb./Mar. 2002, pp. 341–342, doi: 10.1109/ICDE.2002.994748.

WEN-CHIAO HSU received the B.S. degree in international trade from Chinese Culture University, Taiwan, R.O.C., in 1993, the M.S. degree in computer information systems from the Florida Institute of Technology, USA, in 2003, and the Ph.D. degree in computer science and engineering from National Chung Hsing University, Taichung, Taiwan, in 2012. She is currently an Assistant Professor with the Department of Information Management, National Taichung University of Science and Technology, Taiwan. Her research interests include database systems, data mining, extensible markup language databases, and recommended systems.

I-EN LIAO received the B.S. degree in applied mathematics from National Chengchi University, Taiwan, in 1978, and the M.S. degree in mathematics and the Ph.D. degree in computer and information science from The Ohio State University, in 1983 and 1990, respectively. He is currently a Professor with the Department of Computer Science and Engineering, National Chung Hsing University, Taiwan. He also leads a Research Team with the Taiwan Information Security Center at NCHU (TWISC@NCHU) working on the design and implementation of secure and resilient mechanisms for critical infrastructure information protection. His research interests include data mining, extensible markup language databases, big data analytics, and information security. He is a member of ACM and the IEEE Computer Society.