An Efficient Prefix-based Labeling Scheme for XML Dynamic Updates using Hexagonal Pattern

Amjad Qtaish1, Jalawi Alshuduki2

1,2 College of Computer Science and Engineering, University of Ha'il, Ha'il, 55473 Saudi Arabia

Corresponding author: Amjad Qtaish (e-mail: am.qtaish@uoh.edu.sa).

ABSTRACT To improve XML query processing, it is necessary to label XML documents efficiently for the indexing process because it allows the structural relationships between the XML nodes to be preserved without having to access the original document. However, XML data on the Web is updated as time passes, which means that the dynamic updating of XML data is an issue that may need to be handled by a XML labeling scheme specifically designed for dynamic updates. Previous XML labeling schemes have limitations when updates take place. For example, a lot of node labels need to be relabeled, a lot of duplicate labels occur during this relabeling process, and the size and time costs of the updated labels are high. Therefore, this paper proposes an efficient prefix-based labeling scheme that uses a hexagonal pattern. The proposed labeling scheme has three main advantages: (i) it avoids the need for node relabeling when XML documents are updated at random locations, (ii) it avoids duplicated labels by creating a new label for every inserted node, and (iii) it reduces the size and time costs of the updated labels. The proposed scheme is evaluated against the three most recent prefix-based labeling schemes in terms of the size and time costs of the updated labels. In addition, the ability of the proposed labeling scheme to handle several updates (such as insertions) in XML documents is also evaluated. The evaluations show that the proposed labeling scheme outperforms previously developed prefix-based labeling schemes in terms of both size and time costs, particularly for large-scale XML datasets, resulting in improved query processing performance. Moreover, the proposed scheme efficiently supports frequent updates at arbitrary positions. The paper concludes with several suggestions for further research.

© 2017 Elsevier Inc. All rights reserved.

INDEX TERMS Dewey labeling, Dynamic update, Prefix labeling, XML.

I. INTRODUCTION

For a long time, XML, or extensible markup language, has become the Web's de facto standard for data exchange and representation [1-3]. Indeed, XML has attracted broad acceptance and support from all of the major providers of databases, servers, and software tools [4, 5]. XML is a hierarchical, self-describing semi-structured data format that employs an easy-to-write and easy-to-parse syntax to simplify data interchange across a variety of Web-based applications [5-7]. It provides communication across multiple computing systems, which was previously extremely difficult, if not impossible. Therefore, XML provides a global format for data interchange, regardless of the platforms and data models utilized by the applications.

Hence, the computing world now has a considered trying method for developing distributed system applications, as well as Data may be exchanged across nearly any hardware, software, or operating system because to XML’s flexibility [2, 4]. However, XML is used to represent a large amount of semi-structured data on the Internet. As a result, handling these data in terms of storage, updating, and querying has become a big challenge [2, 7-10].

One answer to this challenge is the labeling scheme, which assigns a number (label) to each node to represent the identifier or the absolute position of the node in the document [11-16]. It maintains the structural relationships among the nodes such as those between parent and child (P-C), ancestor and descendant (A-D), and siblings [1-3]. This improves XML data storage, updates, and queries. [7, 17,
The adoption of a labeling scheme improves the efficiency of XML data retrieval and indexing because its usage results in a reduction in the query processing time [19, 20]. In a labeling scheme, labels are used to preserve relationships between nodes in XML documents as well as to facilitate in querying. [16, 21-24]. Hence the efficient response to data queries depends largely on XML labeling schemes, keyword searches and XML data queries [25, 26].

When a labeling scheme is used, an XML query is similar to a relational database (RDB) query, which also depends on the indexing of nodes [1, 27, 28]. In addition, an ordered XML tree and structural relationships are encoded into the index. Therefore, the order of these rows and columns is not crucial in a RDB, but it is for text-centric documents such as e-books, e-journals, and e-mails. To address this issue, many labeling schemes have been developed. However, these schemes still have drawbacks in terms of their poor ability to efficiently support the dynamic updating of XML documents [1, 3, 29, 37].

In regard to the support provided by labeling schemes for the dynamic updating of XML data, the challenges include the duplicating of labels and the relabeling of old nodes [1, 11, 34-37]. Two other key challenges are the time needed by labeling schemes to relabel XML data and the resulting size of the labels generated [22, 38, 39]. A big label size might have an adverse influence on query and update performance [40, 41]. There are two forms of skewed insertion that can occur when updating new XML nodes. The first is known as ordered skewed insertion, where nodes are inserted frequently before or after a certain node. The second is termed random skewed insertion, where nodes are inserted frequently between two random nodes. However, in another form, the inserting node is uniform, in which the node is added between random pairings of consecutive nodes [42, 43]. Some labeling schemes have been evaluated only in respect of skewed insertions [1, 44, 45]. Generally, four types of node insertion are considered by labeling schemes: insertion before the leftmost sibling, insertion after the rightmost sibling, insertion between two siblings, and insertion of a child into a leaf node [34, 39, 42, 46-50].

Existing labeling schemes can be classified into four main groups [1, 3, 30, 51]: interval-based labeling schemes (also known as range-based labeling schemes) [11, 23, 42, 52-56], prefix-based labeling schemes [1, 11, 13, 20, 30, 34, 46, 57], multiplicative-based labeling schemes [48, 53, 58-64], and hybrid-based labeling schemes [11, 29, 30, 65, 66]. However, regardless of the type of scheme, an efficient labeling scheme should possess four main properties. Firstly, its labeling must be dynamic, i.e., it must avoid relabeling XML nodes during XML documents updates. Second, the amount of time it takes to create labels initially and after dealing with skewed node insertions must be as short as possible. Third, the size of XML label must be less, i.e., the labeling scheme must produce compact labels that lead to reduced storage cost for both the initial labeling and after the skewed node instructions have been dealt with [39, 45, 67, 68]. Fourthly, the labeling scheme must support all types of queries because labels play an important role in efficient query and answer processing in XML enabled databases and native XML databases [2, 20, 69, 70].

In this paper, an efficient prefix-based labeling scheme is proposed that uses a hexagonal pattern to support dynamic updates in XML documents. The proposed labeling scheme is designed based on previous prefix-based labeling schemes for generating labels [1, 11, 13, 20, 30, 34, 46, 57]. The proposed labeling scheme has three major advantages, and its novelty lies with: First, preventing node relabeling when XML documents are often updated at random locations. Second, avoiding duplicated labels by creating a new label for every inserted node. Third, reducing the size and time costs of the updated labels. The proposed labeling scheme is compared to various prefix-based labeling schemes in terms of the size of the produced labels, the time required for the labeling process, and its ability to process many cases of updates.

The remainder of this work is structured as follows: First, Section II presents the related work. Next, Section III presents the basic operators that support XML updates. Then, Section IV goes into detail about the proposed labeling scheme. Section V describes the experimental design. Section VI analyzes and discusses the experimental results in terms of label size and labeling process time. Finally, Section VII concludes with a summary and some recommendations for further work.

II. RELATED WORK

A. INTERVAL-BASED LABELING SCHEME

The interval-based labeling scheme (also known as the range-based labeling scheme) is characterized by incorporating <start, end> arguments into the labels [3, 71]. Using these arguments, the start, and the end of the corresponding position of the nodes in the XML tree are determined [1, 3, 23]. Therefore, this labeling scheme efficiently preserves the P-C and A-D relationships among the nodes [1, 3]. However, it is difficult to maintain the lowest common ancestor (LCA) [1]. In addition, it does not provide efficient support for dynamic XML documents [1, 3, 56]. This because any insertion of new nodes may incur the relabeling of numerous nodes, which increases the time...
taken to label the nodes [1, 52]. Hence, using this type of scheme, the labeling time and size are both high, especially for larger XML documents [23, 56].

Nevertheless, many interval-based labeling schemes for XML documents have been proposed. Zhang et al. [72] adopted a region-based labeling approach to build their labeling scheme, in which each node in the XML document is handled by two sets of values (Begin, End). The first value, “Begin”, is the sequence number that is assigned to the node when it is visited for the first time by the parser, i.e., when the tag is opened. The second value, “End”, is the sequence number that is assigned to the node when it is visited for the second time, i.e., when the tag is closed. For example: If the region of node $u$ is inside the region of the other node $v$, then $v$ is the ancestor of $u$. This property is termed the containment property [50, 56]. By comparing the values, the structural and sequential relationships among the nodes can be determined by this labeling scheme in a rapid manner. However, when new nodes are inserted, other nodes must be relabeled, resulting in reduced updating efficiency [29]. Moreover, because this labeling scheme visits XML nodes twice to generate a label for each node, the labeling time and label size are both high, especially when the XML document grows in size [56]. Figure 1 below illustrates how, in the interval-based labeling scheme, the insertion of a node $a$ incurs relabeling of other nodes (within the dotted box in the figure).

Li and Moon [52] presented another region-based labeling approach in which each XML document node is assigned to two sets of values (Order, Size). The first value, “Order”, is the unique identifier of the XML document's preorder traversal. The second value, “Size”, represents the number of descendant nodes. By utilizing the redundancy of “Size”, this labeling scheme, rather than the second encoding of other nodes, can offer a partial solution to this issue of new nodes being inserted [29]. However, this scheme is not well suited to handling other information such as the sibling and level for retrieval [23].

![Figure 1. Interval-based labeling scheme.](image)

### B. PREFIX-BASED LABELING SCHEME

The prefix-based labeling scheme is similar to the Dewey decimal classification traditionally used by libraries [57]. This labeling scheme can maintain different kinds of structural relationships between nodes such as P-C, A-D, and siblings [73]. In prefix-based labeling schemes, the parent node's label is encoded as a prefix to the self-label node. These labels are created by depth-first tree traversal, and they are separated by a delimiter, either ' ' or '.' [1, 20, 56, 73]. A popular prefix-based labeling scheme known as Dewey encoding was proposed by [20], which is essentially a Dewey order labeling scheme.

The Dewey order labeling scheme is a combination of two order labeling schemes: the global scheme and the local scheme [20]. As shown in Figure 2(a) below, each node in the global order labeling scheme is allocated a number that specifies the node's absolute position in the document. Dynamic updating is extremely difficult with this scheme because all nodes following the inserted node must be relabeled and determining the P-C and A-D relationships is almost impossible. As illustrated in Figure 2(b) below, each node in the local order labeling scheme is allocated a number that denotes its relative position among its siblings. Under this scheme, a node's actual position inside the document is determined by combining its position with that of its ancestors as a path vector. Because only the siblings of the new node need to be relabeled, an update in the local labeling scheme has less overheads than an update in the global labeling scheme [37, 74]. However, obtaining the P-C and A-D relationships remains extremely challenging.

On the other hand, in Dewey order labeling scheme, each node is assigned a label based on Dewey decimal classification. Each node is given a label that indicates the root-to-node pathway in the XML document. Consequently, each part of its path indicates the local order of an ancestor node, as illustrated in Figure 2(c) below. Label 1.2.5, for example, denotes node #5 in level 3, whose parent is node #2 in level 2, and whose ancestor is the document's root. Hence this scheme provides a straightforward way to extract node labels from ancestors. When inserting a new node, however, all the sibling nodes to the right of the inserted node, as well as their descendants, must be relabeled. Because of its simplicity, the Dewey order labeling scheme [20] has gained popularity among indexing schemes [75-78]. However, it is unsuitable with dynamic XML data. For example, to insert a new node $a$ between two sibling nodes in an XML tree, as shown in Figure 2(c), the Dewey order labeling scheme necessitated the relabeling of all the right sibling nodes of node $a$ as well as their descendants (within the dotted box part of the figure) [78]. However, due to its advantages, numerous labeling schemes have been proposed based on prefix-based labeling scheme. In [49], ORDPATH scheme was designed based prefix-based labeling scheme, which processes structural queries efficiently based on the preceding label and the current label of a node.
For initial labeling, it assigns a distinct label for each node in the XML document based on the ancestor and parent labels, followed by an odd and positive number. Each integer is separated by “.” to distinguish between the ancestor and the descendant labels. For updating or insertions into an existing document, it uses even and negative integer labels. The drawback of ORDPATH is that it only allows for a limited number of insertions, which increases the complexity of decoding for the inserted labels [30, 34, 78]. Therefore, overall, it has an adverse impact on the efficiency with which XML queries are processed [34, 79].

On the other hand, two dynamic labeling schemes named dynamic Dewey encoding (DDE) and compact dynamic Dewey encoding (CDDE) were proposed by [34]. The CDDE scheme is an enhanced version of the DDE scheme, especially designed to improve the performance of DDE for insertions. DDE and CDDE have the same initial labels as the Dewey system proposed in [46] for the insertion between two consecutive siblings “1.1” and “1.3”. However, then DDE and CDDE add together each Dewey component of these two siblings in order to label node a, which becomes “2.5”, i.e., (1+1).(2+3), as shown in Figure 3(a) below. According to the results of a comparison with some previous works, DDE and CDDE show better performance in the case of XML node insertion. However, because the values of the DDE labels grow exponentially when nodes are frequently inserted between two consecutive siblings, as shown in Figure 3(b) below, the labels in DDE and CDDE are not compact, which commonly results in an extra storage cost [1, 46, 59].

According to Figure 3(b), nodes b, c, d, e are labeled 4.4.101, 6.6.152, 10.10.253, and 16.16.405, respectively. These multiple insertions make DDE consume a lot of numbers. This disadvantage can be considerable because it is usual for an XML repository to have many XML documents. In addition, DDE and CDDE are not appropriate for efficiently preserving structural relationships across numerous XML documents [1, 46, 73, 78]. For example, given two labels “2.5” and “2.3”, while it is known that these are siblings in Dewey, this sibling relationship may not be maintained in DDE (see node a labeled 2.5 and 2.3, respectively, in Figure 3(a) and 3(b)). As a result, DDE requires an additional document identifier to distinguish labels across various XML documents.

Consequently, DDE is slower in determining structural relationships in XML query processing, and storage costs rise [1].

Bao, et al. [80] presented the Pathed-Dewey Order, a novel labeling scheme for capturing both path and position information of each node in an XML tree, as well as a double-layered mapping structure for storing documents with flexible schemas into NoSQL databases such as HBase tables. To reduce storage space and query response time, four optimization techniques were developed and implemented. Extensive experiments on two well-known XML benchmarks show that this scheme outperforms three state-of-the-art methods. However, this scheme did not support data updating and inserting operations, efficiency optimization of data mapping and storing, or Pathed-Dewey Order compression to store more encodings in memory.

The dynamic float-point Dewey labeling (DFPD) scheme was proposed to improve the way dynamic queries in XML documents are handled [46]. This labeling scheme generates initial labels based on the Dewey order labeling scheme. It overcomes the limitation of previous schemes in
respect of single insertions in XML data processing. It handles XML updates by considering the same three cases that are covered by the DDE scheme: insertion before the leftmost sibling, insertion after the rightmost sibling, and insertion of a child as a leaf node. However, because the DFPD scheme generates a float-point number for a label when it places the label between two following siblings, this increases the size of the label as well as the time required to calculate the label [1, 78]. In addition, this scheme consumes a lot of numbers by assigning and calculating new labels for the inserted nodes without taking into account the reuse of the deleted labels [1].

The DFPD scheme is shown in Figure 4(a) below, where new nodes inserted in XML trees are shown by the dotted circles that contain alphabetical letters. The order of inserting nodes is based on the alphabetical order of these letters. According to the DFPD labeling scheme, node e is inserted between “1.2” and “1.3” and its label is 1.5/2, which is equal to 1.2 + 1)/1 + 1). Node b is inserted between “1.5/2” and “1.3” and its label is 1.8/3, which is equal to 1.5 + 3)/(2 + 1). Node c is inserted between “1.5/2” and “1.8/3” and its label is 1.13/5, which is equal to 1.5 + 8)/(2 + 3). Node d is inserted between “1.13/5” and “1.8/3” and its label is 1.21/8, which is equal to 1.13 + 8)/(5 + 3). When insertion and deletion in the XML document occur alternately, the values of the DFPD components rapidly grow. As a result, the newly inserted labels in DFPD become less compact, resulting in an increased requirement for storage [1, 78].

An enhanced version of the DFPD scheme, named the dynamic prefix-based labeling scheme (DPLS), was proposed by [1]. The initial labeling in the DPLS is based on the Dewey order labeling scheme [20]. Each label in the DPLS uses a series of components to describe an unique path from the root that is labeled with a non-zero number, which is represented as 1 until it reaches a node. Like the DFPD scheme, the DPLS uses a fraction fragment (i.e., a floating-point number) to assign the last part of the label. This is a drawback because it leads to limited accuracy [71, 78]. However, the DPLS has some advantages when compared to the DFPD scheme: (i) it reduces the query costs, (ii) it avoids the relabeling process under various scenarios, and (iii) it can reuse deleted node labels. In DPLS, four cases of insertions are proposed, which are: insert leftmost (node a), insert rightmost (node b), insert leaf (node c), insert between siblings (node d, Node e, node f, node g).

Figure 4(b) below illustrates the insertion process of the DPLS, where new nodes inserted into XML tree are presented by the dashed circles and lines. Node a is inserted to the leftmost (before the first child of the root), thus, it is labeled as 1.0 (subtracting 1 to the local order of 1.1). Node b is inserted to the rightmost (after the last child of the root), thus, it is labeled as 1.4 (adding 1 to the local order 1.3). Node c is inserted as the child of the node labeled 1.2, thus, it is labeled as 1.2.1 (adding a new component with the number 1). Nodes d, e, f, and g are inserted in alphabetical order between 1.2 and 1.3, thus, they are labeled as 1.5/2, 1.8/3, 1.13/5, and 1.21/8, respectively.

C. OTHER LABELING SCHEMES

A number of other labeling schemes of note have also been proposed. For example, the dynamic prefix encoding scheme based on fraction (DPESF) was proposed by [14] in an attempt to further improve performance. This scheme improves time and storage performance, while supporting the dynamic updating process. In general, the DPESF and the DPLS are similar except that the DPESF represents the numerators by using a collection of alphabet letters. This scheme involves a number sequence [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] as well as a set of alphabet letters [A, B, C, D, E, F, G, H, I, J]. The equivalent rule function f is therefore defined as [(0, A), (1, B), (2, C), (3, D), (4, E), (5, F), (6, G), (7, H), (8, I), and (9, J)]. According to f, a label like 123/11 is represented as BCD11, where BCD equals 123. Like the DPLS, four cases of insertions are considered in the DPESF, namely, insert leftmost, insert rightmost, insert leaf, and insert between two siblings. For example, node a is inserted between two siblings with labels 1.1.2.1 and 1.1.2.2 and its label is 1.1.2.2.((1+2)/(1+1)) = 1.1.2.3.2. As a result, the “3” is substituted with a “D”, yielding the label 1.1.2.2.D2. Similarly, the label of node b is 1.1.2.(3+2)/(2+1)) = 1.1.2.5.3, which is equivalent to 1.1.2.F3. On the other hand, other cases of insertions are handled in the same way as in the DPLS without a fractional number being generated. However, it is not clear whether the DPESF achieves improvements in the label size and the updating time when compared with the ORDPATH scheme [38].

![Figure 4](image-url)
Ko and Lee [79] proposed an improved binary string labeling (IBSL) scheme. This scheme uses a string-based binary encoding approach, with the IBSL label consisting of a sequence of digits 0 and 1. When updating XML documents, the IBSL scheme avoids the requirement for relabeling by reusing the deleted label in the same position. However, in the case of frequently skewed insertions, it raises the cost of storage [1, 78]. Furthermore, the IBSL scheme exclusively evaluates leaf node insertions. Ghaileb and Mohammed [13] proposed a dynamic XDAS, which is a combination of the original XDAS [81] and the IBSL scheme [79]. The XDAS generates labels based on a lexical order with masking techniques. For example, the label of node $a$ is 1,001, where the first part denotes the level, followed by the self-label ID 001, which is called the first child. Node $b$, which is the sibling of node $a$, will have the label 1,010 and so on. On the other hand, the label of the first child node of node $a$ is 2,01001, where 2 is the level, 01 is the self-label ID, and 001 is the label of its parent node. The XDAS significantly reduces the label size, but the updating time is not improved when compared with that of the IBSL scheme [38, 11].

A quaternary encoding scheme (QED) for dynamic XML documents is proposed [47]. QED label is a sequence of numbers in the range "0", "1", "2", and "3", which are each stored with two bits. The number "0" is used as a separator in QED scheme, and the numbers "1", "2", and "3" are utilized in the QED label itself. In [20], a new labeling scheme, called Reuse is proposed based on the QED scheme. When XML updates occur, the reuse technique employs all deleted labels to handle the increasing label size and enhance query performance. In addition, SCOOTER is a dynamic encoding scheme proposed by O’Connor and Roanetre [45] for handling updates in XML documents that employs the quaternary encoding used in QED and allows the reuse of shorter deleted node labels. However, when new nodes are skewed insertions, the lengths of QED labels extensively [34]. According to the QED, when new nodes are skewed insertions, the lengths of the IBSL scheme, enhanced binary string labeling (EBSL) and dynamic XDAS, as well as SCOOTER labels, would increase rapidly. Furthermore, the IBSL, EBSL, dynamic XDAS, and QED schemes do not allow for the reuse of deleted labels, which has a negative impact on both storage and updating performance [1, 34].

Dhanalekshmi and Krishna [82] proposed a lexicographic-based persistent labelling (LPLX) scheme based on the prefix-based labeling scheme. LPLX assigns a distinct label to each node in the XML tree, which includes three sections (Prefix, Level, Selfcode). The label of the parent node is shown by the Prefix section. The Prefix section includes strings that are a collection of letters and digits. The Level section is an integer that shows the node’s depth from the root of the XML tree, while the Selfcode section specifies the node’s self-label. Each node in LPLX is labeled using a combination of digits (0–9) and uppercase letters (A–Z). LPLX supports dynamic XML document updates without relabeling existing nodes. Furthermore, structural relationships such as A-D, P-C, and siblings have been calculated across time. Furthermore, experiments were carried out on XML data sets with varying node counts and depths. However, the lengths of LPLX labels vary greatly in the case of frequently skewed insertions in large, deep XML documents. This is because the LPLX scheme adds a new letter to the end of the Selfcode section for insertions; for example, if the final letter in Selflabel is "Z", it adds a new letter beginning with "A", and so on. Moreover, the LPLX scheme has not been evaluated with many of the existing labeling schemes.

Another labeling scheme was proposed by [29], which was named the prime-based middle fraction labeling scheme (PMFLS). This scheme is built based on a set of algorithms that preserve structural relationships between nodes while also supporting for XML updates. The PMFLS is a hybrid labeling scheme, which means it combines a prefix-based and an interval-based (region-based) labeling scheme. Order-sensitive updates are also supported by the PMFLS without the need for recalculation. However, however, when XML data is updated frequently during insertions, the size of prefix labels may grow, producing overflow issues [78]. Another scheme, named the ReLab scheme, was designed and proposed based on the region-based labeling scheme [23]. In regard to the amount of time required to create labels for each XML node, the experimental evaluation of the ReLab scheme showed that, it outperformed other region-based schemes like Dietz and region. The Relab scheme is used not just for XML node unique identification, but also for structural relationship purposes. It processes XML queries efficiently using the depth-first traversal approach. Another labeling scheme known as the ME scheme, is a robust hybrid scheme for dynamic updates in XML databases [53]. The ME scheme determines and maintains structural relationships among XML nodes, as well as supporting dynamic updates without the need to relabel nodes in the case of an update. The ReLab scheme’s simplicity allows it to produce labels more rapidly than existing region-based schemes [65, 78]. The drawback of ReLab is that it only supports static XML documents, not dynamic XML data [1, 65, 78].

The research [83] proposed a new prefix based labelling scheme which is compact, dynamic, maintaining the structural relationships among XML nodes for improving the query processing. The proposed scheme can handle both static as well as dynamic XML documents. The experiments are carried out to evaluate the proposed scheme in terms of its performance of storage requirement, structural relationship computation, and update processing. The result is compared with some of the existing labelling mechanisms.
From the above, many recent labeling schemes have been proposed to support XML updates efficiently. However, these labeling schemes have many limitations: (i) some of them do not support dynamic updates [20, 23, 25], (ii) some have been evaluated only on a limited number of insertions [49], (iii) some generate a large label size, which increases the storage and labeling time, especially when frequent and skewed insertions occur between two siblings [25, 34, 45, 48], and (iv) others have a negative impact on the efficiency of XML query processing [1, 45, 49]. Table I summarizes the focus and contributions of the existing labeling schemes that are reviewed above.

In light of the above, this paper aims to propose a novel prefix-based labeling scheme, which prevents node relabeling when the XML documents are frequently updated at arbitrary positions, avoids duplicated labels by creating a new label for every inserted node, reduces the size and time costs of the updated labels, and supports dynamic updates. In addition, the proposed labeling scheme is implemented to maintain the structural information among the nodes based on four different insertions cases, namely, insertion before the leftmost sibling, insertion after the rightmost sibling, insertion between two siblings, and insertion of a child into a leaf node. Moreover, the proposed labeling scheme is evaluated by comparing it with the most recent labeling schemes in terms of label size, labeling time, and its ability to handle several insertions in XML documents. Furthermore, the proposed labeling scheme is intended to provide random skewed updates with fast computations and has a simple implementation.

![Diagram of XML nodes and insertions](image)

**FIGURE 5.** Primitive node insertions.

| Existing Labeling schemes | Focus and Contribution(s) | Limitations of the Scheme |
|---------------------------|---------------------------|---------------------------|
| Zhang et al. [72]         | It determines the structural and sequential relationships among the nodes in a rapid manner. | The nodes need to relabel, resulting in reduced updating efficiency. The labeling time and label size are both high, especially when the XML document grows. |
| Li and Moon [52]          | It provides a partial solution to the issue of new nodes being inserted. | It is not well suited to handling other information such as the sibling and level for retrieval. Dynamic updating is extremely difficult. All nodes need to relabel. It has negative performance in obtaining the P-C and A-D relationships. It is unsuitable with dynamic XML data. |
| Dewey Order [20]          | It has become popular among indexing schemes because of its simplicity. | It only allows for a certain number of insertions. It has a negative effect on the efficiency of XML query processing. The labels in DDE and CDDE are not compact, which affects storage cost. They are not appropriate for efficiently preserving structural relationships across numerous XML documents. |
| ORDPATH [49]              | It processes structural queries efficiently based on the preceding label and the current label of a node. | It consumes a lot of numbers by assigning and calculating new labels for the inserted nodes without taking into account the reuse of the deleted labels. The newly inserted labels in DFPD become less compact, resulting in higher storage cost. |
| DDE & CDDE [34]           | They show better performance comparing with some previous works in the case of XML node insertion. | It uses a fraction fragment (i.e., a floating-point number) to represent the last component of the label, which leads to limited accuracy. As compared to DFPD, label size and update time of DPLS were slightly improved. |
| DFPD [46]                 | It overcomes the limitation of previous schemes in respect of single insertions in XML data processing. It handles XML updates. | It is not clear whether the DPESF achieves improvements in the label size and the updating time when compared with the ORDPATH scheme. |
| DPLS [1]                  | It reduces the query costs. It avoids the relabeling process under various scenarios. It can reuse deleted node labels. It outperforms several state-of-art labeling schemes in terms of label size and updating time. This is because the deleted labels are reused for encoding new inserted nodes. | They have high storage cost in the case of frequently skewed insertions. They exclusively use leaf node labels. The lengths of the IBSL labels would increase rapidly when new nodes are |
| DPESF [14]                | It improves time and storage performance, while supporting the dynamic updating process. | |
| IBSL & EBSL [79]          | They avoid the requirement for relabeling by reusing the deleted label in the same position when updating | |

**TABLE I.** SUMMARY OF EXISTING LABELING SCHEMES
III. XML LABEL UPDATES

The issues associated with XML label updates are presented in this section. The main question that needs to be addressed is as follows: How can the labeling scheme maintain the structural relationships among nodes when XML updates occur? Updates on XML trees can be defined by using node deletion and insertion operations. As regards node insertions, the XML data as an ordered and directed tree is defined and called X, and four primitive insertions are determined to keep the structural relationships among the nodes with aim of avoiding a relabeling cost and reducing the size and time costs of the updated labels. These primitive insertions are insertion before the leftmost sibling, insertion after the rightmost sibling, insertion between two siblings, and insertion of a child into a leaf node. So, let \( M, P, \) and \( N \) be three nodes, \( X \) be an order and directed XML tree, \( L \) be a label for an inserted node \( c \) which is the parentLabel(c) || selfLabel(c), and the primitive node insertions be given as follows:

- \( \text{insertLeftmost}(X, M, N, L) \): insertion of a new node \( M \) to the left and before the first sibling node \( N \) in \( X \), and its label \( L \) which contains parentLabel(M) || selfLabel(M).
- \( \text{insertRightmost}(X, N, M, L) \): insertion of a new node \( M \) to the right and after the last sibling node \( N \) in \( X \), and its label \( L \) which contains parentLabel(M) || selfLabel(M).
- \( \text{insertLeaf}(X, M, N, L) \): insertion of a new node \( M \) as a child of a node \( N \) in \( X \), and its label \( L \) which contains parentLabel(M) || selfLabel(M).
- \( \text{insertSibling}(X, M, P, N, L) \): insertion of a new node \( P \) between two sibling nodes \( M \) and \( N \) in \( X \), and its label \( L \) which contains parentLabel(P) || selfLabel(P).

The lengths of QED labels extensively when new nodes are skewed insertions. It also does not allow for the reuse of deleted labels, which has a negative impact on both storage and updating performance. The lengths of SCOOTER labels would increase rapidly when new nodes are skewed insertions. It does not allow for the reuse of deleted labels, which has a negative impact on both storage and updating performance. The lengths of LPLX labels extensively in the case of frequently skewed insertions in large XML documents with high depth. It has not been evaluated with many of the existing labelling schemes. The lengths of PMFLS labels may grow, producing overflow issues, when XML data is updated frequently during insertions. It only supports static XML documents, not dynamic XML data.

The inserted nodes in Figure 5 are shown by dotted circles, which represent basic instances of primitive node insertions in a simple tree. When deleting a node, consider the following four primitive deletions: (i) deletion of a node to the left of all existing children of a node, (ii) deletion of a node to the right of all existing children of a node, (iii) deletion of a leaf node, and (iv) deletion of a middle node between two siblings. Obviously, these four node deletions will not generate new nodes into XML documents; therefore, no new labels for the deleted nodes are required. Furthermore, because the labels before the deletions are distinct, the deletion has no effect on the relationship (or order) of other labels, and the labels after the deletions remain distinct. In other words, utilizing the encoding scheme, the structural relationship between nodes can still be identified by directly comparing their labels. As a result, deletions have no effect on the labeling scheme's effectiveness [1]. As deletions do not affect the effectiveness of the labeling scheme, this means that the real challenge that needs to be overcome by the proposed labeling scheme is the efficient handling of insertions. Therefore, the focus of this study shifted from finding a way to deal with the processing of updates to instead explore how to effectively manage insertions.
IV. PROPOSED LABELING SCHEME

In this section, we propose an efficient prefix-based labeling scheme to improve the label size and labeling time when updates occur in XML trees. The proposed labeling scheme is designed based on the hexagonal pattern theorem [84] for insertions when XML trees are updated. The Hexagonal numbers are those indicating a collection of objects which can be arranged in the form of a regular hexagon (i.e., all the sides are the same length, and all the vertices have the same angle). The nth hexagonal number \( h(n) \) is the number \( (n) \) of vertices in the form of regular hexagon [85]. The nth hexagonal number's formula is

\[
h(n) = n(2n - 1)
\]  
(1)

where \( n = 1, 2, 3, 4, \ldots \) etc.

According to Equation (1) above, when \( n \) is a sequence of numbers (original numbers) starting from 1 to 20, the equivalent hexagonal numbers will be 1, 6, 15, 28, 45, 66, 91, 120, 153, 190, 231, 276, 325, 378, 435, 496, 561, 630, 703, 780. In the proposed scheme, it is assumed that the nodes are synchronized with the prefix-based labeling scheme (i.e., the original number) and \( h \) is the hexagonal function for generating the pattern can be represented by \( (n, h) \), where \( n \) is synchronized with the prefix-based labeling scheme (i.e., the original number) and \( h \) is the hexagonal function for generating the hexagonal number. An overview of the architecture framework for proposed labeling scheme is shown in Figure 6 below. The initial labeling and the handling of updates in the proposed scheme are described in the following two subsections.

A. INITIAL LABELING

The initial labeling in the proposed scheme is based on the Dewey ordering labeling scheme. Each label is a series of numbers, which are separated by dots and represent a distinct path from a non-zero number labeled root to a node itself. However, the proposed labeling scheme differs from the Dewey order labeling scheme in that the nodes in the proposed scheme are labeled with non-zero numbers and non-hexagonal numbers. As shown in Figure 7 below, the root node is given the label value "1" and is known as the parent label. A series of numbers separated by dots are known as the child node labels. For example, assume that node \( M \) has the label \( m_1, m_2, \ldots, m_i \) in the XML tree, thus \( m_1, m_2, \ldots, m_i \) are assigned the labels of its children, whereas, \( i \) in the initial label is non-hexagonal number, so we cannot begin with \( i = 1 \). The label value "1.2" is assigned to the first child of the root node. The label values "1.3", "1.4", "1.5", and "1.7" are assigned to the next children, respectively. The label value "1.3.2" is assigned to the first child of node "1.3" and so on. The aim is to preserve the hexagonal numbers so that they can be used to support XML updates, especially when new nodes are inserted. Hence, the relabeling of nodes is avoided.

![Figure 7. Initial labeling in the proposed scheme.](image-url)

**FIGURE 7. Initial labeling in the proposed scheme.**

**Definition 1.** Suppose there are two node labels, \( M: m_1, m_2, m_3, \ldots, m_i \) and \( N: n_1, n_2, n_3, \ldots, n_y \), and a subtree containing the node \( N \) has \( M \) node as its root. If there is a connected path of nodes from the root \( M \) to the node \( N \) and node \( M' \)'s self-label is \( m_x \), and the node \( N' \)'s self-label is \( n_y \), where \( x < y \), \( m_1 = n_1, m_2 = n_2, \) and \( m_3 = n_3 \), then node \( M \) is an ancestor of node \( N \) and node \( M \) is a descendant of \( M \).

**Definition 2.** Suppose there are two node labels, \( M: m_1, m_2, m_3, \ldots, m_x \) and \( N: n_1, n_2, n_3, \ldots, n_y \), and node \( M \) is a parent of node \( N \); if \( M \) and \( N \) are directly connected in an XML tree and \( M \) appears exactly one level above \( N \) (i.e., \( m = n - 1 \)), then \( M \) is a parent of node \( N \) and \( N \) is a child of \( M \).

**Definition 3.** Suppose there are two sibling node labels, \( M: m_1, m_2, m_3, \ldots, m_x \) and \( N: n_1, n_2, n_3, \ldots, n_y \). This indicates that both nodes are on the same level and have the same parent in an XML tree (i.e., \( m_1 = n_1, m_2 = n_2, m_3 = n_3, \ldots, m_x = m_y \), and \( x = y \)). In an ordered XML tree, if node \( M \) appears to the left of \( N \), \( M \) is considered a pre-order sibling to node \( N \), whereas \( N \) is called a post-order sibling to node \( M \).
In the proposed scheme, Algorithm 1 is employed to produce the initial labels. The digit '1' is assigned to the root label in the first two lines. The digit '1.2' is allocated to the first child label, which is preceded by a sequence of prefixLabel(M) || selfLabel(M). Because '1' is a hexagonal number, selfLabel(M) cannot begin with '1'. By incrementing the label, the labels of the remaining child nodes are generated. The scheme avoids utilizing hexagonal numbers at this stage.

Algorithm 1 Initial labeling function of the proposed scheme

Input: XML document
Output: initial prefix labels
01 If (M is the root)
02 label(root) = 1
03 Else
04 label(M) = parentLabel(M) || selfLabel(M)
05 child = 1
06 While (child <= child_count(M)) loop
07 n = child
08 y = ((Math.sqrt(8 * n + 1) + 1) / 4)
09 If (y == Math.round(y))
10 n++
11 End If
12 label(child) = label(M) | n
13 child++
14 End loop

B. HANDLING XML UPDATES

This section addresses the issue of dealing with XML updates, especially with regard to a dynamic labeling scheme that handles insertions without relabeling existing nodes. In XML updates, the proposed labeling scheme completely avoids relabeling existing nodes. Labeling schemes generally consider four cases of node insertion [34, 46, 78]. These four cases (see Section III) are considered in the proposed labeling scheme, as follows:

- insertLeftmost(X, M, N, L), which represents inserting a new node M to the left and before the first sibling node N in X, and its label L which contains parentLabel(M) || selfLabel(M).
- insertRightmost(X, N, M, L), which represents inserting a new node M to the right and after the last sibling node N in X, and its label L which contains parentLabel(M) || selfLabel(M).
- insertLeaf(X, M, N, L), which represents inserting a new node M as a child of a node N in X, and its label L which contains parentLabel(M) || selfLabel(M).
- insertSibling(X, M, P, N, L), which represents inserting a new node P between two sibling nodes M and N in X, and its label L which contains parentLabel(P) || selfLabel(P).

The implementations of the above four cases of insertion are done by using the following algorithms:

Algorithm 2 InsertLeftmost function for inserting node M before N

Input: previous node labels and the inserted node M
Output: prefix label for the inserted node M
01 If (M has right siblings only)
02 selfLabel(M) = selfLabel(N) - 1
03 If (selfLabel(M) is a hexagonal number)
04 selfLabel(M) = selfLabel(M) - 1
05 End If
06 label(M) = parentLabel(M) | selfLabel(M)
07 End If
08 Return (label(M))

Algorithm 3 InsertRightmost function for inserting node M after N

Input: previous node labels and the inserted node M
Output: prefix label for the inserted node M
01 If (M has left siblings only)
02 selfLabel(M) = selfLabel(N) + 1
03 If (selfLabel(M) is a hexagonal number)
04 selfLabel(M) = selfLabel(M) + 1
05 End If
06 label(M) = parentLabel(M) | selfLabel(M)
07 End If
08 Return (label(M))

Algorithm 4 InsertLeaf function for inserting leaf node M

Input: previous node labels and the inserted node M
Output: prefix label for the inserted node M
01 If (M is the first child)
02 selfLabel(M) = 2
03 label(M) = parentLabel(M) | selfLabel(M)
04 End If
05 Return (label(M))

Algorithm 5 InsertSibling function for inserting node P between two siblings M and N

Input: previous node labels and the inserted node P
Output: prefix label for the inserted node P
01 If (P between M and N)
02 If (selfLabel(M) and selfLabel(N) are non-hexagonal numbers)
03 Label = selfLabel(M) + selfLabel(N)
04 selfLabel(P) = hexagonal_function(Label)
05 label(P) = parentLabel(P) | selfLabel(P)
06 End If
07 If (selfLabel(M) and selfLabel(N) are hexagonal numbers)
08 Label1 = original_function(selfLabel(M))
09 Label2 = original_function(selfLabel(N))
10 selfLabel(P) = hexagonal_function(Label1 + Label2)
11 label(P) = parentLabel(P) | selfLabel(P)
12 End If
13 If ((selfLabel(M) is hexagonal number) and (selfLabel(N) is non-hexagonal number))
14 Label = original_function(selfLabel(M))
15 selfLabel(P) = hexagonal_function(Label + selfLabel(N))
16 label(P) = parentLabel(P) | selfLabel(P)
17 End If
18 If ((selfLabel(M) is non-hexagonal number) and (selfLabel(N) is hexagonal number))
19 Label = original_function(selfLabel(N))
Let $M$ and $N$ be two nodes, where node $M$ is labeled as $(m_1, m_2, \ldots, m_s)$ and node $N$ is labeled as $(n_1, n_2, \ldots, n_t)$. Algorithm 2 is used for the insertion of a new node (such as $M$) before the leftmost sibling (such as $N$). This done by reducing the self-label of the leftmost sibling $N$ by 1 as shown in line 02, and then by applying the following Equation (2) to create the new label of a new node $N$. If the self-label of $M$ is the same as the result of the equation, then it is a hexadecimal number, so reduce the self-label of $N$ by the value of 1 again. The parent label of $M$ is linked with the new self-label of $M$ to generate a new label. For instance, the leftmost sibling node is labeled $m_1, m_2, \ldots, m_s$, and the generated new label is $m_1, m_2, \ldots, (m_s-1)$. However, if $(m_s-1)$ is a hexadecimal number, then the generated new label is $m_1, m_2, \ldots, (m_s-2)$.

$$O(n) = \frac{1 + \sqrt{8n + 1}}{4} \quad (2)$$

Algorithm 3 is used for the insertion of a new node (such as $M$) after the rightmost sibling (such as $N$). This done by incrementing the self-label of the rightmost sibling $N$ by 1 as shown in line 02, and then applying Equation (2) above to create the new label of a new node $M$. If the self-label of $M$ is the same as the result of the equation, then it is a hexadecimal number, so increment the self-label of $N$ by the value of 1 again. The parent label of $M$ is linked with the new self-label of $M$ to generate a new label. For instance, the rightmost sibling node is labeled $m_1, m_2, \ldots, m_s$, and the generated new label is $m_1, m_2, \ldots, (m_s+1)$. However, if $(m_s+1)$ is a hexadecimal number, then the generated new label is $m_1, m_2, \ldots, (m_s+2)$.

Algorithm 4 is used to insert a child into a leaf node, such as $M$, where the new label is produced by linking the parent label of $M$ with the number “2”. For instance, the generated child node $M$ is labeled as $m_1, m_2, \ldots, m_s, 2$.

Algorithm 5 is used to handle four sub-cases of inserting node $P$ between two siblings $M$ and $N$. Firstly, if the self-label of a node $M$ and the self-label of a node $N$ are non-hexagonal numbers, then the self-label of a node $P$ is created by adding the self-label of $M$ to the self-label of $N$ and then applying a hexadecimal conversion function. For instance, suppose we have two siblings, the left sibling is labeled as $M: m_1, m_2, \ldots, m_s$ and the right sibling is labeled as $N: n_1, n_2, \ldots, n_t$, where $M = N$ and $m_s$, $n_t$ are non-hexagonal numbers; Algorithm 6 is applied by using a hexadecimal conversion function for $(m_s + n_t)$ in order to obtain the new self-label of $P$ and generates the new label as the parent label and concatenated with the new self-label.

Secondly, if the self-label of a node $M$ and the self-label of a node $N$ are hexagonal numbers, then the original conversion function is applied by using Algorithm 7 and Equation (2) to obtain the self-labels of nodes $M$ and $N$, and these two self-labels are added together to obtain a new value. This value (i.e., the summation of the self-labels of a node $M$ and a node $N$) is converted by using the hexadecimal conversion function in Algorithm 6 and Equation (1) to obtain the new self-label of a node $P$. The new label is produced by linking the parent label of $P$ with the new self-label of a node $P$. For instance, suppose we have two siblings, the left sibling is labeled as $M: m_1, m_2, \ldots, m_s$ and the right sibling is labeled as $N: n_1, n_2, \ldots, n_t$, where $M = N$, and $m_s$, $n_t$ are hexagonal numbers; Algorithm 7 is applied by using an original conversion function for $m_s$ and $n_t$, and then applying the hexadecimal conversion function for the original values $m_s^2 + n_t$ in order to get the new self-label. Then, the concatenation between the prefix label for the parent and the new self-label is assigned to the new label.

Thirdly, if the self-label of a node $M$ is a hexagonal number and the self-label of a node $N$ is a non-hexagonal number, the original conversion function for the self-label of $M$ is applied by using Algorithm 7 and Equation (2) and the two values are added together. Then, in order to produce a new self-label, the hexadecimal conversion function is used. The parent label of $P$ is linked with the new self-label of $P$ to generate a new label. For instance, suppose we have two siblings, the left sibling is labeled as $M: m_1, m_2, \ldots, m_s$ and the right sibling is labeled as $N: n_1, n_2, \ldots, n_t$, where $M = N$, $m_s$ is a hexagonal number, and $n_t$ is a non-hexagonal number; the algorithm 7 is applied by using the original conversion function for $m_s$ and then the hexadecimal conversion function is also applied for the original values $m_s^2 + n_t$ in order to obtain the new self-label. Then, the concatenation between the parent's prefix label and the new self-label is assigned to the new label.

Fourthly, if the self-label of a node $M$ is a non-hexagonal number and the self-label of a node $N$ is a hexagonal number, the original conversion function for the self-label...
of $N$ is applied by using Algorithm 7 and Equation (2) and the two values are added together. Then, in order to produce a new self-label, the hexagonal conversion function is used. The new label is produced by linking the parent label of $P$ with the new self-label of $P$. For example, given two siblings, the left sibling is labeled as $M = m_1, m_2, \ldots, m_v$ and the right sibling is labeled as $N = n_1, n_2, \ldots, n_y$, where $M = N$, $m_v$ is a non-hexagonal number, and $n_y$ is a hexagonal number; the algorithm 7 is applied by using the original conversion function for $n_y$ and then the hexagonal conversion function is also applied for the original values $m_v + n_y'$ in order to get the new self-label. Then, the new label is produced by concatenating the parent's prefix label with the new self-label.

Figure 8 below illustrates all four cases of node insertion in the proposed approach. The new nodes inserted into the XML trees are represented by dashed circles and lines. The alphabetical order of the inserted nodes is represented by the letters inside the dashed circles. Node $a$ is inserted before the first leftmost child; the leftmost sibling is labeled as 1.2, and the newly generated label is 1.0. Node $b$ is inserted between two non-hexagonal numbers, label nodes 1.2 and 1.4, and its 1.hexagonal (2 + 4), which is 1.66. Node $c$ is inserted between hexagonal and non-hexagonal numbers, which are 1.66 and 1.4, respectively. So, the original number is returned, which is 6 and its 1.hexagonal (6 + 4) is 1.190. Node $d$ is inserted between two hexagonal numbers, 1.66 and 1.190, and the original numbers, 6 and 10, are returned; its 1.hexagonal (6 + 10) is 1.496. Node $e$ is inserted after the rightmost sibling and labeled as 1.4; the produced new label is 1.5 since 1.(4+1) is not a hexagonal number and so does not need to be increased again by 1. Node $f$ is inserted as a child of label node 1.2, and the produced new label is 1.2.2.

**FIGURE 8. Handling XML updates (insertions) in the proposed scheme.**

V. Experimental design

A. EXPERIMENTAL SETUP

A set of experiments were carried out in order to evaluate the proposed labeling scheme, which involved comparing the scheme with the dynamic Dewey encoding (DDE) scheme, dynamic prefix labeling scheme (DPLS), and the ORDPATH scheme. These conducted experiments were on the initial labeling as well as the handling of XML updates (in the case of insertions). The comparisons were made in terms of labeling time in milliseconds and the label size in Kbytes. Java programming language was used to code the compared schemes and the proposed labeling scheme. The labeling schemes were run by using the same experimental platform: Eclipse IDE 4.18.0 with a Java platform (JDK 8) as the Java interface. All the experiments were run on the same machine with an Intel Core i7 processor, 8 GB of main memory and a Windows 10 64-bit operating system.

B. Experimental DATASETS

The experiments were conducted on different XML datasets [86] in order to test the schemes on a variety of XML trees that vary in terms of number of nodes, file size and depth. A variety of XML datasets that reflect their scalability in evaluation and results is considered important. In this study, real-life XML datasets were employed which are as follows: Digital Bibliography Library Project (DBLP), XMark, TreeBank, NASA, Sigmod, Ebay, and UWM.

The DBLP database is a massive XML file that contains data about computer science journals, conferences, series, and books. Many XML database systems employ the DBLP dataset, which was chosen for this study because it can supply a wide range of sibling nodes with a maximum breadth of 328,858 [87]. The XMark dataset is well-known as the most prominent XML data management benchmark [88, 89]. The XMark is a scalable document database with a deep recursive ancestor structure and a large file size of 111 MB. Moreover, the total number of descendant nodes are 25,500 with depth of 12 and with a varying breadth at each level. The TreeBank dataset was generated by the University of Pennsylvania’s Department of Computer and Information Science, and it has a size of 82 MB and a maximum breadth of 144,493 [86]. The NASA dataset, which was generated by a NASA XML Project from a flat file format, provides reliable astronomical data. NASA’s XML file is 23 MB in size and has a maximum width of 80,396. The other three datasets are the Sigmod dataset, which is commonly used to analyze and examine small XML documents [86]; the EBay dataset, which contains auction data converted to XML from Web sources, and the UWM dataset, which stores data about university courses derived from university websites [86]. These XML datasets are summarized in the following Table II.

| TABLE II SUMMARY OF XML DATASETS |
|----------------------------------|
| Dataset | Size | No. of Elements | No. of Attributes | Max-Depth |
|---------|------|----------------|------------------|-----------|
| DBLP    | 127 MB | 3332130 | 404276 | 6 |
| XMark   | 111 MB | 2048193 | 381878 | 12 |
| TreeBank| 82 MB  | 2437666 | 1 | 36 |
| NASA    | 23 MB  | 476646  | 56317 | 8 |
| UWM     | 2 MB   | 66729   | 6 | 5  |
| Sigmod  | 467 KB | 11526   | 3737  | 6  |
| Ebay    | 34 KB  | 156     | 0     | 5  |
VI. EXPERIMENTAL RESULTS AND DISCUSSION

Five experiments were carried out to evaluate the proposed labeling scheme in order to compare its performance against that of the DDE, DPLS and ORDPATH schemes. The first experiment was carried out to measure the initial labeling process of each scheme in terms of the labeling time and label size. This process was individually executed on the datasets using the various schemes many times because it has been recommended that the number of runs had to total at least 10 for a certain scheme [90, 91]. However, it has also been suggested that there should be at least 30 runs for more accurate results [92]. In these experiments, the first three runs were excluded to make the results more reliable and accurate as well as to clear cache memory.

The first experiment was conducted to assess the initial labeling time and label size. As shown in Figure 9 below, the proposed scheme outperformed the other schemes (i.e., DDE, DPLS, and ORDPATH) in all datasets (see Table II above) because the time taken to label the documents was less and more efficient when compared with the other schemes, particularly when the XML document size became larger and had greater depth. As mentioned above, it is necessary to have an adequate number of runs to gain significant results, therefore all schemes were run 13 times with the first three runs ignored to achieve accuracy in the results for the time needed to generate initial labels in the different datasets.

In terms of the initial labeling size, the results in Figure 10 show that there was no substantial difference between the schemes. This because DPLS and DDE utilize the same scheme for producing the initial labels, despite the proposed scheme avoids the hexagonal labels at this stage. In addition, the expectation of growth in the label size would be high, particularly, when XML document become bigger in size. As a result, the size of loading the initial labels for the proposed scheme, the DPLS, and DDE schemes is nearly the same for the DBLP, XMark, Treebank, NASA, UWM, Sigmod, and Ebay datasets, as shown in Figure 10. In addition, the results in the figure indicate that the initial label size produced by the ORDPATH scheme is the highest as compared to the remaining schemes.

Because the effectiveness of the dynamic labeling schemes arise during consecutive sibling insertions [34, 42], the second experiment was conducted to evaluate the random skewed insertions that handle new nodes between two consecutive siblings' nodes on a frequent basis. This experiment was conducted to compare the proposed scheme with the DDE, DPLS, and ORDPATH schemes in terms of labeling time and label size because all the compared schemes possess a powerful labeling dynamic when applied to XML data.
According to the labeling time measurement results, the proposed scheme achieved the fastest labeling time when the random skewed node insertions are handled as compared with the other schemes, as shown in Figure 1 above. A lot of insertions between two siblings were supported efficiently by the proposed scheme. Therefore, the proposed scheme’s reliability was reasonable. The results show that DPLS had slowest labeling time when the random skewed node insertions are handled because it used a fraction fragment (i.e., a floating-point number) to represent the last component of the label for each insertion. As regards the label size, the proposed scheme also outperformed the other schemes, especially when a large number of skewed nodes has been updated randomly, as shown in Figure 2 above. The results show that the label size of newly inserted nodes in DPLS was highest as compared the remaining schemes.

The remaining three experiments were conducted to evaluate the performance of the proposed labeling scheme compared with other schemes when handling XML updates based on three different cases: the insertion of nodes before the leftmost sibling, the insertion of nodes after the rightmost sibling, and the insertion of leaf nodes. In these experiments, the labeling time and label size were measured when the number of insertions increased. Each experiment was individually executed many times, and the number of runs had to total at least 10 for a certain scheme [90, 91]. In the current study, the first three runs were excluded to ensure the results be more reliable and accurate, as well as to clear cache memory.
In respect of labeling time, the results in Figures 13, 15, and 17 show that the proposed scheme was able to outperform the other schemes (i.e., DDE, DPLS, and ORDPATH) in handling XML updates (i.e., the increase in the number of insertions) for the three cases: the insertion of nodes before the leftmost sibling, the insertion of nodes after the rightmost sibling, and the insertion of leaf nodes, especially when the number of insertions increased in large XML document with high depth. The results also show that the other schemes (i.e., DDE, DPLS, and ORDPATH) had negative performance (i.e., highest labeling time) in handling large XML updates (i.e., the number of insertions increased) for those three cases. This because that they only allowed for a certain number of insertions and did not support dynamic updates efficiently [23, 25, 83].

Regarding to label size, the results in Figures 14 and 16 show that the proposed scheme was able to outperform the other schemes in handling XML updates (i.e., the number of insertions increased) for the two cases: the insertion of nodes before the leftmost sibling and the insertion of nodes after the rightmost sibling, respectively. It generated more compact labels every time as compared to the other schemes, leading to reduced storage costs, especially, when the number of insertions increased in large XML document with high depth. In addition, the results in the figure indicate that the label size produced by the DDE and DPLS schemes was the same because they used the same scheme in updating. Moreover, the results also show that the label size produced by ORDPATH was the highest as compared to the remaining schemes because it used even and negative integer labels for the inserted nodes, which led high label size when the number of insertions increased in large XML document with high depth. However, the results in Figure 18 shows that there was a negligible difference in how the labeling schemes handled XML updates (i.e., the increase in the number of insertions) in case of the insertion of leaf nodes. This because that they had the same scheme when labelling new leaf nodes.

VII. CONCLUSION

In this study, an efficient prefix-based labeling scheme that uses a hexagonal pattern to support dynamic updates in XML documents was proposed. The proposed labeling scheme is based on the prefix-based labeling scheme. According to the experiment results, the scheme is considered to improve on the performance of the DDE scheme and the DPLS. The proposed scheme prevents node relabeling when the XML documents are updated at random locations, avoids duplicated labels by creating new labels for every inserted node, and reduces the size and time costs for the updated labels. In addition, the proposed labeling scheme was evaluated against other prefix-based labeling schemes in terms of the label size produced and time taken for the labeling process and in terms of its ability to handle different types of updates. Five experiments were conducted to compare the proposed scheme’s effectiveness and scalability to handle initial labeling and XML updates (insertions) with other schemes, namely, the DDE scheme, DPLS, and ORDPATH scheme.

The first experiment was carried out to assess labeling time and label size for different types and sizes of XML dataset. The experiment indicated that the proposed scheme outperformed other schemes in most of the XML datasets because it took less time to label the documents and was more efficient as compared with the other schemes, particularly when the XML document size became larger and had greater depth, as in the DBLP, XMark and Treebank datasets. The experiment results also indicated that the size of the initial labels produced by the proposed scheme and by the DDE scheme and DPLS were almost the same. This because the DDE scheme and DPLS use the same procedure to produce the initial labels, and the proposed scheme avoids using hexagonal numbers during the initial labeling stage.

The second experiment was carried out to measure the random skewed insertions between two consecutive siblings’ nodes. The results of this experiment indicated that the proposed scheme outperformed the other schemes in terms of labeling time and label size. The proposed scheme produced labels with lesser size every time as compared to the other schemes, leading to reduced storage costs. It also outperformed when a large number of randomly skewed nodes was updated.

The remaining three experiments were carried out to evaluate the performance of the proposed labeling scheme as compared to that of the other schemes when handling XML updates based on three different cases: the insertion of nodes before the leftmost sibling, the insertion of nodes after the rightmost sibling, and the insertion of leaf nodes. In respect of labeling time, the experiments indicated that the proposed scheme outperformed the other schemes in handling XML updates (i.e., the increase in the number of insertions) for the three cases. As regards the label size, the experiments showed that the proposed scheme was able to outperform the other schemes in handling XML updates (i.e., the increase in the number of insertions) for the two cases: the insertion of nodes before the leftmost sibling and the insertion of nodes after the rightmost sibling, respectively. It generated more compact labels every time as compared to the other schemes, leading to reduced storage costs, especially, when the XML document size became larger and had greater depth. However, the results also show that there was a negligible difference in how the labeling schemes handled XML updates (i.e., the increase in the number of insertions) in case of the insertion of leaf nodes.

The contributions of the research presented in this paper can be summarized as follows:
1) The existing and different types of XML labeling schemes were reviewed and discussed such as Interval-based labeling schemes, Prefix-based labeling scheme, and other labeling schemes in handling XML updates.

2) The limitations of the existing XML labeling schemes were presented and summarized in terms of labeling time and size in handling XML updates.

3) An efficient prefix-based labeling scheme that uses a hexagonal pattern to support dynamic updates in XML documents was proposed. The advantages of the proposed labeling scheme are: (i) it avoids node relabeling when the XML documents are updated at random locations, (ii) it avoids duplicated labels by creating a new label for every inserted node, (iii) reduces the size and time costs of the updated labels, and (iv) it supports dynamic updates.

4) The proposed labeling scheme was implemented to maintain the structural information among the nodes based on four different insertions cases, namely, insertion before the leftmost sibling, insertion after the rightmost sibling, insertion between two siblings, and insertion of a child into a leaf node.

5) The proposed labeling scheme was evaluated comparing with the existing labeling schemes in terms of label size, labeling time, and its ability to handle several insertions in XML documents based on four different insertions cases.

Based on the findings of this study, it is hoped that the labeling scheme proposed in this paper will contribute to the ability of commercial organizations to exchange and manage their data efficiently on the Web in terms of both label size and labeling time. The scheme also serves as an introductory basis upon which researchers can build to further enhance the efficiency of labeling schemes to support dynamic XML updates and extensions into new research areas. In our future work, we aim to research the issue of reusing deleted labels. Another possible dimension of future work is to consider the syntactic and semantical relatedness of node labels in designing new labeling scheme.

References

[1] J. Liu and X. Zhang, "Dynamic labeling scheme for XML updates," Knowledge-Based Systems, vol. 106, pp. 135-149, 2016.

[2] A. Qtaish and K. Ahmad, "Ancestor: An efficient mapping approach for storing and querying XML documents in relational database using path-based technique," Journal of Knowledge-Based Systems, vol. 114, no. 2016, pp. 167-192, 2016.

[3] E. Khanjari and L. Gaeini, "A new effective method for labeling dynamic XML data," Journal of Big Data, 2018.

[4] M. Adam, I. Fahkharaldien, E. Khalid, M. Jasni, and S. Norrozila, "Efficient algorithms to index, store and query XML documents," in International Conference on Computational Science and Information Management (ICoSIM), 2012, vol. 1, pp. 281-286.

[5] E. Song and S.-C. Haw, "XML-REG: Transforming XML Into Relational Using Hybrid-Based Mapping Approach," IEEE Access, vol. 8, pp. 177623-177639, 2020.

[6] Z. Boualem and I. Cherti, "XMap: A Novel Approach to Store and Retrieve XML Document in Relational Databases," Journal of Software, vol. 10, no. 12, 2015.

[7] A. Qtaish and M. T. Alshammari, "A Narrative Review of Storing and Querying XML Documents Using Relational Database," Journal of Information & Knowledge Management, vol. 18, no. 04, p. 1950048, 2019.

[8] A. Qtaish and K. Ahmad, "Model-Mapping Approaches for Storing and Querying XML Documents in Relational Database: A Survey," Journal of Convergence Information Technology, vol. 9, no. 2, 2014.

[9] A. Qtaish and K. Ahmad, "Model mapping approaches for XML documents: A review," Journal of Information Science, vol. 41, no. 4, pp. 444-466, April 14, 2015 2015, doi: 10.1177/0165551515579995.

[10] A. Qtaish and K. Ahmad, "Query mapping techniques for XML documents: A comparative study," in Electrical Engineering and Informatics (ICEEI), 2015 International Conference on, 2015: IEEE, pp. 529-534.

[11] M. F. O'Connor and M. Roantree, "Desirable properties for XML update mechanisms," in Proceedings of the 2010 EDBT/ICDT Workshops, 2010, pp. 1-9.

[12] C. Li, T. W. Ling, and M. Hu, "Efficient updates in dynamic XML data: from binary string to quaternary string," in VLDB Journal, vol. 17, no. 3, pp. 573-601, 2008.

[13] T. A. Ghaele and S. Mohammed, "A dynamic labeling scheme based on logical operators: a support for order-sensitive XML updates," Procedia Computer Science, vol. 57, pp. 1211-1218, 2015.

[14] Y. He, "A Novel Encoding Scheme for XML Document Update-supporting," in International Conference on Advances in Mechanical Engineering and Industrial Informatics (AMEII), Zhengzhou, Atlantis Press, 2015.

[15] H. Wang, S. Park, W. Fan, and P. S. Yu, "ViST: A dynamic index method for querying XML data by tree structures," in Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 2003, pp. 110-121.

[16] S. Alsubai and S. North, "A prime number approach to matching an XML twig pattern including parent-child edges," in Proceedings of the 13th International Conference on Web Information Systems and Technologies, 2017: SCITEPRESS, pp. 204-211.

[17] T. Bray, J. Paoli, C. M. Sperber-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (XML)," https://www.w3.org/TR/1998/REC-xml-19980210 (accessed 10-07-2014).

[18] S.-C. Haw, "XMapDB-Sim: Performance Evaluation on Model-Based XML to Relational Database Mapping Choices," Indonesian Journal of Electrical Engineering and Computer Science, vol. 7, no. 2, pp. 551-566, 2017.

[19] H. Kaplan, T. Milo, and R. Shabo, "A comparison of labeling schemes for ancestor queries," in SODA, 2002, vol. 2: Citeseer, pp. 954-963.

[20] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekitka, and C. Zhang, "Storing and querying ordered XML using a relational database system," in Proceedings of the 2002 ACM SIGMOD international conference on Management of data, 2002: ACM, pp. 204-215.

[21] I. Mlynková, "An analysis of approaches to xxml schema inference," in 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems, 2008: IEEE, pp. 16-23.

[22] E. Cohen, H. Kaplan, and T. Milo, "Labeling dynamic XML trees," in Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2002, pp. 271-281.
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3178438, IEEE Access

Otaish & Alshudukhi: Preparation of Papers for IEEE Access (February 2017)

ACM international conference on Digital libraries, 1996, pp. 91-99.

D. D. Kha, M. Yoshikawa, and S. Uemura, "A structural numbering scheme for XML data," in International Conference on Extending Database Technology, 2002: Springer, pp. 91-108.

R. Al-Shaikh, G. Hashim, A. BinHuraih, and S. Mohammed, "A modulo-based labeling scheme for dynamically ordered XML trees," in 2010 Fifth International Conference on Digital Information Management (ICDIM), 2010: IEEE, pp. 213-221.

S.-C. Haw and A. Amin, "Node indexing in XML query optimization: A review," Indian Journal of Science and Technology, vol. 8, no. 32, pp. 1-9, 2015.

H. Su-Cheng and L. Chien-Sing, "Node labeling schemes in XML query optimization: a survey and trends," IETE Technical Review, vol. 26, no. 2, pp. 88-100, 2009.

R. K. Wong, F. Lam, and W. M. Shui, "Querying and maintaining a compact XML storage," in Proceedings of the 16th international conference on World Wide Web, 2007, pp. 1073-1082.

Q. Zou, S. Liu, and W. W. Chu, "Ctree: a compact tree for indexing XML data," in Proceedings of the 6th annual ACM international workshop on Web information and data management, 2004, pp. 39-46.

U. Nambiar, Z. Lacroix, S. Bressan, M. L. Lee, and Y. G. Li, "Efficient XML data management: an analysis," in International Conference on Electronic Commerce and Web Technologies, 2002: Springer, pp. 87-98.

W. Meier, "eXist: An open source native XML database," in Web, Web-Services, and Database Systems: Springer, 2003, pp. 169-183.

T. Amagasa, M. Yoshikawa, and S. Uemura, "QRS: A robust numbering scheme for XML documents," in Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37403), 2003: IEEE, pp. 705-707.

C. Zhang, J. Naughton, D. Devitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," in ACM SIGMOD Record, 2001, vol. 30, no. 2: ACM, pp. 425-436.

B. G. Assefa and B. Ergenc, "OrderBased labeling scheme for dynamic XML query processing," in International Conference on Availability, Reliability, and Security, 2012: Springer, pp. 287-301.

I. Dweib, A. Awadi, S. E. F. Elrhman, and J. Lu, "Schemaless approach of mapping XML document into Relational Database," in Computer and Information Technology, 2008: CIT 2008. 8th IEEE International Conference on, 2008: IEEE, pp. 167-172.

J. Zhou et al., "Top-down XML keyword query processing," IEEE Transactions on Knowledge and Data Engineering, vol. 28, no. 5, pp. 1340-1353, 2016.

J. Li, C. Liu, R. Zhou, and W. Wang, "XML keyword search with promising result type recommendations," World wide web, vol. 17, no. 1, pp. 127-159, 2014.

Y. Zeng, Z. Bao, and T. W. Ling, "Supporting range queries in XML keyword search," in Proceedings of the Joint EDBT/ICDT 2013 Workshops, 2013, pp. 97-104.

E. Taktek and D. Thakker, "Pentagonal scheme for dynamic XML prefix labeling," Knowledge-Based Systems, vol. 209, p. 106646, 2020.

H.-K. Ko and S. Lee, "A binary string approach for updates in dynamic ordered XML data," IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 4, pp. 602-607, 2009.

L. Bao, J. Yang, C. Q. Wu, H. Qi, X. Zhang, and S. Cai, "XML2HBase: Storing and querying large collections of XML documents using a NoSQL database system," Journal of Parallel and Distributed Computing, vol. 161, pp. 83-99, 2022.

T. A. Ghaleb and S. Mohammed, "Novel scheme for labeling XML trees based on bits-masking and logical matching," in 2013 World Congress on Computer and Information Technology (WCCIT), 2013: IEEE, pp. 1-5.

G. Dhanalekshmi and A. Krishna, "LPLX-lexigraphic-based persistent labelling scheme of XML documents for dynamic update," International Journal of Web Science, vol. 2, no. 4, pp. 237-257, 2014.

G. Dhanalekshmi and K. Asawa, "An efficient prefix based labelling scheme for dynamic update of XML documents," International Journal of Advanced Intelligence Paradigms, vol. 18, no. 4, pp. 464-480, 2021.

M. Clements, "Hexagonal numbers," Mathematics in School, vol. 3, no. 3, pp. 32-32, 1974.

"Hexagonal number," https://en.wikipedia.org/wiki/Hexagonal_number (accessed 8/10/2021).

University of Washington's Website. "UW XML Data Repository." http://www.cs.washington.edu/research/xmldatasets/wwwrepository.html (accessed 10-11-2021).

M. Al-Badawi, "A Performance Evaluation of a New Bitmap-based XML Processing Approach," The University of Sheffield, 2010.

A. Schmidt, S. Waas, M. Kersten, M. Carey, J. Manolescu, and R. Busse, "XMark: A benchmark for XML data management," in Proceedings of the 28th international conference on Very Large Data Bases, 2002: VLDB Endowment, pp. 974-985.

M. Franceschet, "XPathMark: An XPath benchmark for the XMark generated data," in International XML Database Symposium, 2005: Springer, pp. 129-143.

S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar, "Walavege, A systematic review of the application, and empirical investigation of search-based test case generation," IEEE Transactions on Software Engineering, vol. 36, no. 6, pp. 742-767, 2009.

J. Wegener, A. Barasel, and H. Shamer, "Evolutionary test environment for automatic structural testing," Information and software technology, vol. 43, no. 14, pp. 841-854, 2001.

A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," Software Testing, Verification and Reliability, vol. 24, no. 3, pp. 219-250, 2014.

Amjad Qtaish is an assistant professor in the Computer Science and Software Engineering Department, College of Computer Science and Engineering, University of Hail, Saudi Arabia. He received his B.Sc. degree in computer science from Zarqa University, Jordan, in 2002. M.Sc. degree in distributed system from University Putra Malaysia, Malaysia, in 2003 and his PhD degree in computer science (XML mapping using relational databases) from National University of Malaysia, Malaysia, in 2016. His research interests include XML mapping, data science, big data, relational databases, data management, and information retrieval.

Jalawi Alshudukhi is an associate professor in the fields of Wireless Sensor Networks WSN, Wireless Communication and Internet of Things (IoT). He received the BSc degree in Computer Science from University of Hail, KSA in 2006, then he earned his M.Sc. degree in Computer Networks from La Trobe University, Australia, in 2010. He received his Ph.D Computer Science from Oxford Brookes University, United Kingdom in 2016. He has occupied several executive management positions as chairman of Information and Computer Science department at University of Hail. Currently, he has been appointed as Dean of College of Computer Science and Engineering at University of Hail. He also served as a reviewer, association editor, or TPC for several conferences such as WCI-2015 (reviewer), ATDA18 (TPC member), IURA-2020 (reviewer) and ICOMITEE-2019 (reviewer). He is an active member of different societies and councils. He is a member of the IEEE Communications Society and member of IEEE Sensors Council. And other local organizations.