Labeling Dynamic XML Documents: An Order-Centric Approach

Liang Xu, Tok Wang Ling, and Huayu Wu
School of Computing
National University of Singapore

Abstract—Dynamic XML labeling schemes have important applications in XML Database Management Systems. In this paper, we explore dynamic XML labeling schemes from a novel *order-centric* perspective. We compare the various labeling schemes proposed in the literature with a special focus on their orders of labels. We show that the order of labels fundamentally impacts the update performance of a labeling scheme and develop an order-based framework to classify and characterize XML labeling schemes. Although there are dynamic XML labeling schemes that can completely avoid re-labeling, the gain in update performance all come with considerable costs such as larger label size and lower query performance, even if the XML documents are hardly updated. We introduce vector order which is the foundation of the dynamic labeling schemes we propose. Compared with previous solutions that are based on natural order or lexicographical order, vector order is a simple, yet most effective solution to process updates in XML DBMS. We show that vector order can be gracefully applied to both range-based and prefix-based labeling schemes with little overhead introduced. Moreover, vector order-based labeling schemes are not only efficient to process, but also resilient to skewed insertions. Qualitative and experimental evaluations confirm the benefits of our approach compared to previous solutions.

Index Terms—Tree node labeling, XML data management, Update processing, Query processing.

1 Introduction

The increasing importance of XML data management has led to extensive research on native XML storage and query support. The challenge of managing XML data comes from its ordered tree-structured model which provides rich semantic content and is essential for querying XML data at a fine level of granularity. Labeling schemes encode both order and structural information (E.g. Parent/Child, Ancestor/Descendant) of the XML tree into extremely compact labels, which is a widely adopted approach for XML query processing and has received a lot of research attention.

Existing XML labeling schemes are usually classified into two categories: range-based[13], [18], [14], [1] and prefix-based[7], [2], [15], [12], [10]. Both range-based and prefix-based labeling schemes provide fine-grained labeling of XML data and are adopted by an array of applications. We consider the following criteria important for the comparison of labeling schemes:

- Order and structural information. Documents obeying XML standard are intrinsically ordered and typically modeled as a tree. Labeling schemes encode both document order and structural information so that queries can exploit them. While document order is essential to be encoded, the amount of structural information contained in the labels may vary. For example, sibling relationship can be derived from prefix-based labeling schemes, but in general not from range-based labeling schemes.
- Query efficiency. Deriving structural information from labels should be as efficient as possible.
- Update efficiency. It is desirable to have a persistent labeling scheme, i.e. updating XML documents should not require existing labels to be re-labeled. This is crucial for low update costs and for the users to be able to query the changes of the XML data over time[2].
- Size. Size is an important factor that contributes to query and update performance.

However, designing labeling schemes that fulfill all these criteria turns out to be a challenging problem. Most early works[13], [18], [14], [1], [7], [2], [15] on labeling schemes cannot satisfy the third criteria and requires re-labeling when updating the XML documents. More dynamic solutions[5], [4], [6], [12] have been proposed, however at the cost of lower query performance and less compact size even for XML documents that are seldom updated. In this paper, we tackle this problem from a novel *order-centric* perspective. We argue that the order of labels is the key to process updates in dynamic XML documents. We establish a order-based framework to categorize existing labeling schemes, which provides insight into their update processing. Based on the framework, we introduce a novel order concept, vector order, with its application to both range-based and prefix-based labeling schemes. The resulting labeling schemes including V-Containment, V-Pre/post and Dynamic DEwey (DDE) are not only tailored for static documents, but also can completely avoid re-labeling for updates. Both qualitative and experimental comparisons demonstrate the advantages of our labeling schemes over the previous approaches.

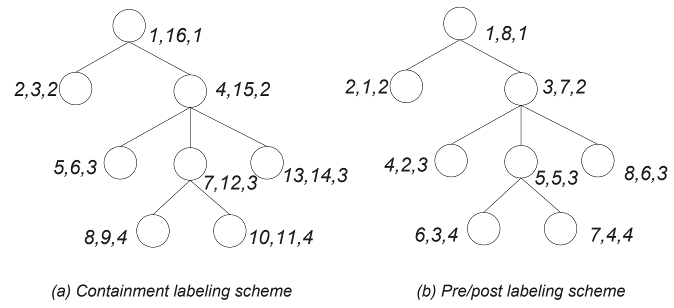


Fig. 1. Range-based labeling schemes

1.1 Road map

In Section 2, we describe the various labeling schemes that have been proposed in the literature. We compare and contrast the orders adopted by these labeling schemes which play an important part in their behaviors for both update and query processing. We show the limitations of these labeling schemes which motivate the introduction of our vector order in Section 3. We present range-based and prefix-based labeling schemes based on vector order in Section 4 and Section 5. Qualitative comparisons are presented in Section 6 which characterize our labeling schemes and existing ones under a unified framework. Experimental results in Section 7 demonstrate the benefits of our labeling scheme and therefore justify the use of vector order. We conclude the paper in Section 8.

2 Preliminary

2.1 Labeling tree-structured data

2.1.1 Range-based labeling schemes

In Figure 1, we present examples of containment[18] and pre/post[13] labeling schemes which both belong to range-based labeling schemes.

In containment labeling scheme, each element node is assigned a label of the form $start$, end, $level$ where $start$ and end define a range that contains all its descendant's ranges. Each label in pre/post labeling scheme is of the form pre, $post$, $level$ where pre and $post$ are the ordinal numbers of the element node in preorder and postorder traversal sequences respectively. For both labeling schemes, $level$ represents the level of the element node in the XML tree.

Given two containment labels $A(s_1, e_1, l_1)$ and $B(s_2, e_2, l_2)$, the following structural information can be derived:

P1 Ancestor/Descendant(AD). A is an ancestor of B if and only if $s_1 < s_2 < e_2 < e_1$, which can be simplified as $s_1 < s_2 < e_1$. The simplification is based on the observation that it is impossible to have $s_1 < s_2 < e_1 < e_2$ which implies the elements are not properly nested.

P2 Parent/Child(PC). A is the parent of B if and only if A is an ancestor of B and $l_1 = l_2 - 1$.

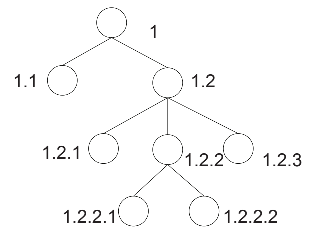


Fig. 2. Dewey labeling scheme

Both AD and PC relationships can be derived from pre/post labels as well. Here we highlight the following difference:

- Given two pre/post labels $A(pre_1,\ post_1,\ l_1)$ and $B(pre_2,\ post_2,\ l_2)$, A is an ancestor of B if and only if $pre_1 < pre_2$ and $post_2 < post_1$. This condition is different from that of containment labeling scheme and cannot be similarly simplified.

Example 2.1: In Figure 1 (a), (4,15,2) is an ancestor of (8,9,4) because $4 < 8 < 15$. (7,12,3) is the parent of (8,9,4) because $7 < 8 < 12$ and 3=4-1. In Figure 1 (b), $3, 7, 2$ is an ancestor of $6, 3, 4$ because $3 < 6$ and $3 < 7$.

In order/size labeling scheme[14], each label consists of a triplet $order$, $size$, $level$. order/size labeling scheme can be seen as a variation of containment labeling scheme where a range is defined by $order$ and $(order + size)$.

2.1.2 Prefix-based labeling schemes

An example of Dewey labeling scheme[7], which is the representative of prefix-based labeling schemes, is shown in Figure 2. The order that Dewey labeling scheme makes heavy use of is the order among siblings, which we refer to as *local order*. By concatenating the label of its parent ($parent\_label$) with its own local order, a Dewey label uniquely identifies a path from the root to an element.

Given two Dewey labels $A : a_1.a_2 \ldots a_m$ and $B : b_1.b_2 \ldots b_n$, the following rules can be used to derive structural information from them:

P1 Ancestor/Descendant(AD). A is an ancestor of B if and only if $m < n$ and $a_1 = b_1, a_2 = b_2, \ldots, a_m = b_m$.

P2 Parent/Child(PC). A is the parent of B if and only if and only if A is an ancestor of B and $m = n - 1$

P3 Sibling. A is the sibling of B if and only if $m = n$ and $a_1 = b_1, a_2 = b_2, \ldots, a_{m-1} = b_{m-1}$, i.e. A's $parent\_label$ matches B's $parent\_label$.

Example 2.2: In Figure 2, 1.2 is an ancestor of 1.2.2.1 because 1.2 is a prefix of 1.2.2.1. 1.2.2 is the parent of 1.2.2.1 because 1.2.2 matches the $parent\_label$ of 1.2.2.1.

2.1.3 Prime labeling scheme

Prime labeling scheme[8] represents a unique approach to encode the tree structure of XML data. In prime labeling scheme, each node is associated with a unique prime number ($self\_label$). The label of a node is a number which is the product of its $self\_label$ and the label of its parent node ($parent\_label$). Since all $self_label$s are distinct prime numbers, the factorization of a label can be used to identify a unique path in an XML tree. Given two nodes n and m, n is an ancestor of m if and only if $label(m)\ mod\ label(n) = 0$. n is the parent of m if and only if $label(n) = label(m)/self\_label(m)$. Although both AD and PC relationships can be encoded elegantly in this way, deciding document order remains a hard problem. We describe how Prime labeling scheme encodes document order in Section 2.2.1.

2.2 Order encoding and update processing

Compared to *unordered* relational data, a key difference we face when processing *ordered* XML data is how to encode the order information[7]. Important order information defined in XML documents include *document order* and *local order*.

Definition 2.1 (Document order): Document order is the order in which the start tags of the element nodes are encountered when the document that contains them is parsed. Note that document order is equivalent to preorder defined on the element nodes if we think of XML documents as linearizations of tree structure.
Local order is the document order among siblings which is trivially consistent with document order.

Given the one-to-one correspondence between labels and element nodes, we can derive document order from a set of labels if they and their associated element nodes have the same ordering. When XML documents are subject to updates, i.e. element nodes are be inserted or deleted at arbitrary positions in the documents, labels have to be inserted or deleted accordingly while preserving the correct order information. This turns out to be a challenging problem especially if no existing labels should be modified. We further elaborate the problem by summarizing the orders used by different labeling schemes.

2.2.1 Range-based labeling schemes and natural order

Since document order is equivalent to preorder on the element nodes, pre/post labeling scheme naturally encodes document order by incorporating the preorder traversal ordinal numbers into the labels. Given two pre/post labels $A(pre_1,\ post_1,\ l_1)$ and $B(pre_2,\ post_2,\ l_2)$, A precedes B in document order if and only if $pre_1 < pre_2$. Similarly, the $start$ values in containment labels are strictly increasing if they are ordered according to document order. Thus, document order can be derived from containment labels from their $start$ values.

The ordering of pre/post and containment labels follows from the *natural order* ($<$) on integers, i.e. pre or $start$. As we know, insertion between two integers requires the use of some new integers which falls between them in natural order. This is not possible if the existing two integers are consecutive, in which case re-labeling is necessary. The re-labeling may have global effect, that is, the whole document has to be re-labeled in the worst case. Leaving gaps[14] in labels only delays re-labeling until some gap is filled. Using floating point number[1] instead of integer does not solve the problem because (a)In standard floating point format, the mantissa is represented by a fixed number of bits, implying that floating point numbers are of limited accuracy; (b)The mantissa can be consumed by as many as 2 bits per insertion, which can lead to overflow quickly and (c) Floating point numbers are inherently less efficient to process than integers.

Prime labeling scheme uses a list of SC(Simultaneous Congruence) values to derive the mapping from $self_label$s to global orders, which are basically ordered by natural order. Whenever a node is inserted or deleted, the global orders are re-ordered. As a result, on average half of the SC values have to be re-calculated based on Euler's quotient function, which has been shown to be very time consuming[6].

2.2.2 Prefix-based labeling schemes and lexicographical order

Document order can be derived from Dewey labels based on lexicographical order (denoted as $\prec_l$) which is defined as follows:

Definition 2.2 (Lexicographical order): Given two Dewey labels $A : a_1.a_2 \dots a_m$ and $B : b_1.b_2 \dots b_n$, $A \prec_l B$ if and only if one of the following two conditions holds:

C1. $m < n$ and $a_1 = b_1, a_2 = b_2, \dots, a_m = b_m$.
C2. $\exists k \in [0, min(m, n)]$, such that $a_1 = b_1, a_2 = b_2, \dots, a_{k-1} = b_{k-1}$ and $a_k < b_k$.

Consider the Dewey labels of two consecutive sibling element nodes, they have the $parent\_label$ and consecutive local orders. From C2 in lexicographical order, the comparison of two labels eventually lead to comparison of local orders in natural order. As a result, re-labeling is unavoidable for insertion between two consecutive siblings, regardless of whether integer or floating point number is used. However, the scope of re-labeling for Dewey labeling scheme is restricted to the subtree in which the new element node is inserted. In this sense, lexicographical order already appears to be more robust than natural order against updates.

2.2.3 Transforming natural order to lexicographical order

After showing that natural order is rigid and inevitably leads to re-labeling, it becomes clear that a different order is necessary to solve the problem of updates. QED encoding scheme has been proposed to transform integers to quaternary strings, which, if we see from the order perspective, is from natural order to lexicographical order.

Definition 2.3 (Quaternary string): Given a set of numbers $A = \{1, 2, 3\}$ where each number is stored with 2 bits, a quaternary string is a sequence of elements in A. Note that number 0 does not appear in quaternary string because it is used as the separator of the quaternary strings for physical encoding. A *QED code* is a quaternary string that ends with 2 or 3. As the following example illustrates, QED codes are robust enough to allow insertions without re-labeling.

Example 2.3: Let 22, 23 be two QED codes satisfying $22 \prec_l 23$, we can insert 222 which is another QED code between them and we have $22 \prec_l 222 \prec_l 23$. To continue to insert between 22 and 222, for example, we can use 2212, satisfying $22 \prec_l 2212 \prec_l 222$.

QED encoding scheme can be applied to both range-based and prefix-based labeling scheme. Application of QED encoding scheme to containment labeling scheme transforms $start$ and end into QED codes and we call the resulting labeling scheme QED-Containment labeling scheme. QED-Dewey labeling scheme is similarly derived by applying QED encoding scheme to Dewey labeling scheme. However, the lengths can QED codes can increase quite fast for ordered insertions (2 bits per insertion as in Example 2.3).

Variable Length Endless Insertable code(VLEI)[3] has been proposed to reduce the cost of insertions. An VLEI code is a bit sequence $v = \{0|1\}*$ which are ordered by lexicographical order. An insertion algorithm can be applied to two VLEI codes to produce a new VLEI code between them. VLEI-Dewey labeling scheme is the result of applying VLEI codes to Dewey labeling scheme, where "9" is used as delimiter between components. The application of VLEI codes has achieved reduction in update time with respect to the use of floating point numbers[1].

ORDPATH [12] is based on dewey and uses only odd numbers at the initial labeling. Even numbers are reserved for insertions and only used as 'caret's. To insert between two ORDPATH labels whose last components are consecutive odd numbers, the new label is generated using an additional even number which falls between the two odd numbers. We refer to this as the 'careting in' technique. For example, to insert between two ORDPATH labels 1.3 and 1.5, we use 4 which is the even number between 3 and 5 as the 'caret'. The new label is 1.4.1 where 4, the caret, is not counted as a component that increases the level of a node.

Based on the 'careting in' technique, each level in an ORDPATH label is possibly represented by a variable number of even numbers followed by an odd number. This property complicates the processing of ORDPATH labels and therefore negatively affects the query performance. For example, computing the LCA of dewey labels is equivalent to finding the longest common prefix of them. For ORDPATH labels, however, extra care has be to taken to make sure the LCA is a valid ORDPATH label. As an example, the longest common prefix of two ORDPATH labels 1.6.2.1 and 1.6.2.3.5 is 1.6.2 whereas their LCA should be 1. The complexity introduced by the 'careting in' technique *fundamentally* affects the query processing with ORDPATH labels even if no update actually takes place.

QED-Dewey, VLEI-Dewey and ORDPATH labeling schemes are similarly ordered, which can be captured by the generalized lexicographical order we introduce.

2.2.4 Generalized lexicographical order

We begin by generalizing the notion of Dewey label.

Definition 2.4 (Generalized Dewey label): A generalized Dewey label is a sequence of components separated by dots, which we denote as $[a_1].[a_2]\ldots[a_m]$. There is no restriction on the content of each component, which can be an integer, a string, a sequence of integers, etc. Nevertheless, the components should be encoded in such a way that allows them to be separable from each other.

For example, QED-Dewey labels fit into the definition of generalized Dewey label because it is a sequence of QED codes separated by delimiter 0. VLEI-Dewey labels are sequences of VLEI codes with "9" as delimiters. In ORDPATH labeling scheme, a label can be thought of as a generalized Dewey label where each component is a variable of even numbers followed by an odd number.

Generalized Dewey labels are compared based on generalized lexicographical order.

Definition 2.5 (Generalized lexicographical order): Given two generalized Dewey labels $A : [a_1].[a_2]\ldots[a_m]$ and $B : [b_1].[b_2]\ldots[b_n]$, A precedes B in generalized lexicographical order if and only if one of the two conditions holds:

C1. $m < n$ and $a_1 \equiv b_1, a_2 \equiv b_2, \ldots, a_m \equiv b_m$.

C2. $\exists k \in [0, min(m,n)]$, such that $a_1 \equiv b_1, a_2 \equiv b_2, \ldots, a_{k-1} \equiv b_{k-1}$ and $a_k \prec b_k$.

$\equiv$ and $\prec$ denote generalized equivalence and generalized less than relation respectively. For generalized lexicographical order to correctly reflect document order, it has to be (a) *total* on the set of labels, i.e. any two generalized Dewey labels from the set of labels are comparable with respect to generalized lexicographical order and (b) *transitive* because document order itself is transitive.

3 Vector order

In this section, we introduce vector order which formalizes our idea in [9]. It is also a generalization of our previous work which restricted vector codes to have both positive x and y components.

3.1 Vector code ordering

Definition 3.1 (Vector code): A vector code is a binary tuple of the form (x, y) with $x > 0$.

A vector code (x, y) can be graphically interpreted as an arrow from the origin to the point (x, y) in a two dimensional plane. The arrow only falls into the first or the forth quadrant because we require $x > 0$. Three vector codes (2,3), (3,2) and (1,-2) are shown in Figure 3. We use the term *vector* to refer to the graphical representation of a vector code. Given the one-to-one correspondence between vector and vector codes, we will use the two terms interchangeably in the rest of the paper.

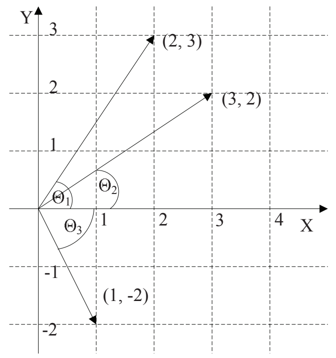


Fig. 3. Graphical representation of vector codes

Before formally defining vector order, we elaborate on the intuitive meaning behind it.

Intuitively, vector codes are ordered by $tan(\Theta)$ where Θ is the angle a vector makes with X axis. If we "rotate" a vector from the negative Y axis to the positive Y axis, Θ goes from $-90°$ (excluding $-90°$ itself) to $90°$ (excluding $90°$ itself) and $tan(\Theta)$ increases monotonously from $-\infty$ to ∞. In Figure 3, we have $tan(\Theta_3) < tan(\Theta_2) < tan(\Theta_1)$ and the three vector codes are ordered accordingly. Note that the condition $x > 0$ restricts vector codes to be in the first and forth quadrant where vector order is a total order.

Given two vector codes $A : (x_1, y_1)$ and $B : (x_2, y_2)$, vector preorder is defined as:

Definition 3.2 (Vector preorder): A precedes B in vector preorder (denoted as $A \preceq_v B$) if and only if $\frac{y_1}{x_1} \leq \frac{y_2}{x_2}$.

Vector equivalence is defined based on preorder.

Definition 3.3 (Vector equivalence): A is equivalent to B (denoted as $A \equiv_v B$) if and only if $A \leq B$ and $B \leq A$, or equivalently $\frac{y_1}{x_1} = \frac{y_2}{x_2}$.

Equivalence relation is both symmetric and transitive.

Lemma 3.1 (Symmetry of vector equivalence): If $A \equiv_v B$, then $B \equiv_v A$.

Lemma 3.2 (Transitivity of vector equivalence): Suppose $A \equiv_v B$ and $B \equiv_v C$, then $A \equiv_v C$.

Graphically speaking, if two vector codes are equivalent, then they have the same direction. As the following lemma implies, equivalence relation can be reduced to natural equality if two vector codes have the same X component.

Lemma 3.3: Suppose $A \equiv_v B$ and $x_1 = x_2$, then $y_1 = y_2$.

We refer to this special form of vector equivalence as equality.

Definition 3.4 (Vector equality): A is equal to B (denoted as $A{=}B$) if and only if $x_1 = x_2$ and $y_1 = y_2$.

Given vector preorder and equivalence, vector order can be defined as follows:

Definition 3.5 (Vector order): $A \prec_v B$ if and only if $A \preceq_v B$ and $A \not\equiv_v B$ ($\not\equiv_v$ is the negation of $\equiv_v$), equivalently, $\frac{y_1}{x_1} < \frac{y_2}{x_2}$ or $y_1 \times x_2 < x_1 \times y_2$.

Two vector codes are comparable under vector order if and only if they are not equivalent to each other. We say a set of vector codes is *inequivalent* if it does not contain two vector codes that are equivalent to each other.

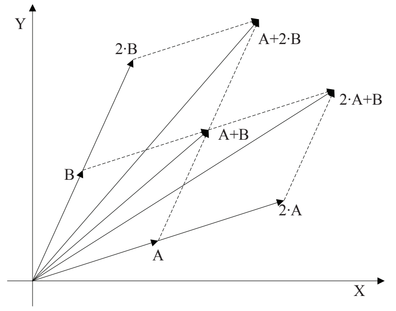


Fig. 4. Vector code addition and multiplication

The following lemma addresses a special case where vector order can be reduced to natural less than relation.

Lemma 3.4: Suppose $A \prec_v B$ and $x_1 = x_2$, then $y_1 < y_2$.

Under the constraint that $x > 0$, this lemma follows immediately from Definition 3.5.

Same as equivalence relation, vector order is transitive.

Lemma 3.5 (Transitivity of vector order): If $A \prec_v B$ and $B \prec_v C$, then $A \prec_v C$.

The following lemma establishes the connection between vector equivalence and vector order.

Lemma 3.6: If $A \equiv_v B$ and $B \prec_v C$, then $A \prec_v C$; If $A \prec_v B$ and $B \equiv_v C$, then $A \prec_v C$.

3.2 Vector code functions

We start by introducing two primitive functions to determine a new vector code that precedes or follows a given vector code $A : (x, y)$ in vector order.

- $BEF(A)$: return (x,y-1).
  //returns a vector code before A
- $AFT(A)$: return (x,y+1).
  //returns a vector code after A

It is readily verifiable from Lemma 3.4 that $BEF(A) \prec_v A \prec_v AFT(A)$.

To determine a new vector code that falls between two given vector codes in vector order, we introduce the following addition function.

Definition 3.6 (Vector code addition): Addition of two vector codes $A : (x_1, y_1)$ and $B : (x_2, y_2)$ is defined as:

$$A + B = (x_1 + x_2, y_1 + y_2) \qquad (1)$$

Multiplication function computes a vector code that is equivalent to the given vector code.

Definition 3.7 (Vector code and scalar multiplication): Multiplication of an integer r and a vector code $A : (x, y)$ is defined as:

$$r \cdot A = (r \times x, r \times y) \qquad (2)$$

Addition and multiplication of vector codes are illustrated in Figure 4. Intuitively, a vector code and its multiples are equivalent to each other and can be represented as vectors of the same direction. That is, they make the same angle with X axis and are equivalent with respect to vector order. Given two vector codes that are not equivalent, e.g. A and B, the addition of them should produce a vector code that falls between them in vector order. Because the angle that the resulting vector makes with the X axis is between those that A and B make. We formalize our observations with the following results.

Let $A : (x_1, y_1)$ and $B : (x_2, y_2)$ be two vector codes,

Lemma 3.7: Suppose $A \preceq_v B$, then $A \preceq_v (A + B) \preceq_v B$.

Proof: From $A \preceq_v B$, we have $y_1 \times x_2 \leq y_2 \times x_1$. Therefore, $y_1 \times (x_1 + x_2) = y_1 \times x_1 + y_1 \times x_2 \leq y_1 \times x_1 + y_2 \times x_1 = x_1 \times (y_1 + y_2)$. It follows that $A \preceq_v (A + B)$. Proof of the other half the lemma is similar, so we ignore it here. □

Theorem 3.1: Suppose $A \equiv_v B$, then $A \equiv_v (A + B) \equiv_v B$.

Proof: $A \equiv_v B$ implies both $A \preceq_v B$ and $B \preceq_v A$. It then follows from Lemma 3.7 that $A \preceq_v (A + B) \preceq_v B$ and $B \preceq_v (A + B) \preceq_v A$. Thus, $A \equiv_v (A + B) \equiv_v B$. □

Theorem 3.2: Suppose $A \prec_v B$, then $A \prec_v (A + B) \prec_v B$.

Proof: It follows from $A \prec_v B$ that $A \preceq_v B$ and $A \not\equiv_v B$. Therefore, $A \preceq_v (A + B) \preceq_v B$ (1). Assume $A \equiv_v (A + B)$, we have $y_1 \times (x_1 + x_2) = x_1 \times (y_1 + y_2)$ which can be simplified as $y_1 \times x_2 = x_1 \times y_2$, and thus, $A \equiv_v B$. It is a contradiction to our assumption, therefore $A \not\equiv_v (A + B)$ (2). In the same way, we can prove $(A + B) \not\equiv_v B$ (3). Combining (1), (2) and (3), the theorem follows. □

The following corollary generalizes Theorem 3.2.

Corollary 3.1: Given two vector codes A and B such that $A \prec_v B$, it follows that $A \prec_v \ldots \prec_v (3 \cdot A + B) \prec_v (2 \cdot A + B) \prec_v (A + B) \prec_v (A + 2 \cdot B) \prec_v (A + 3 \cdot B) \prec_v \ldots \prec_v B$.

4 Apply vector order to range-based labeling schemes

Application of vector order is through the process of order-preserving transformation.

4.1 Order-preserving transformation

The ranges in a set of containment labels come from a sequence of integers from 1 to $2n$ for an XML tree with n elements. For pre/post labeling scheme, there are two identical sequences of integers from 1 to n. Let Z denote the set of integers and V denote the set of vector codes, a transformation $f : Z \rightarrow V$ is order-preserving if the following condition holds: $f(i) \prec f(j)$ if and only if $i < j$ for $i, j \in Z$. In [9], we proposed recursive transformation which is order preserving. In this paper, we introduce a more efficient transformation method: linear transformation.

4.1.1 Linear transformation

Linear transformation $f : Z \rightarrow V$ is defined as follows,

$$f(i) = (1, i) \qquad for \quad i \in Z \tag{3}$$

It is an order-preserving transformation because, given any $i, j \in Z$ such that $i < j$, it follows from Lemma 3.4 that $(1, i) \prec_v (1, j)$. We use linear transformation to illustrate the application of vector order in this paper.

4.2 V-Containment labeling scheme

We apply linear transformation to containment labels and refer to the resulting labels as V-Containment labels. We have described in Theorem 3.2 how to insert a new vector code between two consecutive ones in vector order. Insertion processing with V-Containment labeling schemes is slightly different, as two vector codes have to be inserted at one time which form the range of the new element node. We introduce the concept of *granularity sum* to guide such insertions.

Definition 4.1 (Granularity sum): The granularity sum of a vector code $A : (x, y)$ (denoted by $GS(A)$) is defined as $x + y$.

The insertion algorithm is presented in Algorithm 1 which tries to minimize the granularity sums of new labels.

Algorithm 1: InsertTwoVectorCodes(A, B)

```
Data: $A$ and $B$ which are two vector codes
      satisfying $A \prec_v B$
Result: $C$ and $D$ such that $A \prec_v C \prec_v D \prec_v B$
1 if $GS(A) > GS(B)$ then
2     return (A+B) and (A+2·B);
3 else
4     return (2·A+B) and (A+B);
5 end
```

To study how insertion of a new node A can be processed with V-Containment labeling scheme, it is sufficient to consider the following three principles: (a) The range of A should be inside the range of A's parent; (b) The *start* of A should be be less than the *end* of its closest preceding sibling (if it exists) and (c) The *end* of A should be be greater than the *start* of its closest following sibling (if it exists). Based on containment property, (a) obviously holds. If (b) or (c) is violated, it means there is some range that A and its sibling(s) have in common. If some new node is inserted as a descendant of A or one of A's siblings and assigned a range that is inside the common range, it would be a violation of the tree structure. Moreover, (b) and (c) guarantee that A has the correct document order. In all cases, the *level* of A equals to the level of A's parent plus 1.

Example 4.1: In Figure 5, the solid circles represent the elements nodes that are initially in the XML tree. Their labels are transformed from containment labeling scheme through linear transformation. Consider inserting element node A before the first child of the root. The *start* and *end* of A should fall between the *start* of A's parent and *start* of A's following sibling, that is, (1,1) and (1,2). Since $GS(1,1) = 2 < 3 = GS(1,2)$, it follows from Algorithm 1 the *start* and *end* of A should be $(3,4)$ $(= (2 \times 1 + 1, 2 \times 1 + 2))$ and $(2,3)$

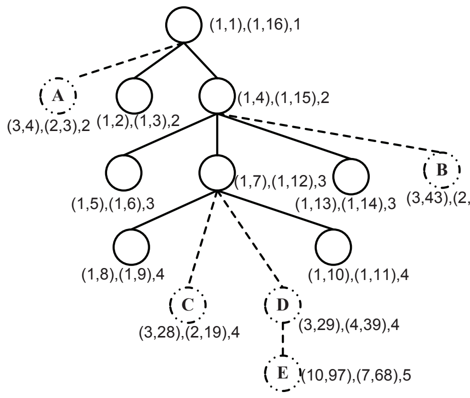


Fig. 5. Process Updates with V-Containment labeling scheme

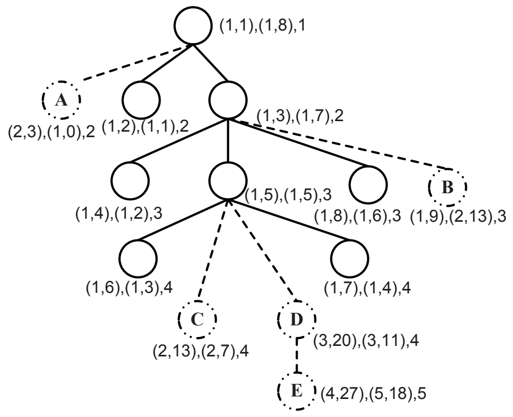


Fig. 6. Process Updates with V-Pre/post labeling scheme

$(= (1+1, 1+2))$. B is inserted after the last child of a node, its *start* and *end* should be bounded by the *end* of its preceding sibling and the *end* of its parent: (1,14) and (1,15). Applying Algorithm 1, the *start* and *end* of B should be $(3, 43)$ $(= (2 \times 1 + 1, 2 \times 14 + 15))$ and $(2, 29)$ $(= (1+1, 14+15))$. C is inserted between two consecutive element nodes. Its *start* and *end* should be between the *end* of its preceding sibling and the *start* of its following siblings. From Algorithm 1, the *start* and *end* of C should be $(3, 28)$ $(= (2 \times 1 + 1, 2 \times 9 + 10))$ and $(2, 19)$ $(= (1+1, 9+10))$. Similarly, the *start* and *end* of C should be $(3, 29)$ $(= (2 \times 1 + 1, 2 \times 9 + 10))$ and $(4, 39)$ $(= (1+1, 9+10))$. The range of D is confined by its parent's range. The *start* and *end* of C should be $(10, 97)$ $(= (2 \times 3 + 4, 2 \times 29 + 39))$ and $(7, 68)$ $(= (3+4, 29+39))$.

4.3 V-Pre/post labeling scheme

In V-Pre/post labeling scheme, insertion of a new node A can be processed based on two principles: (a)the *pre* of A should be between the *pre*s of two nodes that immediately precede and follow A during preorder traversal (if they exist) and (b)the *post* of A should be between the *post*s of two nodes that immediately precede and follow A during postorder traversal (if they exist).

Example 4.2: First we consider the insertion of A which is a leftmost insertion under the root. To maintain correct document order, the *pre* of A should be between the *pre* of A's parent and A's following sibling. That gives us (2,3) which is the sum of (1,1) and (1,2). In addition, to keep AD and PC relationships, the *post* of A should be less than the *post* of A's following sibling, that is (1,1). We therefore assign $BEF(1, 1) = (1, 0)$ to the *post* of A. Since there is no element nodes that follow B during preorder traversal, we assign $AFT(1, 8) = (1, 9)$ to the $B.pre$. In addition, $B.post = (2, 13) = (1, 6) + (1, 7)$. Insertions between two consecutive siblings (C and D) are processed in a similar manner. Insertion of a leaf node is more complicated with V-Pre/post labeling scheme than with V-Containment labeling scheme. Recall that in V-Containment labeling scheme, the label of the parent alone is sufficient to determine the label of the new leaf node. However, if we consider the insertion of D in Figure 6, its *pre* should fall between the *pre* of its parent and the *pre* of its parent's following sibling. In addition, the *post* of D is confined by the *post* of its parent and the *post* of its parent's preceding sibling. Thus, the label of D is determined by three labels.

5 Apply vector order to prefix-based labeling schemes

In this section, we present two prefix-based labeling schemes that are based on vector order.

5.1 V-Prefix labeling scheme

We provide a brief introduction of V-Prefix labeling scheme which is the most straight forward application of vector order to Dewey labeling scheme. It is derived from Dewey labeling scheme by transforming every dewey label into a sequence of vector codes through linear transformation.

The initial labeling of V-Prefix labeling scheme is shown in Figure 7, with solid circles representing the element nodes initially in the XML tree. All vector codes are enclosed by brackets for easy reference.

Given a V-Prefix label of the form $(x_1, y_1).(x_2, y_2) \ldots (x_m, y_m)$, we denote it as: $v_1.v_2 \ldots v_m$ where $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2)$ $\ldots$ $v_m = (x_m, y_m)$. Thus, V-Prefix label can be seen as a generalized Dewey label where every component is a vector code.

V-Prefix labels are ordered by V-Prefix order.

Definition 5.1 (V-Prefix order): Given two V-Prefix labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$, A precedes B in V-Prefix order (denoted as $A \prec_{vp} B$) if and only if one of the following two conditions holds:

C1. $m < n$ and $v_1 = w_1$, $v_2 = w_2$, $\ldots$, $v_m = w_m$.
C2. $\exists k \in [0, min(m, n)]$, such that $v_1 = w_1$, $v_2 = w_2$, $\ldots$, $v_{k-1} = w_{k-1}$ and $v_k \prec_v w_k$.

We illustrate how to process updates with V-Prefix labels with the following example.

Example 5.1: First we consider the leftmost insertion of element node A in Figure 7. A should have the same *parent_label* as its parent's label and a local order less

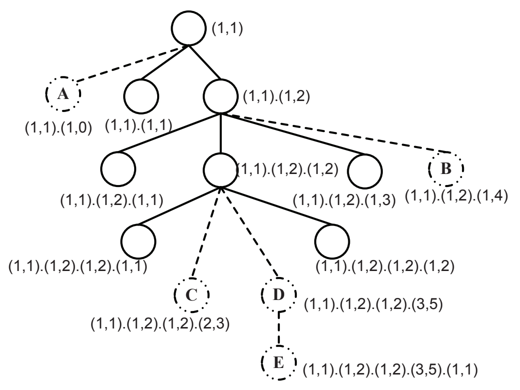


Fig. 7. Process Updates with V-Prefix labeling scheme

than (1,1). Thus, we get the new label of A by concatenating its parent's label (1.1) to $BEF(1,1) = (1,0)$. Since B is inserted at the rightmost position after (1.1).(1.2).(1.3), we derive its local order to be $AFT(1,3) = (1,4)$. C is inserted between two consecutive siblings. Its $parent\_label$ is the same as its parent's label whereas its local order should fall between the local orders of its two siblings. That is, (2,3)=(1,1)+(1,2). The local order of D is similarly computed: (3,5)=(2,3)+(1,2). We process the insertion of a leaf node (E) by concatenating its $parent\_label$ with an additional component, say, $(1,1)$.

5.2 DDE labeling scheme

We presented Dynamic DEwey (DDE) labeling scheme in [10]. Here we characterize DDE in terms of the order framework we have developed in this paper. DDE generalizes the idea of vector order and vector equivalence.

5.2.1 motivation

While V-Prefix labeling scheme appears to be the straight forward application of vector order to Dewey labeling scheme, its drawbacks are also obvious. Transforming from integer to vector codes doubles the number of components of the labels and increases the overall label size.

If different labeling schemes were to be used for static and dynamic XML documents, different storage and query mechanisms need to be enforced, making updating and querying complicated. To make matters worse, the system administrator bears the burden of deciding which of the documents are dynamic or static. This in general is a difficult, if not impossible task as the updating frequency of a document can vary according to time: a document can, for example, be frequently updated for a period of time and remain unchanged after that. Here we introduce DDE labeling scheme which is dynamic enough to completely avoid re-labeling, while introducing minimum additional complexity to static documents.

5.2.2 initial labeling

Every DDE label is a sequence of integers separated by dots. The initial labeling of DDE labeling scheme is the same as Dewey (Figure 2). However, the semantic meanings of DDE and Dewey are very different. A Dewey label can be seen as a concatenation of local orders from the root to an element node whereas we interpret a DDE label as a sequence of vector codes that share a common X component. We will show that the directions of these vectors together with the ordering of them can uniquely determine a path from the root to an element node.

A more intuitive representation of DDE labels is defined in terms of vector codes.

Definition 5.2 (Vector representation of DDE label): Given a DDE label of the form $x.y_1.y_2 \ldots y_m$, its vector representation is $v_1.v_2 \ldots v_m$ where $v_1 = (x, y_1)$, $v_2 = (x, y_2) \ldots v_m = (x, y_m)$.
We can see that the first component of a DDE label is shared by the sequence of vector codes as the X component. Note that the root element 1 does not fit into this interpretation and has to be specially dealt with. We consider a DDE label as a generalized Dewey label where each component is a vector code (all of which share a common X component).

5.2.3 DDE label ordering

DDE labels are ordered by DDE order which can be defined as follows:

Definition 5.3 (DDE order): Given two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$, A precedes B in DDE order (denoted as $A \prec_{dde} B$) if and only if one of the following two conditions holds:

C1. $m < n$ and $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_m \equiv_v w_m$.
C2. $\exists k \leq min(m, n)$, such that $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_{k-1} \equiv_v w_{k-1}$ and $v_k \prec_v w_k$.

The label of the root is minimum with respect to DDE order, i.e. it precedes all other labels. DDE order can be seen as generalized lexicographical order where component wise comparison is based on vector equivalence and vector order.

DDE order is transitive.

Lemma 5.1 (Transitivity of DDE order): Given three DDE labels A, B and C such that $A \prec_{dde} B$ and $B \prec_{dde} C$, it follows that $A \prec_{dde} C$.

Proof: From Definition 5.3, $\prec_{dde}$ can imply one of two conditions. Therefore there are four cases to consider, which can be proved based on Lemma 3.5, Lemma 3.2 and Lemma 3.6. We ignore the details here. □

The equivalence relation on DDE labels can be defined as:

Definition 5.4 (DDE equivalence): Two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$ are equivalent (denoted as $A \equiv_{dde} B$) if and only if $m = n$ and $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_m \equiv_v w_m$.

Two DDE labels are comparable with respect to DDE order if and only if they are not equivalent. We say that a set of DDE labels is *inequivalent* if there does not exist two DDE labels in the set with equivalence relation. Let A and B be two distinct DDE labels from an inequivalent set of DDE labels, we have either $A \prec_{dde} B$ or $B \prec_{dde} A$ (not both).

Lemma 5.2 (Transitivity of DDE equivalence): Given three DDE labels A, B and C, if $A \equiv_{dde} B$ and $B \equiv_{dde} C$, then $A \equiv_{dde} C$.
This lemma easily follows from the transitivity of vector equivalence.

5.2.4 DDE label properties

A DDE label implicitly stores the level information as the number of components in that label. This property will remain true after arbitrary insertions and deletions.

Given two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_n$, we summarize the properties of DDE labels as follows:

P1 (AD Relationship). A is an ancestor of B if and only if $m < n$ and $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_m \equiv_v w_m$. (The case where A is the root always returns true.)
P2 (PC Relationship). A is the parent of B if and only if A is an ancestor of B and $m = n - 1$.
P3 (Document Order). A precedes B in document order if and only if $A \prec_{dde} B$.
P4 (Sibling Relationship). A is a sibling of B if and only if $m = n$ and $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_{m-1} \equiv_v w_{m-1}$.
P5 (LCA). The LCA of A and B is C, such that C is an ancestor of both A and B, and either (1) $|C| = min(m, n)$, or (2) $v_{|C|+1} \not\equiv_v w_{|C|+1}$.

From Lemma 3.3 and Lemma 3.4, $\equiv_v$ and $\prec_{dde}$ can be reduced to $=$ and $<$ respectively if two vector codes have the same X component. Such reductions can be applied to all the initial DDE labels because they all have 1 as their first component and, as we know, the first component serves as the X component for the sequence of vector codes in every DDE label. For example, AD relationship can be simplified for initial DDE labels as follows.

P1 (AD Relationship (for initial DDE labels)). A is an ancestor of B if and only if $m < n$ and $v_1 = w_1$, $v_2 = w_2$, ..., $v_m = w_m$.

It follows from the reduction that the initial DDE labels can be treated as Dewey labels which we consider to be tailored for static XML documents.

5.2.5 Correctness of initial labeling

Lemma 5.3: Based on DDE labeling scheme, the set of initial DDE labels is inequivalent.

Proof: We establish the proof by contradiction. Suppose the set of initial DDE labels is not inequivalent, there exist two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_m$, such that $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_m \equiv_v w_m$. However, since all the initial DDE labels start with 1, it follows that $v_1 = w_1$, $v_2 = w_2$, ..., $v_m = w_m$, which means A and B are the same. We have a contradiction here because all DDE labels are different in initial labeling. □
Since the set of initial DDE labels is inequivalent, it follows that any two of them are comparable with respect

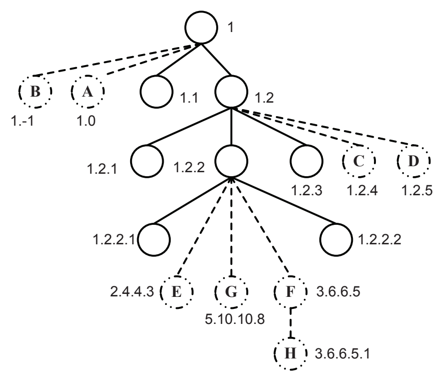


Fig. 8. Processing insertions with DDE labels

to DDE order. In addition, DDE order can be reduced to Dewey order for the initial DDE labels because all of them start with 1. The fact that our initial label assignment is the same as Dewey implies that document order is correct with respect to Dewey order and therefore DDE order. The same reasoning applies to all the other properties of DDE labels.

5.2.6 DDE label addition

To process dynamic insertions between DDE labels while preserving their relative order, we introduce addition operation on DDE labels. The addition operation is defined on DDE labels with the same number of components.

Definition 5.5 (DDE label addition): Given two DDE labels with the same number of components $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_m$, $A + B$ is defined as:

$$A + B = (v_1 + w_1).(v_2 + w_2) \ldots (v_m + w_m) \tag{4}$$

Note that the first integer in a DDE label, which is the common X component, only needs to be added once.

The following theorems formalizes important properties of the addition operation.

Theorem 5.1: Given two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_m$ such that A is a sibling of B and $A \prec_{dde} B$, then $A \prec_{dde} (A + B) \prec_{dde} B$.

Proof: Since A and B are siblings, we have $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_{m-1} \equiv_v w_{m-1}$. From Theorem 3.1, $v_1 \equiv_v (v_1 + w_1) \equiv_v w_1$, $v_2 \equiv_v (v_2 + w_2) \equiv_v w_2$, ..., $v_{m-1} \equiv_v (v_{m-1} + w_{m-1}) \equiv_v w_{m-1}$. In addition, $A \prec_{dde} B$ implies that $v_m \prec_v w_m$. It then follows from Theorem 3.2 that $v_m \prec_v (v_m + w_m) \prec_v w_m$. As a result, $A \prec_{dde} (A + B) \prec_{dde} B$. □

Theorem 5.2: Given two DDE labels $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_m$ such that $A \equiv_{dde} B$, then $A \equiv_{dde} (A + B) \equiv_{dde} B$.

Proof: From $A \equiv_{dde} B$, we have $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, ..., $v_m \equiv_v w_m$. Applying Lemma 3.2, we have $v_1 \equiv_v (v_1 + w_1) \equiv_v w_1$, $v_2 \equiv_v (v_2 + w_2) \equiv_v w_2$, ..., $v_m \equiv_v (v_m + w_m) \equiv_v w_m$, and therefore $A \equiv_{dde} (A + B) \equiv_{dde} B$. □

We use the following example to illustrate the properties of DDE labels that have been introduced so far.

Example 5.2: Consider the XML tree in Figure 8, the dotted circles represent the new nodes inserted into the XML tree. We ignore for now how their labels are generated. Node 1.2 is an ancestor of node E as $(1,2) \equiv_v (2,4)$ and $|1.2| < |E|$ ($|E|$ denotes the number of components in E). Node 1.2.2 is the parent of G as $(1,2) \equiv_v (5,10)$ and $|1.2.2| = |G| - 1$. $A \prec_{dde} E$ as $(1,0) \prec_v (2,4)$, so H precedes E in document order. E is a sibling of F because $|E| = |F|$ and $(2,4) \equiv_v (3,6)$. In addition, $E \prec_{dde} F$ as $(2,4) \equiv_v (3,6)$ and $(2,3) \prec_v (3,5)$. Note that G=E+F as $5.10.8 = 2.4.3 + 3.6.5$, since E is a sibling of F and E $\prec_{dde}$ F, we have $E \prec_{dde} G \prec_{dde} F$ based on Theorem 5.1. To verify, $E \prec_{dde} G$ as $(2,4) \equiv_v (5,10)$ and $(2,3) \prec_v (5,8)$, $G \prec_{dde} F$ as $(5,10) \equiv_v (3,6)$ and $(5,8) \prec_v (3,5)$.

5.2.7 Processing updates

Similar to Dewey labels, it is clear that the deletion of DDE labels does not affect the order of the other labels. The challenging part is how to handle insertions without re-labeling. Note that, like ORDPATH, we extend the domain of component values of DDE labels to positive number, negative number and 0. However, since ORDPATH only uses odd numbers at initial labeling, its labels are not as compact as DDE and Dewey.

First we introduce how DDE labeling scheme processes insertions with an example.

Example 5.3: In Figure 8, node A is inserted before the first child of the root, we get its label 1.0 by decreasing the local order of 1.1 by 1. Node B is then inserted before A and its label is therefore 1.-1. Node C is inserted after the node with label 1.2.3, we get its label 1.2.4 by adding 1 to the local order of 1.2.3. Similarly, the label of node D is 1.2.5. Node E is inserted between two nodes with labels 1.2.2.1 and 1.2.2.2 and its label is 2.4.4.3 which equals to 1.2.2.1+1.2.2.2. Likewise, the labels of node F and G are 3.6.6.5 (2.4.4.3+1.2.2.2) and 5.10.10.8 (2.4.4.3+3.6.6.5) respectively. Node H is inserted as the child of leaf node 3.6.6.5, its label is the concatenation of its parent's label and 1.

Among the insertions shown Figure 8, we consider the correctness of the following special cases obvious because the resulting labels are almost the same as the initial labeling, so proofs are ignored here.

- **Leftmost insertion.** When a new node is inserted before node $A : v_1.v_2 \ldots v_m$ where A is the first child of a node, we assign label $v_1.v_2 \ldots BEF(v_m)$ to this node.
- **Rightmost insertion.** When a new node is inserted after node $A : v_1.v_2 \ldots v_m$ where A is the last child of a node, we assign label $A : v_1.v_2 \ldots AFT(v_m)$ to this node.
- **Insertion below a leaf node.** When a new node is inserted below a leaf node $A : v_1.v_2 \ldots v_m$, we assign label $A : v_1.v_2 \ldots v_m.1$ to this node.

In general, insertions can be made between any two consecutive siblings.

- **Insertion between two consecutive siblings.** When a new node is inserted between two consecutive siblings with labels A and B, we assign label $A+B$ to this node.

We prove the correctness of this case in Section 5.2.8.

5.2.8 Correctness

By definition, two DDE labels A and B have sibling relationship if and only if their $parent_label$s are equivalent. Given Theorem 5.2, the $parent\_label$ of $A+B$ is equivalent to both A and B, which from transitivity of DDE equivalence (Lemma 5.2), implies that $A+B$ is a sibling of A, B and all siblings of A and B. Sibling relationship is therefore correctly maintained.

A DDE label C is an ancestor of another DDE label A if and only if C is equivalent to a proper prefix of A. We have shown that, the $parent\_label$ of $A+B$ is equivalent to the $parent_label$s of A and B if A and B are siblings. As a result, transitivity of DDE equivalence (Lemma 5.2) indicates that, any ancestor of A and B is equivalent to a proper prefix of $A+B$ and is therefore an ancestor of $A+B$. The insertion is also correct with respect to PC relationship because the number of components of a DDE label is kept the same as the $level$ of the corresponding element node.

The correctness of DDE insertion with respect to document order follows from Theorem 5.1, Lemma 5.1 and the following lemma.

Lemma 5.4: Given three DDE labels A and B and $C = A+B$, such that A is a sibling of B and $A \prec_{dde} B$, if A' is a descendant of A, then $A' \prec_{dde} C$.

Proof: We denote A and B as $A : v_1.v_2 \ldots v_m$ and $B : w_1.w_2 \ldots w_m$ respectively. From A is a sibling of B and $A \prec_{dde} B$, we have $v_1 \equiv_v w_1$, $v_2 \equiv_v w_2$, $\ldots$, $v_{m-1} \equiv_v w_{m-1}$, $v_m \prec_v w_m$. It follows from Theorem 5.2 and Theorem 5.1 that $v_1 \equiv_v (v_1 + w_1)$, $v_2 \equiv_v (v_2 + w_2)$, $\ldots$, $v_{m-1} \equiv_v (v_{m-1} + w_{m-1})$, $v_m \prec_v (v_m + w_m)$. Since A' is a descendant of A, we can denote A' as $v'_1.v'_2 \ldots v'_m \ldots v'_{n-1}v'_n$ where $v'_1 \equiv_v v_1$, $v'_2 \equiv_v v_2$, $\ldots$, $v'_{m-1} \equiv_v v_{m-1}$, $v'_m \equiv_v v_m$. From Lemma 3.2 and Lemma 3.6, $v'_1 \equiv_v w_1$, $v'_2 \equiv_v w_2$, $\ldots$, $v'_{m-1} \equiv_v w_{m-1}$, $v'_m \prec_v w_m$. Thus, $A' \prec_{dde} C$. □

5.2.9 Compact DDE (CDDE)

In [10], we have introduced Compact DDE (CDDE) which is designed to enhance the performance of DDE for insertions. By defining a one-to-one mapping from CDDE labels to DDE labels, we can adapt the order and properties of DDE labels to CDDE labels. How CDDE labels are ordered is essentially the same as DDE labels.

6 Qualitative comparison

6.1 Order and update

We summarize the orders of the labeling schemes in Table 1. For range-based labeling schemes, how the ranges are ordered determines if they can avoid re-labeling or not. Both lexicographical order and vector order are dynamic enough to make the labeling schemes persistent,

Labeling scheme	Order	Component-wise equality	Component-wise order
Containment	natural	NA	NA
Pre/post	natural	NA	NA
Prime	natural	NA	NA
QED-containment	lex	NA	NA
QED-Pre/post	lex	NA	NA
V-Containment	vector	NA	NA
V-Pre/post	vector	NA	NA
Dewey	lex	natural	natural
VLEI-Dewey	generalized lex	natural	lex
QED-Dewey	generalized lex	natural	lex
ORDPATH	generalized lex	natural	lex
V-Prefix	generalized lex	natural	vector
DDE	generalized lex	v-equivalence	vector
CDDE	generalized lex	v-equivalence	vector

TABLE 1
Summary of orders of different labeling schemes

but natural order is weak against insertions. For prefix-based labeling schemes, it is the component-wise order that determines if a labeling scheme is dynamic. As we can see, generalized lexicographical order can be used to characterize existing prefix-based dynamic labeling schemes.

In addition to different orders, it is worth noting the inherent differences between prefix-based and range-based labeling schemes. Compared with range-based labeling schemes, an obvious advantage of prefix-based labeling schemes is its ability to determine Sibling and LCA relationships. However, the performance of prefix-based labeling schemes is sensitive to the structure of the XML documents as the size of a prefix label increases linearly with its level. Range-based labeling scheme, on the other hand, perform consistently regardless of the depth of the XML tree.

6.2 Analysis of update and query performance

Although natural order is easy to compare, it is too rigid to allow dynamic insertions without re-labeling. Lexicographical order appears to be more robust because, intuitively, both the value of each component and the number of components contribute to the ordering of labels. Insertion between two components that are consecutive in value can be accommodated by extending the number of components. However, frequent extensions of components can lead to significant increase in the overall size. For example, QED-based labeling schemes perform poorly for ordered insertions with increase in length at 2 bits per insertion.

In addition, QED based labeling schemes come with additional encoding costs. That is, the time and computational costs spent on transforming containment, pre/post or Dewey labels to the corresponding QED codes. The process is especially complicated for Dewey labels, considering that the encoding has to be applied to every sibling group from root to leaf. Each component in ORDPATH labeling scheme, as we have seen, consists of a variable number of even numbers followed by an odd number. This fact complicates the processing of ORDPATH labels in several ways. First of all, all ORDPATH labels in the initial labeling have to skip even numbers, which makes them less compact than Dewey. Moreover, the number of components in an ORDPATH label do not necessarily reflect the level of the associated element nodes. We have to count the number of odd numbers in an ORDPATH label to derive the level information. This also leads to more complicated relationship computation such as PC and Sibling, even if the XML document does not get updated at all.

Dataset	Size (MB)	Total No. of nodes	Max/average fan-out	Max/average depth
XMark	113	1666315	25500/3242	12/6
Nasa	23.8	476646	2435/225	10/7
Treebank	85.4	2437666	56384/1623	36/8
DBLP	127	3332130	328858/65930	6/3

TABLE 2
Test data sets

7 Experiments and Results

7.1 Experimental setup

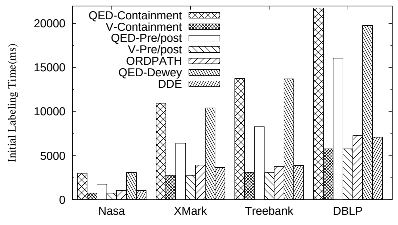
We focus on the comparison of our vector order-based labeling schemes against QED-based labeling schemes and ORDPATH which are all persistent labeling schemes. It has been shown that persistent labeling schemes have much lower updating time than labeling schemes that require re-labeling[6].

The evaluation of these labeling scheme was performed with XMark Benchmark[17], Nasa, Treebank and DBLP [16] data sets and their characteristics are shown in Table 2. All the experiments were conducted on a 2.33GHz dual-core PC with 4 GB of RAM.

7.2 Initial labeling

The evaluation of initial labeling is shown in Figure 9, with measures of label generation time and label size. It can be seen that the label generation time of vector order-based and ORDPATH labeling schemes are approximately the same, which is dominated by scanning the the document once. QED-based labeling schemes have much higher label generation time, because, in addition to scanning the document, they have to perform encoding into QED codes.

The labels of vector order-based and ORDPATH labeling schemes are stored in compressed ORDPATH format[12]. QED-based labeling schemes use their own physical storage format, with 0 as the separator between every two QED codes. The label size of range-based labeling schemes is generally larger than that of prefix-based labeling schemes. For range-based labeling schemes, the label size of QED-based labeling schemes is slightly less than that of vector order-based ones. For prefix-based labeling schemes, DDE has the most compact initial label size for all the four data sets.


(a) Label generation time

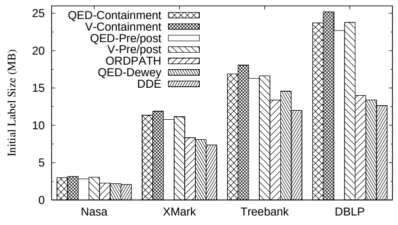
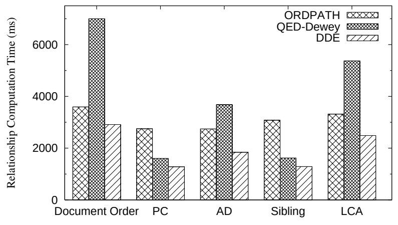

(b) Initial label size

Fig. 9. Initial Labeling


(a) Prefix-based labeling schemes

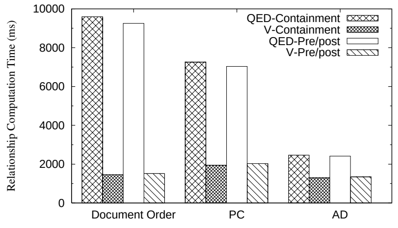

(b) Range-based labeling schemes

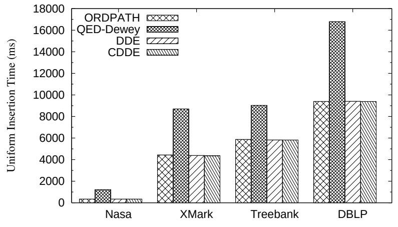
Fig. 10. Querying initial labels

7.3 Querying static document

We test the query performance on all the four data sets. We present the results from Treebank and DBLP and the other two data sets shown similar trends. Without any updates, the labels used for processing queries remain the same as the initial labels. We evaluate the query performance on initial labels by computing the most commonly used five relationships: document order, AD, PC, sibling and LCA. We choose the first 10000 labels from the initial labels of Treebank data set in document order and, for each pair of the labels, we compute all the five relationships. Note that as pointed out in [11], the LCA of a set of nodes is effectively the LCA of the first and the last node of the set in document order. Therefore we only test the computation of the LCA of two labels rather than many labels.

The querying time for prefix-based labeling schemes are shown in Figure10 (a) on all the five relationships. CDDE is not shown here because its performance is the same as DDE for static documents. While QED-Dewey is more efficient than ORDPATH for computing PC and sibling relationships, it is significantly slower for comparing document order and less efficient for AD relationship and LCA computation. For all the five relationships, our DDE outperforms ORDPATH and QED-Dewey.

Range-based labeling schemes are evaluated based on three relationships including document order, AD and PC. Sibling and LCA are excluded because they are not supported by range-based labeling schemes. Results in Figure 10 (b) show that V-Containment and V-Prefix support the three relations more efficiently than QED-based labeling schemes.


(a) Updating time

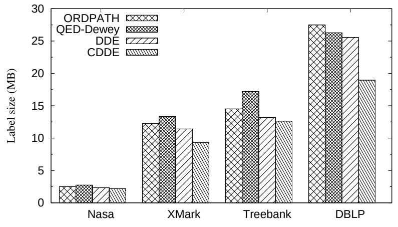
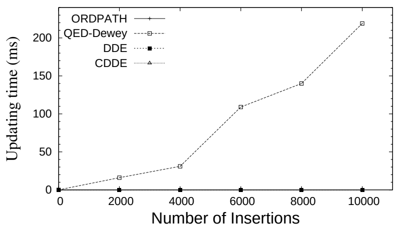

(b) Label size

Fig. 11. Uniform insertions

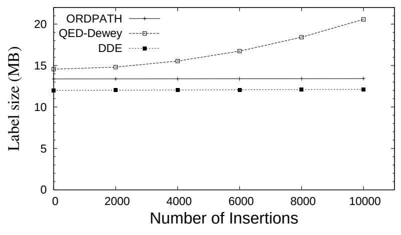
7.4 Update processing

7.4.1 Uniform insertions

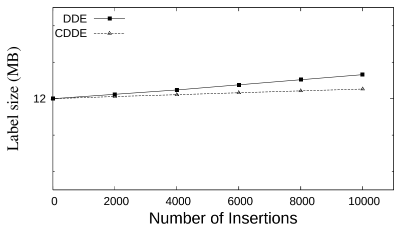
We test with insertions made uniformly between every two consecutive siblings. How these labeling schemes respond to uniform insertions is shown in Figure 11. The insertion time of ORDPATH is approximately the same as our DDE and CDDE whereas QED shows a slower updating time, as illustrated in Figure 11 (a). In Figure 11 (b), the comparison of label size after uniform insertions remains similar to that for the initial labels (Figure 9 (b)), with CDDE giving the most compact labels. The comparison of range-based labeling schemes and query performance after uniform insertions are ignored here, since the quality of these labeling schemes is not much affected by uniform insertions.



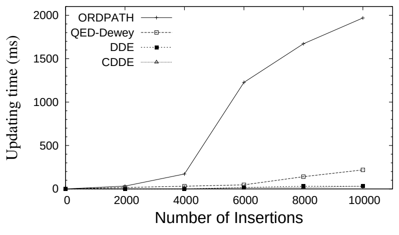
(a) Updating time



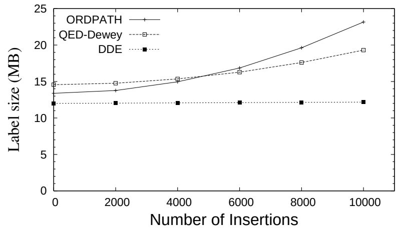
(b) Label size



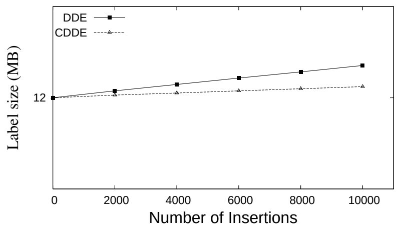
(c) Label size



(d) Updating time



(e) Label size



(f) Label size

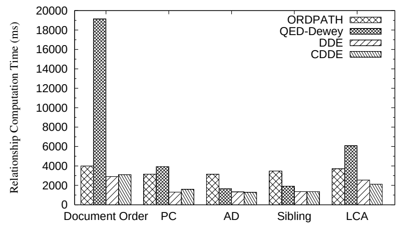
Fig. 12. Random skewed insertions

7.4.2 *Skewed insertions*

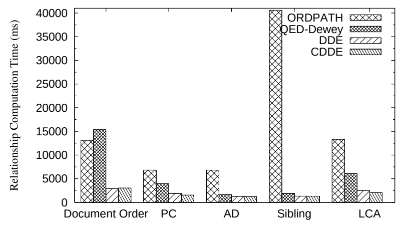
We classify skewed insertions into two different cases that are common in practice:

- **Ordered skewed insertion** refers to repeatedly inserting before or after a particular node.
- **Random skewed insertion** refers to repeatedly inserting between two nodes in random order.

Compared with uniform insertions, skewed insertions can have a more significant impact on the resulting qualities of labels. Figure 12 (a) (b) and (c) shows the updating cost and label size after ordered skewed insertions. The insertion time of ORDPATH, DDE and CDDE are negligible and their label sizes only increase slightly. In contrast, QED-Dewey has relatively higher updating time and its label size has shown a much higher increase. This result conforms to our previous discussions that the lengths of QED codes can increase at 1 or 2 bits per insertion in case of ordered skewed insertion, resulting in the fast increase of the overall label size. The results for random skewed insertions are shown in Figure 12 (d), (e) and (f). The updating time and label size of ORDPATH increase at a much faster rate than the other labeling schemes. This is because random skewed insertions greatly increase the amount of 'caret's that are needed to be used in ORDPATH labels. For both types of insertions, our DDE and CDDE have shown the best performance in terms of updating time and label size. In addition, the label size of CDDE increases at a slower rate than DDE, which is what we have expected. Figure 14 and 15 show the response of range-based labeling schemes to ordered skewed insertions. The result for random skewed insertions is similar. It can be seen that V-Containment and V-Prefix labeling schemes are little affected by ordered insertion sequence while QED encoded range-based labeling schemes have shown much higher growth rate in label size.



(a) Ordered skewed insertions

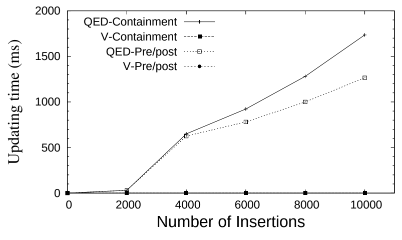


(b) Random skewed insertions

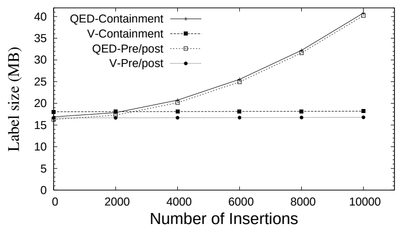
Fig. 13. Relationship computation time

7.5 Querying dynamic document

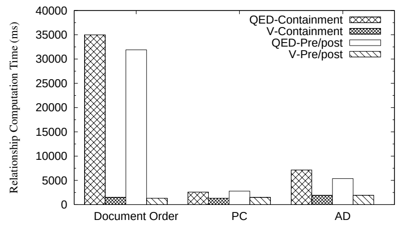
To compare the query performance on dynamic XML documents, we adopt the same settings as the static case except the 10000 labels chosen include 2000 labels that are newly inserted. Figure 13 (a) gives the comparison of relationship computation time after ordered skewed insertions. Given the fast increase of QED-Dewey label size, it conforms to our expectation that its query response time also increases significantly, especially for document order. The comparison after random skewed insertions is shown in Figure 13 (b) where the query response time of ORDPATH increases significantly, particularly for sibling relationship. Nevertheless, our DDE and CDDE have demonstrated robust performance regardless of the order and number of insertions. Their



(a) Updating time

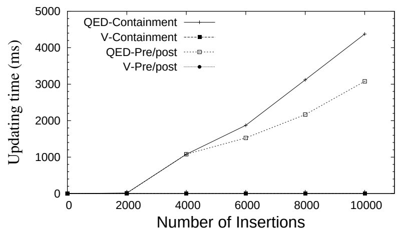


(b) Label size

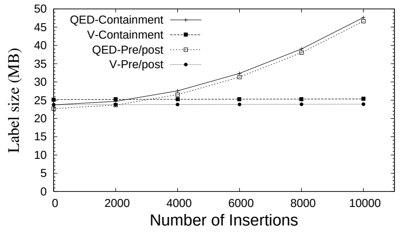


(c) Querying time

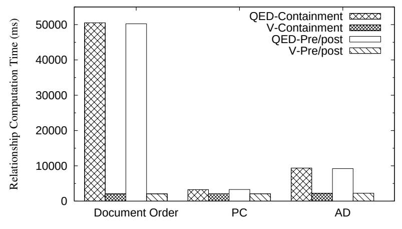
Fig. 14. Updating Treebank data set



(a) Updating time



(b) Label size



(c) Querying time

Fig. 15. Updating DBLP data set

query response times are least affected after both types of skewed insertions. We have similar observation for range-based labeling schemes in Figure 14 and 15.

8 Conclusion

In this paper, we study the problem of designing dynamic XML labeling scheme from a novel order-centric approach. A framework has been developed to characterize different labeling schemes and provide insight into their behaviors for updates, paving the way for future studies on this topic. After pointing out the limitations of existing labeling schemes, we propose vector order to solve this problem. We illustrate how vector order can be applied to both range-based and prefix-labeling to process insertions. Our solutions are highly efficient for static documents, while being able to completely avoid re-labeling when updates take place. In addition, experimental results demonstrate the resilience of our solutions against skewed insertions. In future we hope to apply our techniques to other applications involving order-sensitive updates.

References

[1] T. Amagasa and M. Yoshikawa and S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. In ICDE, 2003.
[2] E. Cohen and H. Kaplan and T. Milo. Labeling Dynamic XML Trees. In SPDS, 2002.
[3] Kobayashi et al. VLEI Code: An Efficient Labeling Method for Handling XML Documents in an RDB In *ICDE*, 2005
[4] C. Li and T. W. Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In CIKM, 2005.
[5] C. Li and T. W. Ling and M. Hu. Efficient Processing of Updates in Dynamic XML Data. In ICDE, 2006.
[6] C. Li and T. W. Ling and M. Hu. Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String. In VLDB J., 2008.
[7] I. Tatarinov and S. Viglas and K. S. Beyer and J. Shanmugasundaram and E. J. Shekita and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In SIGMOD, 2002.
[8] X. Wu, M. L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *ICDE*, 2004.
[9] L. Xu and Z. Bao and T. W. Ling. A Dynamic Labeling Scheme Using Vectors. In DEXA, 2007.
[10] L. Xu and T. W. Ling and H. Wu and Z. Bao. DDE: from dewey to a fully dynamic XML labeling scheme. In SIGMOD, 2009.
[11] C. Sun, C.-Y. Chan, and A. K. Goenka. Multiway SLCA-based Keyword Search in XML Data. In *WWW*, 2007.
[12] Patrick O'Neil and Elizabeth O'Neil and Shankar Pal and Istvan Cseri and Gideon Schaller and Nigel Westbury. ORDPATHs: Insert-friendly XML Node Labels. In SIGMOD, 2004.
[13] Paul F. Dietz. Maintaining Order in a Linked List. In Annual ACM Symposium on Theory of Computing, 1982.
[14] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In VLDB, 2001.
[15] S. Abiteboul and S. Alstrup and H. Kaplan and T. Milo and T. Rauhe. Compact Labeling Scheme for Ancestor Queries. In SIAM J. Comput, 2006.
[16] University of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/
[17] XMark - An XML Benchmark Project. http://monetdb.cwi.nl/xml/downloads.html
[18] C Zhang and J. F. Naughton and D. J. DeWitt and Q. Luo and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In SIGMOD, 2001.

PLACE
PHOTO
HERE

Liang Xu Liang Xu joined National University of Singapore in 2002. After an undergraduate degree in Compute science, he is now a PhD candidate and a Graduate Fellowship recipient at School of Computing, National University of Singapore. His current research focuses on XML labeling and query processing.

PLACE
PHOTO
HERE

Tok Wang Ling Tok Wang Ling received his PhD in Computer Science from the University of Waterloo (Canada). He is a professor of the Department of Computer Science at the National University of Singapore. His research interests include Data Modeling, ER approach, Normalization Theory, and Semistructured Data Model and XML query processing. He has published more than 180 papers, co-authored a book, co-edited a book, and co-edited 9 conference proceedings. He is an ACM Distinguished Scientist, a senior member of IEEE, and an ER Fellow.

PLACE
PHOTO
HERE

Huayu Wu Huayu Wu is a PhD student in the Department of Computer Science, School of Computing at the National University of Singapore. He received his B.Sc. (Honors) degree in the School of Computing at the National University of Singapore in 2006. His research interest is database technologies, with specification in XML twig pattern query processing, XML keyword search, semantic approach in XML query processing and XML document labeling. He is advised by Professor Tok Wang Ling.