A Dynamic Labeling Scheme using Vectors

Liang Xu, Zhifeng Bao, Tok Wang Ling

School of Computing, National University of Singapore
{xuliang, baozhife, lingtw}@comp.nus.edu.sg

Abstract. The labeling problem of dynamic XML documents has received increasing research attention. When XML documents are subject to insertions and deletions of nodes, it is important to design a labeling scheme that efficiently facilitates updates as well as processing of XML queries. This paper proposes a novel encoding scheme, vector encoding which is orthogonal to existing labeling schemes and can completely avoid re-labeling. Extensive experiments show that our vector encoding outperforms existing labeling schemes on both label updates and query processing especially in the case of skewed updates. Besides, it has the nice property of being conceptually easy to understand through its graphical representation.

1 Introduction

XML[6] has become a standard to represent and exchange data on the web, and there is a lot of interest in query processing over XML data. The techniques used to facilitate XML queries can be classified into two categories: structural index approach[10] and labeling approach[12, 4, 11]. We focus on labeling approach which requires smaller storage space, yet efficiently determines ancestor-descendant(A-D) and parent-child(P-C) relationships between any two nodes in the XML documents.

Although most existing labeling schemes work well on querying static XML documents, their performances degrade significantly for dynamic XML documents as updating requires re-labeling[12, 4, 11] or label size increases very fast for skewed insertions although re-labeling can be avoided[8]. When XML documents are dynamic, it is of great interest to design a labeling scheme that can avoid re-labeling while having controllable size for skewed insertions.

The main contributions of this paper are summarized as follows:

- We propose a novel compact labeling scheme: vector encoding which can be applied to different labeling schemes and is easy to understand.
- Vector encoding completely avoids re-labeling for updates in XML doc.
- We conduct experiments to show that vector encoding performs better than existing schemes, especially in the case of skewed insertions.

2 Related Work

Current labeling schemes include containment scheme[12], prefix scheme[4] and prime scheme[11]. Due to space constraint we only focus on containment scheme and QED encoding which are most relevant to this paper.

Based on **containment labeling scheme**[12], every node is assigned three values: "start", "end" and "level". For any two nodes u and v, u is an ancestor of v iff $u.start < v.start$ and $v.end < u.end$. Node u is the parent of node v iff u is an ancestor of v and $v.level - u.level = 1$. Although containment scheme is efficient for determining A-D and P-C relationships, the insertion of a node n will lead to re-labeling of all the ancestor nodes of n and all the nodes after n in document order. To solve the re-labeling problem, [5] uses Float-point values for the start and end of the intervals. However, in practice, Float-point is represented physically with a fixed number of bits. As a result, at most 18 nodes can be inserted at a fixed place when consecutive integer values are used for initial labeling.

[8] proposes a novel encoding method: **QED encoding** that completely avoids re-labeling. Four numbers "0", "1", "2" and "3" are used for encoding and each number is stored with two bits, i.e. "00", "01", "10" and "11". The number "0" is reserved as the separator. An important feature of QED is that it is based on lexicographical order, i.e. "0" $\prec$ "1" $\prec$ "2" $\prec$ "3". This encoding scheme allows a QED code to be inserted between any two existing QED codes while preserving lexicographical order. For example, "113" can be inserted between "112" and "12" whereas "1122" can be inserted between "112" and "113". QED encoding method is orthogonal to existing labeling schemes. However, the label size of QED increases very fast in the event of skewed insertion. Especially in the case that new codes are inserted after a fixed code, the size of the new code increases by 2 bits per insertion.

3 Preliminaries

For the complete proofs of the theorems in this section, please refer to [9] which is an extended version of this paper. We ignore most of the proofs due to space constraint.

A *vector* is an object with magnitude and direction. A two dimensional vector consists of binary tuples and is represented as (x, y). In this paper, to make design simple and avoid precision problem, we only consider vector whose x and y components are positive integers. Figure 1(a) gives a graphical interpretation of a vector V that lies in the first quadrant of the X-Y plane.

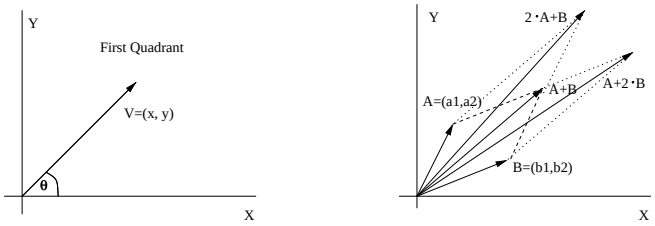
Let A=(x_1, y_1) and B=(x_2, y_2) be two vector. *Addition* of A and B and *Multiplication* of a scalar r and a vector A are defined as:

$$A + B = (x_1 + x_2, y_1 + y_2) \tag{1}$$

$$r \cdot A = (r * x_1, r * y_1) \tag{2}$$

Figure 1(b) shows the graphical representation of vectors A, B, A+B, 2·A+B and A+2·B.

Definition 1. ***(Gradient)*** *The* Gradient *of a vector V=(x, y) (denoted by G(v)) is defined as y/x.*



(a) a vector in the first quadrant (b) vector addition

Fig. 1. Graphical representation of vector and vector addition

In Figure 1(a), vector V makes an angle Θ with the X axis. The *Gradient* of V is y/x, or equivalently, tan(Θ).

Theorem 1. *Given two vectors A=(x_1, y_1) and B=(x_2, y_2) in the first quadrant of the x-y plane, G(A) > G(B) iff $y_1x_2 > x_1y_2$.*

Example 1. G((19,5))=5/19; G((19,5))>G((16,3)) since $5 \times 16 > 19 \times 3$.

It is important to note that although *Gradient* is defined in terms of division, the comparison of the *Gradients* of two vectors can be done via multiplication.

Theorem 2. *Let A, B, C be three vectors in the first quadrant of the x-y plane such that C=A+B and G(A) > G(B), then G(A) > G(C) > G(B).*

Proof. assume that A=(x_1, y_1) and B=(x_2, y_2), then C=($x_1 + x_2, y_1 + y_2$). Since G(A) > G(B), we have from Theorem 1, $y_1x_2 > x_1y_2$. Therefore,

$$G(C) = \frac{y_1 + y_2}{x_1 + x_2} = \frac{y_1x_2 + y_2x_2}{(x_1 + x_2)x_2} > \frac{x_1y_2 + y_2x_2}{(x_1 + x_2)x_2} = \frac{y_2}{x_2} = G(B) \qquad (3)$$

Similarly, we can prove that G(A) > G(C). Therefore, G(A) > G(C) > G(B).

Example 2. The sum of vectors (19,5) and (16,3) is (35,8), G((19,5))>G((35,8)) since $35 \times 5 > 19 \times 8$; G((35,8))>G((16,3)) since $16 \times 8 > 35 \times 3$.

Theorem 3. *Let A, B be two vectors in the first quadrant of the x-y plane such that G(A) > G(B), we can find infinite number of vectors whose Gradients are between G(A) and G(B).*

4 Vector Encoding

In this section, we introduce our vector encoding scheme which completely avoids re-labeling upon insertion. Table 1 shows different encoding schemes for numbers from 1 to 18. Details of how QED encoding is performed are in [8]. For vector encoding, we first assign vector (1,0) to the start position in the range and (0,1) to the end position. Then we work recursively by assigning the middle position of the current range a vector that equals to the sum of two vectors that correspond to the start and end position in each iteration. The formal encoding algorithm is presented in Algorithm 1.

Decimal number	QED	vector	Gradient (accurate to 0.01)	Decimal number	QED	vector	Gradient (accurate to 0.01)
1	112	(1,0)	0	10	223	(1,1)	1
2	12	(5,1)	0.2	11	23	(3,4)	1.33
3	122	(4,1)	0.25	12	232	(2,3)	1.5
4	13	(3,1)	0.33	13	3	(3,5)	1.67
5	132	(2,5)	0.4	14	312	(1,2)	2
6	2	(2,1)	0.5	15	32	(2,5)	2.5
7	212	(5,3)	0.6	16	322	(1,3)	3
8	22	(3,2)	0.67	17	33	(1,4)	4
9	222	(4,3)	0.75	18	332	(0,1)	$+\infty$

Table 1. Comparison of different encoding schemes

Theorem 4. *Let I and J be two decimal numbers and V_I and V_J be their corresponding vector codes generated by Algorithm 1, we have: $I<J$ iff $G(V_I)<G(V_J)$.*

Example 3. Given that the range of integers is from 1 to 18, we assign vector (1,0) (of *Gradient* 0) to the start position in the range which is 1; and (0,1) (of *Gradient* $+\infty$) to the end position in the range which is 18, i.e. v(1)=(1,0) and v(18)=(0,1). Next we apply Algorithm 1 to recursively encode the remaining positions.

Iteration 1 The middle position in the range [1, 18] can be found by: $middle = \lceil(1+18)/2\rceil = 10$. Hence v(middle)=v(10)=v(1) + v(18)=(1,0) + (0,1)=(1,1).

Iteration 2 Now that the range [1,18] is divided into two ranges:[1, 10] and [10, 18]. The middle position of [1, 10] is $\lceil(1 + 10)/2\rceil = 6$; and the middle position of [10, 18] is $\lceil(10 + 18)/2\rceil = 14$. Therefore v(6)=(1,0)+(1,1)=(2,1) and v(14)=(1,1)+(0,1)=(1,2). This process continues until all the positions are encoded, we omit the remaining iterations here.

Definition 2. ***(vector order)*** *The order of vector encodings is based on the numerical ordering of the Gradients of the vectors.*

Table 1 also gives the *Gradients* of the vectors for each row (we define 1/0 to be $+\infty$). It can be seen that the numerical order of the *Gradient*s indeed follow the order of the decimal numbers. It is worth noting that the *Gradients* shown in Table 1 are for illustration purpose only. From theorem 1, we can compare the *Gradients* of two vectors using multiplication instead of division, our method does not involve the calculation of *Gradients* and therefore does not suffer from the float-point precision problem in [5].

4.1 Encoding Delimiters

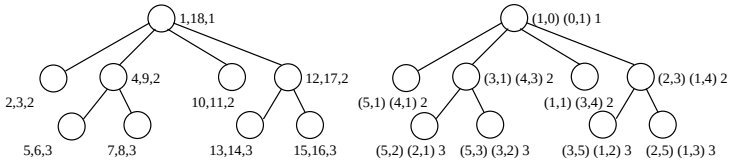
When labels are stored for future reuse, we need to encode delimiters. 0 is reserved as delimiter in QED whereas vector codes use UTF8[7] encoding to process delimiters. In UTF8, a variable number of bytes are used to encode different integer values. A vector V is stored sequentially as $V.x$, $V.y$ where $V.x$ and $V.y$ are encoded using UTF8 encoding.

Algorithm 1 *VectorEncoding*	**Algorithm 2** *LabelTheLeafNodeToBeInserted*
input: n is a positive integer	**input**: n is the leaf node to be inserted
output: return the vector codes in *vcode* for numbers from 1 to n	**output**: return *VContainment* label of n
//*vcode* is an array of n vectors	//$v1$, $v2$ are two bounding vectors
1: $vcode[0] = (1, 0)$	//l is the level of n
2: $vcode[n-1] = (0, 1)$	1: **if** n has preceding sibling(s) $v1$=$cps.endV$
3: RecEncoding($vcode, 0, n-1$)	//cps is the closest preceding sibling of n
4: return *vcode*	2: **else** $v1$=$p.startV$ //p is the parent of n
Procedure RecEncoding($vcode, start, end$)	3: **if** n has following sibling(s) $v2$=$cfs.startV$
1: $m = \lceil (start + end)/2 \rceil$	//cfs is the closest following sibling of n
2: **if** $m ==$ *end* return	4: **else** $v2$=$p.endV$
3: $mV = vcode[start] + vcode[end]$	5: return FindNewLabel($v1$,$v2$,l)
4: $vcode[m] = mV$	Procedure FindNewLabel($v1$,$v2$,l)
5: RecEncoding($vcode, start, m$)	1: **if** $GS(v1) > GS(v2)$
6: RecEncoding($vcode, m, end$)	return ($v1$+$v2$,$v1$+2·$v2$,l)
	2: **else** return (2·$v1$+$v2$,$v1$+$v2$,l)

4.2 Application of Vector Encoding Scheme

Our vector encoding scheme is orthogonal to specific labeling schemes. It can be applied to existing labeling schemes while keeping the original labeling order. In this paper we apply vector encoding scheme to containment scheme and the resulting labeling scheme is called *VContainment* scheme.

Example 4. Figure 2 shows an example of applying vector encoding to containment scheme. The *start* and *end* value of the original containment labels are replaced by their corresponding vector codes (see Table 1 for details). The resulting *VContainment* labels are of format ($startV, endV, level$) where $startV$, $endV$ are two vectors. It is easy to verify that the property of containment scheme holds. For example, Node((2,3),(1,4),2) is the parent of node((3,5),(1,2),3) as G(2,3) < G(3,5) < G(1,2) < G(1,4) and 2+1=3.



(a) containment labels (b) VContainment labels

Fig. 2. Applying vector encoding scheme to containment scheme

5 Support Updating

For dynamic XML documents, especially the ones that require frequent updates, it is important to make the update cost as low as possible. One of the most

important features of vector encoding scheme is that it can completely avoid re-labeling when updates take place. This section provides elaboration on how vector encoding handles updates efficiently. We start by showing how updates can be performed in *VContainment* scheme without re-labeling, then analyze how updates can be optimized in a general context.

5.1 Updating in VContainment Scheme

With *VContainment* scheme, the deletion of a leaf node or an internal node has no side effect. However, handling insertions may require some consideration. First we introduce a definition which we use to measure the size of a vector.

Definition 3. ***(Granularity Sum)*** *The Granularity Sum of a vector V = (x, y) (denoted by GS(v)) is defined as x+y.*

To find a vector between two vectors in vector order, we want its $GranularitySum$ to be as small as possible so that the resulting label size is small.

Inserting Leaf Node Assume all *VContainment* labels are of format ($startV$, $endV$, $level$). n is the node to be inserted and p is the parent of n. cps is its closest preceding sibling(if exists). cfs is its closest following sibling(if exists). If n is a leaf node, to maintain the correctness of *VContainment* scheme, the following inequalities should hold.

1. $G(p.startV) < G(n.startV) < G(n.endV) < G(p.endV)$
2. $G(cps.endV) < G(n.startV)$ 3. $G(n.endV) < G(cfs.startV)$

Note the second inequality is only applicable if n has preceding sibling(s), and the third inequality is applicable if n has following sibling(s). In any case, the $startV$ and $endV$ of n will be bounded by two vectors, we call these two vectors *bounding vectors*. The new label of n can be found by basically finding a pair of vectors between the two *bounding vectors*. Details on how the label of n is found are presented in Algorithm 2.

Inserting Non-leaf Node The case that n is a non-leaf node is similar to the previous case except that another inequality needs to be enforced. Assume fc is the first child of n and lc is the last child of n.

4. $G(n.startV) < G(fc.startV) < G(lc.endV) < G(n.endV)$

We ignore the details of insertion of non-leaf node here.

The core operation of Algorithm 2 is to find two vectors between $v1$ and $v2$ in terms of vector order. In Algorithm 2, the two vectors are either $v1 + v2$ and $v1 + 2 \cdot v2$ or $2 \cdot v1 + v2$ and $v1 + v2$. Based on Theorem 2, we can prove in both cases the two vectors are between $v1$ and $v2$ in vector order. Figure 1(b) shows the graphical representation of the vectors. It can be observed that if we keep inserting before or after a fixed node, the resulting label increases constantly by the *Granularity Sum* of that node. Although this method is simple and efficient, the resulting vector may not yield the minimum *Granularity Sum*. Analysis on the optimization of insertion will be presented in the next subsection. There is no re-labeling involved in the insertion, *VContainment* scheme can support efficient updates without re-labeling any existing labels.

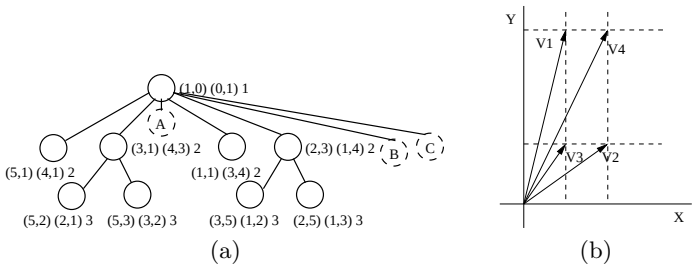


Fig. 3. Insertion in VContainment scheme

Example 5. In Figure 3(a), when inserting node A having both left sibling and right sibling, its *startV* and *endV* are bounded by *endV* of its closest left sibling and *startV* of its closest right sibling, i.e. (4,3) and (1,1). Moreover, GS(1,1)=2<7=GS(4,3). Therefore, the *startV* of A is $v1 + v2 = (5, 4)$ whereas *endV* is $v1+2\cdot v2 = (6, 5)$. When inserting node B which has only left sibling, its *startV* and *endV* are bounded by the *endV* of its closest left sibling the *endV* of its parent, i.e. (1,4) and (0,1). Therefore, the *startV* of B is $v1 + v2 = (1, 5)$ whereas *endV* is $v1+2\cdot v2 = (1, 6)$. Similarly, when we continue to insert C as the last child of the root, its *startV* is $v1+v2 = (1, 7)$ and *endV* is $v1+2\cdot v2 = (1, 8)$.

5.2 Analysis on Insertion

When vector encoding scheme is applied to different labeling schemes including containment scheme, the core operation of insertion is to find the vector between two consecutive vectors in vector order. The choice is not unique, actually there are infinitely many vectors possible(Theorem 3); however, to slow down the increase rate of labels, we would want to find the vector that has the smallest *Granularity Sum* possible. Although we can always use the sum of the two consecutive vectors, the resulting vector may not yield the minimum *Granularity Sum*. Theoretically speaking, the vector that has the smallest *Granularity Sum* can always be found through enumeration, but this can make insertion very expensive to perform. However, we have found that based on the relative positions of the two consecutive vectors, it may be possible to optimize insertion without incurring much additional computational cost. Assume the consecutive vectors are $A = (x_1, y_1), B = (x_2, y_2)$, the vector to be inserted is $C = (x, y)$, insertion may be optimized for the following cases.

Case 1 $x_1 = x_2$ or $y_1 = y_2$. For example, let A and B be V1 and V3 in Figure 3(b) respectively. Since $x_1 = x_2$, we can choose $x = x_1$ and y to be an integer between y_1 and y_2 when $y_1 > y_2 + 1$. When $y_1 = y_2 + 1$, we can choose C to be the sum of A and B. The case when $y_1 = y_2$ is similar.

Case 2 ($x1 < x2$ and $y1 > y2$) or ($x1 > x2$ and $y1 < y2$). For example , let A and B be V1 and V2 in Figure 3 (b) respectively. We can choose C to be V3=(x_1, y_2) or V4=(x_2, y_1) since both V3 and V4 are between V1 and V2 in vector order. V3 may be preferred since it has smaller *Granularity Sum*.

6 Experiments and Evaluation

We have implemented the *VContainment* scheme in JAVA and used SAX from Sun Microsystems as the XML parser. We compare our labeling scheme with QED-containment scheme as they both completely avoid re-labeling. The updating cost of previous labeling schemes[12, 4, 11] are much higher than QED as re-labeling is very expensive to perform[8].

XMark[3], Shakespeare's play[1], Treebank and DBLP[2] data sets have been used to compare the performance of the labeling schemes. Our experiments are performed on Pentium IV 3 GHz with 1G of RAM running on windows XP.

6.1 Label Generation

Data Set	File Size(MB)	No. of Nodes(K)	QED Time(Sec)	vector Time(Sec)	QED Size(MB)	vector Size(MB)
XMark	0.55	8.5	0.142	0.018	0.05	0.06
Shakespeare's play	2.16	49	0.86	0.26	0.31	0.39
Treebank	82	2437.7	33.8	9.1	19.2	27.0
DBLP	127	3332.1	50.9	14.6	26.9	37.8

Table 2. Comparison of label generation

To compare the label generation, we choose one of the documents from Shakespeare's play and enlarge it by 10 times. XMark document is generated using scaling factor 0.005. The time needed to generate labels mostly depends on the size the XML documents and the number of nodes in the XML documents. From the results in Table 2, as the size of the data set gets larger, the generation time of QED and vector labels increase accordingly. However, generating vector labels is much faster than QED as its generation mostly consists of simple calculations.

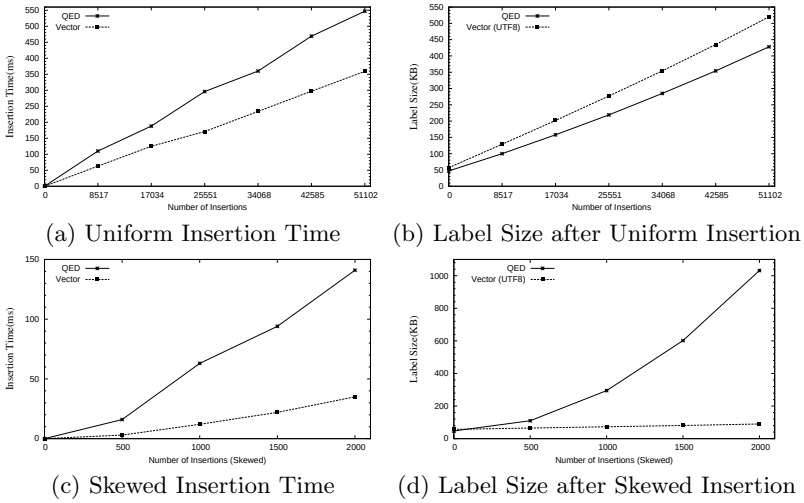
6.2 Uniform and Skewed Insertions

All the four data set have been used to test the performance of the two labeling schemes upon two kinds of insertions: uniform insertion and skewed insertion, and showed similar trends. Here we present the results for XMark data set.

For uniform insertions, firstly we insert one node between every two consecutive nodes. Then we gradually increase the insertions by one at a time up to six. The results are shown in Figure 4(a) and (b). The vector labels are represented using bit strings that correspond to the UTF8 representation of the labels to accommodate dynamic increase in size. But the overhead of UTF8 encoding makes the size of vector labels approximately 20 percent larger than that of QED (including initial labels). The insertion time of QED however is about 50 percent more than that of vector labels.

For skewed insertions, we keep inserting *mail* element after the last *mail* element whose parent is *mailbox*. The results (Figure 4(c) and (d)) illustrate more significant advantage of vector labels. The insertion time of QED is almost four times of that of vector labels while the size of QED labels increases much faster than vector labels upon insertions. Since for skewed insertions, the length of the new QED label increases by 1 or 2 bits per insertion, while the size of the

new vector label remains unchanged unless its value exceeds the current range in which case the label size increases by 1 byte. However, such increases occur infrequently and the size of vector labels increases rather slowly upon skewed insertions comparing with QED labels.



(a) Uniform Insertion Time (b) Label Size after Uniform Insertion

(c) Skewed Insertion Time (d) Label Size after Skewed Insertion

Fig. 4. Comparison of Uniform and Skewed Insertion for XMark data set

6.3 Query Time

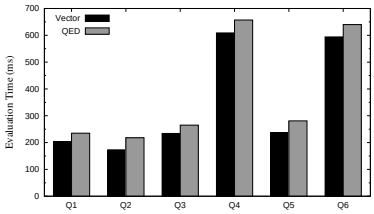
We compare the query time using all the four data sets. Here we only present the results for XMark data set due to space constraint, the other data sets show similar results. The set of queries we used are shown in Figure 5(a). Figure 5(b), (c) and (d) show the comparison of query time on the original data and the data after uniform and skewed insertion. Notice that the time used for determining A-D and P-C relationships only constitute a fraction of the whole query time. In all these cases, query time of vector labels is faster than that of QED labels. The difference is most significant for the case of skewed insertion as when the length of QED label gets larger, lexicographical order is more expensive to compare. The comparison we show here is independent of the algorithm that is used to evaluate the queries. Basically all the algorithms involves determination of A-D and P-C relationships which is more efficient to compute using vector labels.

7 Conclusion

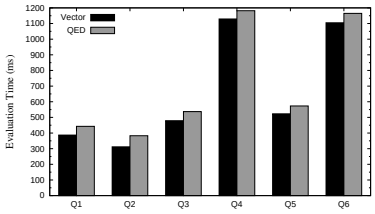
In this paper, we have proposed a novel encoding scheme: vector encoding which is easy to understand and can be applied to various existing labeling schemes to completely avoid re-labeling. We have shown how it can be applied to containment scheme, and how insertions can be optimized. Finally it is experimentally proved that vector encoding outperforms existing schemes on updates and query processing especially in the case of skewed insertion.

Query ID	Query
Q1	/site//mailbox//mail
Q2	//item//mail
Q3	//item/mailbox/mail
Q4	//item/mailbox/mail/text
Q5	//item[./mailbox]//mail
Q6	//item[./mailbox]//mail/text

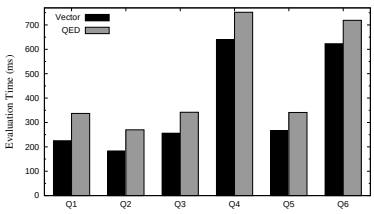
(a) Test query set



(b) Query time on original data



(c) Query time after uniform insertion



(d) Query time after skewed insertion

Fig. 5. Comparison of query response time for XMark data set

We have focused on handling insertion in this paper. Currently we are extending our work to optimize both deletion and insertion. How to control the label size in a dynamic XML document where deletion and insertion frequently occur will be an interesting topic to explore.

References

1. NIAGARA Experimental Data. http://www.cs.wisc.edu/niagara/data.html.
2. University of Washington XML Repository . http://www.cs.washington.edu/ research/xmldatasets/.
3. XMark - An XML Benchmark Project. http://monetdb.cwi.nl/xml/downloads.html.
4. S. Abiteboul, S. Alstrup, H. Kaplan, T. Milo, and T. Rauhe. Compact labeling scheme for ancestor queries. *SIAM J. Comput.*, 2006.
5. T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. In *ICDE*, 2003.
6. T. Bray, J. Paoli, C.M.Sperberg-McQueen, E.Maler, and F. Yergeau. Extensible markup language (XML) 1.0: fourth edition *W3C recommendation 2006*.
7. F.Yergeau. UTF8: A Transformation Format of ISO 10646. Request for Comments (RFC) 2279. January 2003.
8. C. Li and T. W. Ling. QED: a novel quaternary encoding to completely avoid re-labeling in XML updates. In *CIKM*, 2005.
9. X. Liang, B. Zhifeng, and T. L. Wang. A Dynamic Labeling Scheme using Vectors (Extended). http://www.comp.nus.edu.sg/ xuliang/dlsv2007.pdf.
10. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 1997.
11. X. Wu, M. L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *ICDE*, 2004.
12. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD*, 2001.