LSDX: A New Labelling Scheme for Dynamically Updating XML Data

Maggie Duong **Yanchun Zhang**

School of Computer Science and Mathematics
Victoria University
Melbourne, VIC, Australia

maggie.duong1@students.vu.edu.au

yzhang@csm.vu.edu.au

Abstract

In order to facilitate query processing for XML data, several path indexing, labelling and numbering scheme have been proposed. However, if XML data need to be updated frequently, most of these approaches will need to re-compute existing labels which is rather time consuming. In this paper, we propose a new Labelling Scheme for Dynamic XML data (LSDX) that supports the representation of the ancestor – descendant relationship and sibling relationship between nodes. Moreover, LSDX supports the process of updating XML data without the need of re-labelling existing labels, hence facilitating fast update. Some experimental works have been conducted to show its effectiveness.

Keywords: XML, XPath, XQuery, path index, query processing.

1 Introduction

In recent years, the eXtensible Mark-up Language (XML) has emerged as a popular data format for data representation and exchanging information on the web. Query languages like XPath and XQuery are developed by W3C group for XML data. XPath and XQuery are both strongly typed as declarative queries and using path expressions to traverse XML data irregularly. The traditional and most beneficial technique for increasing query performance is the creation of effective indexing. A well- constructed index will allow a query to bypass the need of scanning the entire table for results. (Kaelin, 2004). To make this possible, a unique label is assigned for each node in the XML trees in such a way that can clearly show relationship between any two given nodes (such as ancestor – descendant relationship or sibling relationship). Once this is done, structural queries can be answered by only using the index. There is no need to access to the actual documents. (Lu and Ling, 2004). In order to achieve effective indexing, several techniques have been proposed. These techniques vary from path indexing to numbering scheme. For instance, the works by Grust (2002), Amato, Debole, Rabitti and Zezula (2003), Cooper, Sample, Franklin, Hjaltason, Shadmon (2001), Meuss and Strohmaier (1999) used path indexing for searching XML data. Two other approaches are employed in *Numbering Scheme*. One is called *region-based numbering scheme* (Li and Moon 2001) and the other one is *prefix-based numbering scheme* (Cohen, Kaplan and Milo 2002, Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang 2002, and Kaplan, Milo and Shabo). All of these techniques help to facilitate query processing. However, one disadvantage of most of these works is as follows:

- If deletion and/or insertion occur regularly in the XML data, these techniques would need expensive re-computing of affected labels. (Details of this disadvantage will be discussed in the next section).

Frequently re-computing large amount of elements each time XML data is updated will take time and will reduce performance. It raises the cost of renumbering. In this paper, we propose a new labelling scheme, a LSDX that will make the following contributions:

- LSDX is a new persistent labelling scheme to label XML elements that support the demand of updating XML data without the need of re-labelling existing labels, hence facilitating fast update.
- LSDX also supports the representation of ancestor – descendant relationships and sibling relationships between nodes. One can quickly determine the relationship between any two given nodes by looking at their unique codes.
- LSDX is capable of showing the depth of the tree. Its unique codes specify the level of each node. This advantage makes the tasks of doing retrieving, inserting, deleting or updating much easier.

The rest of this paper is organized as follows:

In section 2, we shall briefly discuss related works in path indexing, and labelling schemes for XML data. Section 3 provides an overview of our proposed labelling scheme, the LSDX. Section 4 will present some experiments based on this new technique. Section 5 concludes this paper.

```
<employee>
  <person>
    <name>Lisa</name>
    <phone>98765432</phone>
  </person>
  <person>
    <name>Michael</name>
    <phone>98765332</phone>
  </person>
</employee>
```

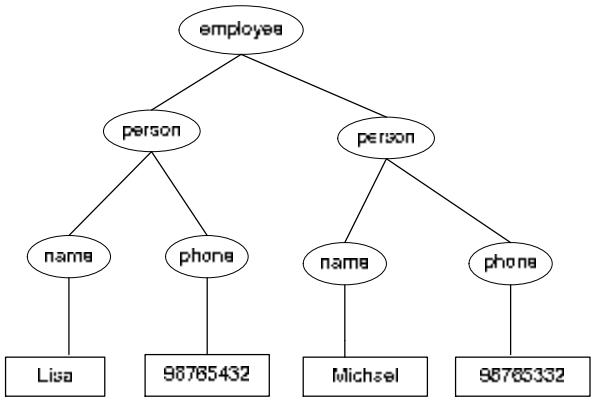

Figure 1: An example of an XML document and its data tree

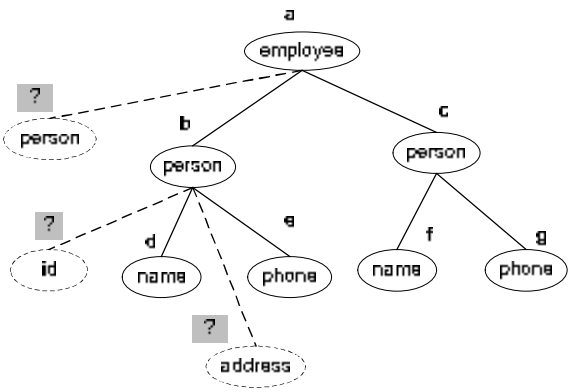

Figure 2: Example of updating problem

2 Related Work

Regarding to facilitating query processing for XML data, there are several approaches proposed such as path indexing, tree labelling and numbering scheme. Amato, Debole, Rabitti and Zezula (2003), Grust (2002), Meuss and Strohmaier (1999) used path indexing for quicker searching XML data. However, these approaches do not support updating XML data dynamically. Whenever XML data need to be updated, chances are existing indexes would need to be re-computed. The similar problem occurs with some other works (Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang 2002, Li and Moon 2001).

Cohen, Kaplan and Milo (2002) proposed two *prefix-based numbering* schemes to assign a specific code to each child of a node *v*. The first approach is *one-bit growth*. For instance, the first child's code of the node *v* is "*0*" which is labelled as *L(v).0*. The second child's code of the node *v* is "*10*" which is labelled as *L(v).10*. The third child's code is "*110*" which is labelled as *L(v).110*. Hence, the *i*th child's code is repeated with '*1*' for each child's code that ends with "*1*", together with a "*0*" attached at the end.

The second approach is *double-bit growth*. Given that u_i's code is $L(v).L'(u_i)$ where $L(v)$ is its direct parent code. It assigns its children as $L'(u_1) = 0$, $L'(u_2) = 10$, $L'(u_3) = 1100$, $L'(u_4) = 1101$, $L'(u_5) = 1110$, $L'(u_6) = 11110000$, etc. In general, it increases the binary code represented by $L'(u_i)$ by 1, that means to assign $L'(u_{i+1})$. However, if the representation of $L'(u_i)$ consists of all ones, it doubles its length by adding a sequence of zeros.

Similar technique is used by Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang (2002). It also used *Dewey prefix-based numbering* scheme, which is presented as follows:

> *Given n children of a node v, coded as L(v), u_1, u_2, u_3, ... u_n . The code of u_i is $L(u_i) = L(v).i$.* Such as *1.2.1*. See Figure 3(a).

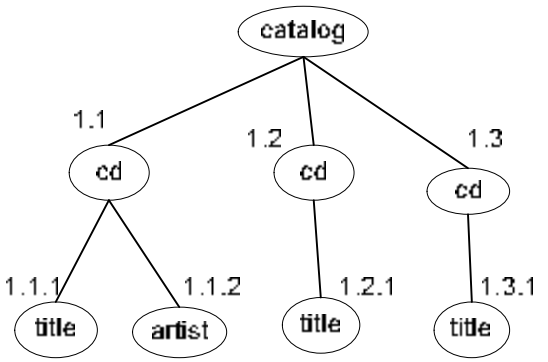

Figure 3(a): Dewey prefix-based by Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang 2002

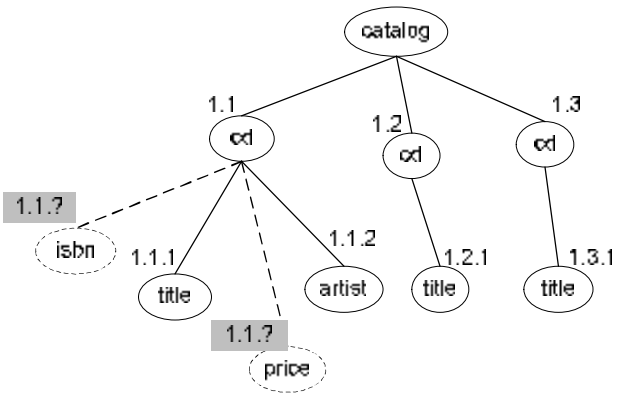

Figure 3(b): Dewey prefix – Updating problem

In Li and Moon (2001), given that a tree node y and its parent x, order(x) < order(y) and order(y) + size(y) <= order(x) + size(x). In other words, interval [order(y), order(y) + size(y)] is contained in interval [order(x), order(x) + size(x)].

It came to our knowledge that the work by Yu, Luo, Meng and Lu (2004) dynamically supports updating XML data. It proposed a *prefix-based PBi Tree* scheme which uses preserved codes between every two child nodes to reduce the possibility of renumbering all siblings and their descendants when updating is needed. This technique works as follows:

> *Suppose a node v has n child nodes, u_1, u_2, u_3 ... u_n. A unique child code is assigned to u_i using m-bits such that $2^{m-1} < n <= 2^m$, indicated as $L'(u_i)$. Let the parent node v's code be L(v), then the code of u_i is $L(u_i) = L(v).L'(u_i)$ where "." is concatenation operator.* See Figure 4(a).

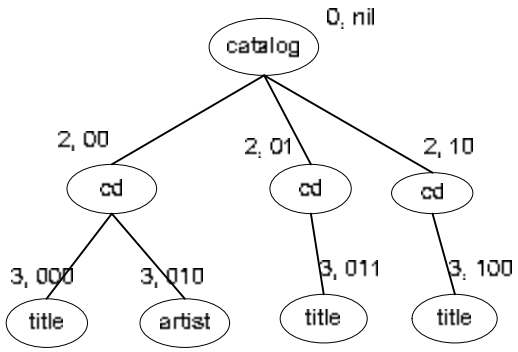

Figure 4(a): P-PBiTree by Yu, Luo, Meng and Lu 2004

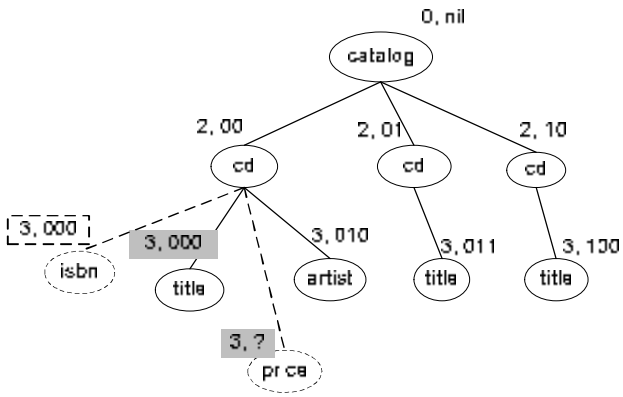

Figure 4(b): P-PBiTree –Updating problem. Shaded codes need to recompute.

Although this approach supports updating XML data, this technique is not flexible because codes must be reserved before hand. Moreover, when all reserved codes are used up, renumbering has to be done again.

In general, those mentioned above path indexing and numbering schemes can facilitate query processing to a great extent. However, eliminating the problem of re-computing when XML data need to be updated remains a challenge. In the next section, we will present our new persistent labelling scheme that will support updating XML data dynamically without the pain of re-labelling.

3 The New LSDX

Instead of using numbers such as Amato, Debole, Rabitti and Zezula (2003), or Cohen, Kaplan and Milo (2002), or Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang (2002), or Li and Moon (2001) or Yu, Luo, Meng and Lu (2004) or Grust (2002) or using letters such as Wang, Jiang, Lu and Yu (2003) to label each node, we combined numbers and letters in our labelling scheme to label XML trees. The advantages of these combinations which make our new labelling scheme compact and persistent are: (a) there's no need to re-labelling existing labels, (b) it supports the representation of the ancestor-descendant relationships and sibling relationships between nodes, (c) in addition, LSDX also presents the depth of the tree and its unique way of labelling will be of help for quick insert, delete, or update and/or start traversing at any arbitrary node. These operations can be implemented using *TreeMap* in Java which algorithms are adapted from *red-black tree* developed by Cormen, Leiserson, Rivest and Stein (2001). This shall guarantee that these basic operations take *log(n)* time in the worst case.

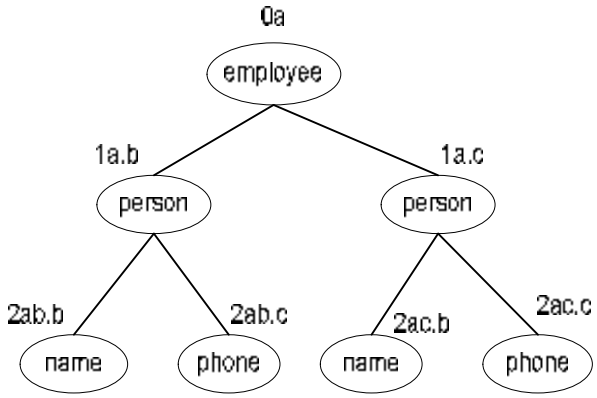

Figure 5(a): LSDX supports updating

Figure 5(b) shown below is a Path lexicon for the above XML data tree.

/employee -> 0a
/employee/person -> 1a.b, 1a.c
/employee/name -> 2ab.b, 2ac.b
/employee/phone -> 2ab.c, 2ac.c

Figure 5(b): Path lexicon

3.1 LSDX Labelling

Our new labelling scheme is demonstrated in Figure 5(a) shown above. Let's start with the document element (employee). We first give it an *"a"*. As there is no parent node for the document element, we assign *"0"* at the front of that *"a"*. *"0a"* is the unique code for the document element (employee). For the children nodes of *"0a"*, we continue with the next level of the XML tree which is *"1"* then the code of its parent node which is *"a"* and a concatenation *"."*. We then add a letter *"b"* for the first child, letter *"c"* for the second child, *"d"* for

the third child and so on. Unique codes for children nodes of *"0a"* shall be *"1a.b"*, *"1a.c"*, *"1a.d"*, etc.

From level 1 of the XML tree and downwards, we choose to use letter *"b"* rather than *"a"* for the first child of the document element because we want to save codes for any *InsertBefore* operation that might be required in the future. The concatenation *"."* is employed to help users figure out relationship between ancestor and descendant nodes. For example, just by looking at node *"1a.b"*, one will realize that it is a descendant of node *"0a"*.

There is a little difference in using the concatenation *"."* at the third (level 2) and other lower levels of the XML tree. As shown in Figure 5(a), the unique code for the first child of node *"1a.b"* is *"2ab.b"* rather than *"1a.b.b"*. We intentionally decide to name it that way because we do not want to confuse users with too many *"."* In general, it works as follows:

> Given a node *v* with *n* child nodes: $u_1, u_2, u_3 \ldots u_n$, a unique code for u_1 is a combination of its *level* + *code of its parent node* + *"."* + *"b"*. The unique code for u_2 is its *level* + *code of its parent node* + *"."* + *"c"*. The labelling continues for the rest of child nodes in alphabetical order.

3.2 Support Updating

For updating and future node insertions, Li and Moon (2001) reserve extra number spaces and Yu, Luo, Meng and Lu (2004) preserve codes. These techniques work well. However, when all reserved spaces and codes are used up and the need for updating continues to arise, affected nodes will need to be recomputed. To overcome this problem, we use a combination of numbers and letters to create unique codes for XML data.

The concept of using letters is similar with Figure 2. Using letters from *a* to *z* to label from the root to the child nodes shall keep order and shall give a faster and easy access to a specific node. The difference is the flexible way we use to generate unique codes for every existing node and also for new nodes, which shall be needed for future updating. The rule for generating unique codes for new nodes is described below.

Rule for generating labels for new nodes: If there is no node standing before the place that a new node shall be added, get the code of the node standing after the new node and insert *"a"* after the *"."*. Otherwise, keep counting from the code standing before it so that the code for the new node will be greater than the code of its previous sibling and less than the code of its next sibling (if have) in alphabetical order. If previous code ends with *"z"*, attach *"b"* at the end.

The following four operations are supported by LSDX to update XML data.

1. **InsertBefore** (ST, N). Insert a sub tree 'ST' into an existing XML tree before a node 'N'. See Figure 6(a) – (b).

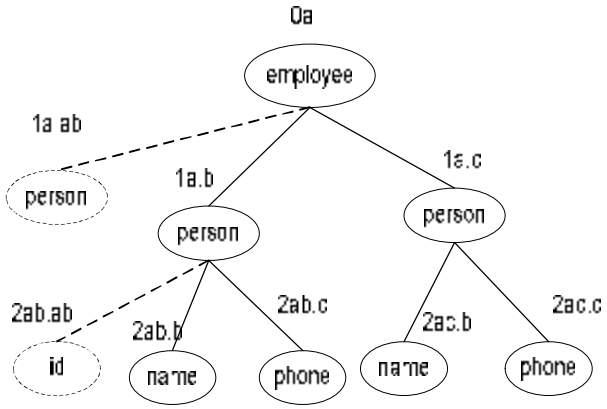

Figure 6(a): Inserting Before

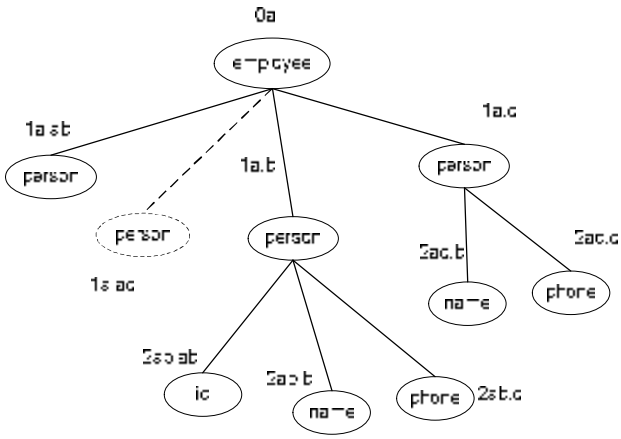

Figure 6(b): Inserting Before

With *InsertBefore* operation, one can insert a single node or a sub tree before any given node. For instance, Figure 6(a) - (b) shows inserted nodes with dot lines. If we want to add a node before the node *"1a.b"*, we will just follow the *Rule for generating labels for new nodes* to do this. In this case, there is no node before *"1a.b"* thus we get the code of this node, then add *"a"* after the *"."*. Thus, the code for the new node shall be *"1a.ab"*. If this is a sub tree, all of its children will have *"2aab."* attach at front then a letter *"b"* for the first child, *"c"* for second child, *"d"* for the third child and so on.

The need for future insertion might continue to arise. For example, if we need to insert another new node before the node *"1a.ab"*, the unique code for the new node will be *"1a.aab"*. Nodes from *"1a.aab" to "1a.aaz*" can be used when more insertion are needed. This technique can be utilized over and over again.

2. **InsertAfter** (ST, N). Insert a sub tree 'ST' into an existing XML tree after a node 'N'. See Figure 7(a) – (b).

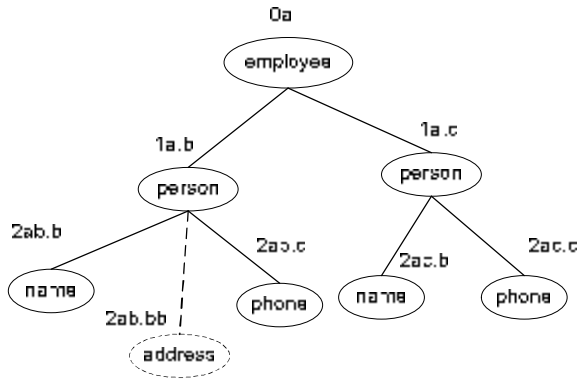

Figure 7(a): Inserting After

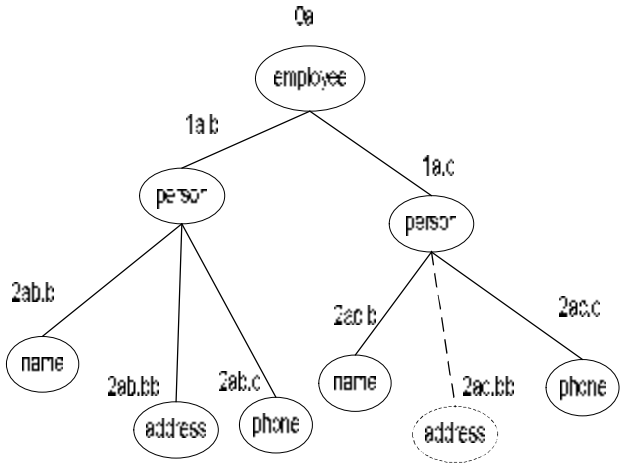

Figure 7(b): Inserting After

With the same spirit as *InsertBefore* operation, *InsertAfter* operation can be used to insert a single node or a sub tree after any given node. Examples are given in Figure 7(a) - (b) with dot lines. If we want to add a new node after the node *"2ab.b"*, we just follow the *Rule for generating labels for new nodes* to generate the unique code for it. In this case, there is a node standing before *"2ab.c"* which is *"2ab.b"*. Thus, we need to continue counting from *"2ab.b"* to generate a code, which shall be less than *"2ab.c"*. The code for the new node will be *"2ab.bb"*. If another new node is needed for insertion after *"2ab.bb"*, its code will be *"2ab.bc"* and shall continue up to *"2ab.bz"*, *"2ab.bzb"* to *"2ab.bzz"*, etc.

3. **Delete** (ST). Delete a sub tree 'ST' from the existing XML tree. See Figure 8.

This operation can be used to delete a single node or a sub tree from the tree. Deleting node/s is quite simple comparing to inserting node/s because generating code is not needed. Code of deleting node/s could be used again when a new node is inserted in its place.

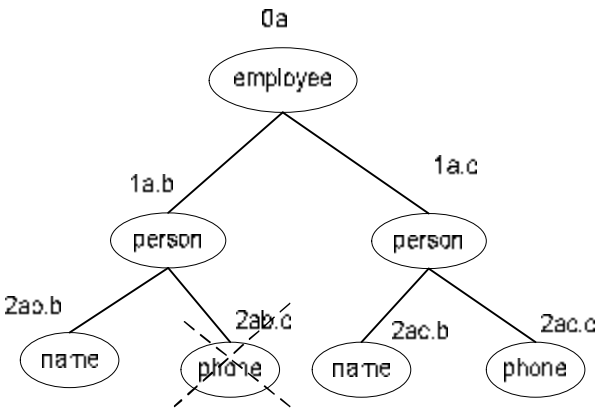

Figure 8: Deleting

4. **Update** (V, N). Update the content of a node "N" with a value "V". See Figure 9.

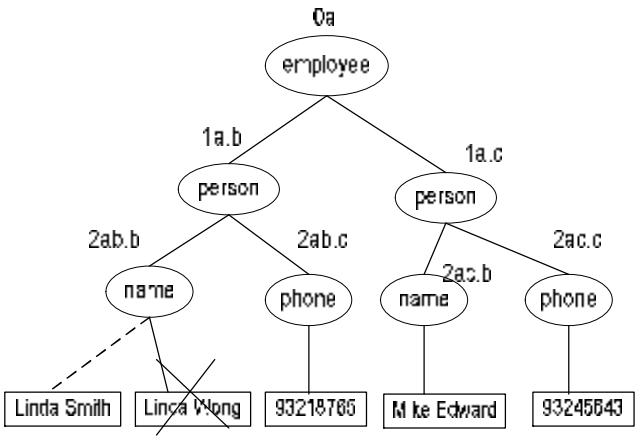

Figure 9: Updating

The content of a node can be updated using this *Update* operation. An example of this is shown in Figure 9 for changing the surname. This operation does not require the need for generating labels.

3.3 Ancestor-Descendant Relationship

Using codes of the parent nodes as a part of creating codes for child nodes helps to determine the ancestor – descendant relationships and the sibling relationship between nodes. For instance, in Figure 5(a), by knowing a node called *"1a.b"*, we can understand that its parent is *"0a"* and all nodes beginning with "*1"* are its siblings. All of its children nodes shall have *"2ab."* attached at front.

3.4 Tree Depth

Another helpful feature in our new labelling scheme is that one can tell the depth (level) of the tree. This can be possible because we add the level number as the first character when we assign unique codes for each node. Moreover, when this labelling scheme is implemented, using level numbers as explained shall help ones quickly gain access to a specific level. Similarly, using unique codes shall help one to get to any specific node.

Therefore, our labelling scheme will help to reduce the number of nodes that need to be accessed to carry out those tasks. These advantages make the tasks of doing retrieving, inserting, deleting or updating a lot easier.

4 Experiments

We have implemented our proposed LSDX labelling scheme in Java 1.4.2 and used SAX from Sun Microsystems as the XML parser. For the database, we used XMark datasets (Schmidt, Waas, Kersten, Carey, Manolescu and Busse 2002) to generate XML documents. We used various scaling factors (0.005 – 0.5) to create from 100KB to 57000KB of data. We chose XMark dataset because it generates a standard, balanced XML document and it is typically encountered in real-world scenarios.

An advantage of our proposed labelling scheme is that it supports updating XML data dynamically without the need of re-labelling existing labels. We have run some experimental works and compared our works with the works by Lu and Ling (2004) and Cohen, Kaplan and Milo (2002) whose labelling techniques are known to support dynamic XML data. Our experiments were performed on the Pentium IV 2.4G with 512MB of RAM running on windows XP with 25G hard disk.

4.1 Length of Labels

Our first experiment was carried out to consider the total length of our proposed labels to others. We have studied the experiment of GRP by Lu and Ling (2004) and SP by Cohen, Kaplan and Milo (2002), which supports dynamic XML data. We used XMark to generate the first ten XML documents based on scaling factors (0.01 – 0.1), same as that used by Lu and Ling (2004). We then generated labels from those documents using our proposed LSDX labelling scheme and compared with those two schemes in term of total length of labels. We discovered that our labelling scheme can be two times shorter compared to GRP scheme and more than 7 times shorter compared to SP scheme. Detailed figures are presented in the Table 1 below.

4.2 Time Used to Generate Labels

Our second experiment was to test how long it takes to generate labels for various datasets that were generated above. We passed an XML document in our Java program, using SAX as a parser, and then concurrently generated labels for it. We noticed that time needed for generating labels varies from 1 second for 1.2MB of data to 28 seconds for 50MB of data. Approximately, it will take one minute to generate 100MB of data. Some results of this experiment are displayed in the Table 2.

XML Doc (MB)	**No of Nodes (K)**	**Total length of labels (MB)**		
		LSDX	**GRP scheme** (Lu, 2004)	**SP scheme** (Cohen, 2002) Figure by (Lu, 2004)
1.2	17	**0.17**	0.64	1.36
2.3	33	**0.41**	1.20	4.64
3.4	50	**0.76**	2.00	10.24
4.7	68	**1.17**	2.72	17.92
5.6	84	**1.63**	3.44	27.04
6.9	100	**2.21**	4.24	39.20
8.0	118	**2.91**	5.04	54.08
9.2	134	**3.60**	5.92	69.12
10.3	151	**4.43**	6.80	88.00
11.4	167	**5.29**	7.60	107.2

Table 1: Total length of labels - A comparison between our LSDX, GRP and SP labelling schemes

Scaling Factor	XML Doc (MB)	No of Nodes (K)	Time (Sec)
0.01	1.2	17	**1.09**
0.02	2.3	33	**1.56**
0.03	3.4	50	**2.25**
0.04	4.7	68	**2.79**
0.05	5.6	84	**3.28**
0.06	6.9	100	**3.90**
0.07	8.0	118	**4.54**
0.08	9.2	134	**5.09**
0.09	10.3	151	**5.59**
0.1	11.4	167	**6.18**

Table 2: Time (in sec) used to generate labels using LSDX

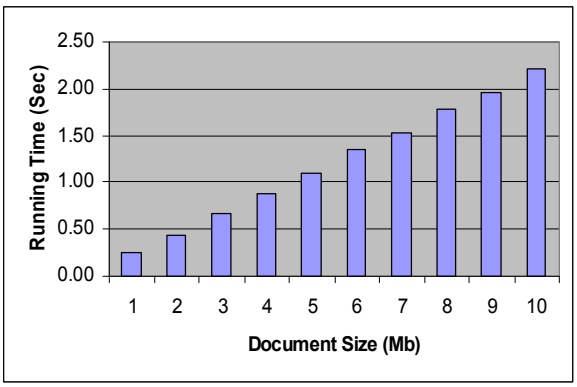

Figure 10(a): Time (in sec) used to insert single nodes

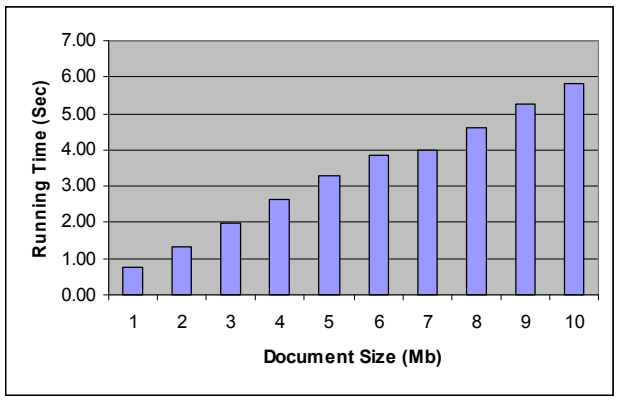

Figure 10(b): Time (in sec) used to insert sub trees

4.3 Insertion and Deletion Time

We did some experiments on inserting single nodes and inserting sub trees to the mentioned above XML documents. Result of time used is shown in Figure 10(a). In this experiment, we first generated a unique label for each of the nodes, and then added it to the XML tree. We also updated the XML data and saved it to the file straight way. While committing all the changes can be done at last to save a great deal of time, we chose to save each change individually merely for the purpose of knowing how much time is needed to commit inserting node/s and sub tree/s operation to the database.

The performance of inserting single nodes is spectacularly quick, shown in Figure 10(a). Inserting sub trees shown in Figure 10(b) took a little bit longer compared to inserting single nodes. A comparison of time used for these operations is shown in Figure 10(c).

We also did some experiments on deleting single nodes and deleting sub trees from the XML documents above. We first removed the node/s from the XML tree and then updated the XML file so that it is permanently removed. The running time of deleting a single node and deleting a sub tree and are not much difference. Some experiments of this are shown in Figure 10(d).

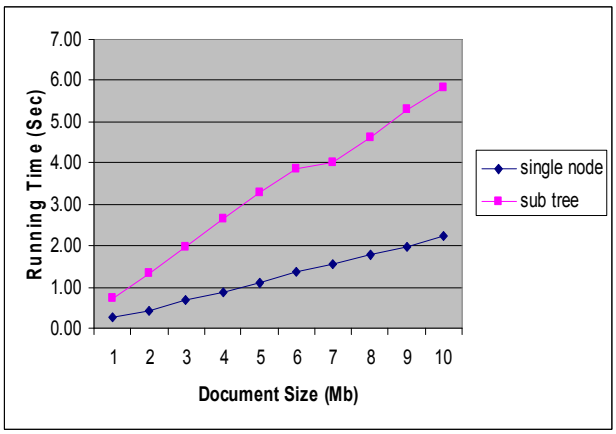

Figure 10(c): Comparison between inserting single nodes and sub trees

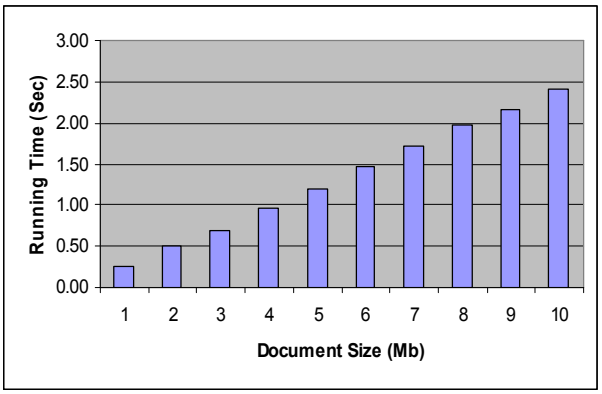

Figure 10(d): Time (in sec) used to delete nodes

As mentioned previously, we are only interested in finding out how much time it takes to permanently remove nodes. Thus, we did the commit straight away so that deleting nodes will be immediately removed. Our experiments shown that deleting nodes took less time than inserting sub trees because we do not have to care about generating labels.

5 Conclusion

In this paper, we present the new LSDX labelling scheme that supports updating XML data dynamically without the need for re-labelling, hence facilitating fast update. LSDX also supports the representation of the ancestor – descendant relationships and sibling relationships between nodes. Our experiments showed that our labelling scheme is about two times shorter in term of total length of labels comparing to GRP (Lu and Ling, 2004) and about 7 - 18 times shorter comparing to SP scheme (Cohen, Kaplan and Milo, 2002) - (see Table 2). Generating labels for XML documents vary from 1 second for 1.2MB of data to one minute to generate 100MB of data. In term of permanently storing individual changes in the files, time used for insertion and deletion are considered spectacularly quick. This will be useful when two or more programs need to use the same XML data.

In the future, we shall develop an XML query tool with a graphical user interface based on our proposed labelling scheme.

6 References

Alstrup, S. and Rauhe, T. (2002): Improved Labelling Scheme for Ancestor Queries. In proceedings of the 13th annual *ACM-SIAM Symposium on Discrete Algorithm*, 2002.

Amato, G., Debole, F., Rabitti, F. and Zezula, P. (2003): Yet Another Path Index for XML Searching, in Proceedings of ECDL 2003. *Research and Advanced Technology for Digital Libraries,* 7th European Conference, Trondheim, Norway, 2003.

Boag, S., Chamberlin, D., Fernández, M., Florescu, D., Robie, J. and Siméon, J. (2003): XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

Boag, S., Chamberlin, D., Robie, J., Florescu, D., Siméon, J. and Fernandez, M. (2003): XQuery 1.0: An XML Query Language. Nov 2003. http://www.w3.org/TR/2003/WD-xquery-20031112/

Chamberlin, D., Robie, J. and Florescu, D. (2000): Quilt: An XML Query Language, Mar 2000. http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html

Cohen, E., Kaplan, H. and Milo, T. (2002): Labelling dynamic XML trees, in Proceedings of *PODS* 2002.

Cooper, F. B., Sample, N., Franklin, J. M., Hjaltason, R. G., and Shadmon, M. (2001): A Fast Index for Semistructured Data, in Proceedings of *VLDB Conference*, 2001.

Cormen, H. T., Leiserson, E. C., Rivest, R.L. and Stein, C. (2001) Introduction to Algorithms. Ch 13, Second Edition 2001.

Derksen, E., Fankhauser, P., Howland, E., Huck, G., Macherius, I., Murata, M., Resnick, M. and Schöning, H. (1999): *XQL (XML Query Language)*. http://metalab.unc.edu/xql/xql-proposal.xml

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1998): XML-QL: A Query Language for XML. http://www.w3.org/TR/1998/NOTE-xml-ql-19980819

Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M, Siméon, J. and Wadler, P. (2004): XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/2004/WD-xquery-semantics-20040220/

Grust, T. (2002): *Accelerating XPath Location Steps,* in Proceedings of the 2002 ACM SIGMOD

International Conference on Management of Data, Madison, Wisconsin, ACM 2002.

Hou, J., Zhang, Y. and Kambayashi, Y. (2001): Object-Oriented Representation for XML Data, Proceedings of the 3rd International Symposium on Cooperative Database Systems for Advanced Applications *(CODAS'2001),* April 23-24, 2001 in Beijing, China, IEEE CS Press.

Kaelin, M. (2004): Database Optimization: Increase query performance with indexes and statistics, TechRepublic, http://techrepublic.com.com/5100-6313_11-5146588.html?tag=search.

Kaplan, H., Milo, T. and Shabo, R: *A* Comparison of Labelling Schemes for Ancestor Queries. http://www.math.tau.ac.il/~haimk/papers/comparison.ps

Lee, K.Y., Yoo, S. J. and Yoon, K. (1996): *Index structures for structured documents*, in ACM First International Conference on Digital Libraries, p 91-99, Bethesda, Maryland, March 1996.

Li, Q. and Moon, B. (2001): *Indexing and Querying XML Data for Regular Path Expressions,* in Proceedings of VLDB 2001.

Lu, J. and Ling, W. T. (2004): Labelling and Querying Dynamic XML Trees, in Proceedings of 6th Asia Pacific Web Conference, APWeb 2004, China.

Meuss, H. and Strohmaier, M. C. (1999): *Improving Index Structures for Structured Document Retrieval.* In 21st BCS IRSG Colloquium on IR, Glasgow, 1999.

Schmidt, A., Waas, F., Kersten, M.,Carey, J. M., Manolescu, I. and Busse, R. (2002): *XMark: A Benchmark for XML Data Management,* in Proceedings of VLDB 2002.

Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E. and Zhang, C. (2002): *Storing and Querying Ordered XML Using a Relational Database System*, in Proceedings of SIGMOD 2002.

Wang, W., Jiang, H., Lu, H. and Yu, X. J. (2003): *PBiTree Coding and Efficient Processing of Containment Joins.* In 19th International Conference on Data Engineering, 2003 Bangalore, India.

Yu, X. J., Luo, D., Meng, X. and Lu, H. (2004): *Dynamically Updating XML Data: Numbering Scheme Revisited,* in World Wide Web: Internet and Web Information System, 7, 2004.